Что такое хороший код?

Геннадий Ковалёв

Различные точки зрения

Производительность

Обеспечивает максимально возможную скорость работы. Иногда самый быстрый код выглядит ужасно: невозможно прочесть, нарушены устоявшиеся принципы разработки.

Красота

Я ещё и так могу! Вызывает эстетическое удовольствие, как произведение искусства. Может быть плохо понятен начинающим разработчикам.

Гибкость

Качественно проработаны функции, связи. Чётко следует продуманной архитектуре. Легко производить рефакторинги.

Тестируемость

Просто писать юнит-тесты без необходимости установки изделий поддержки опорно-двигательного аппарата.

Сопровождаемость

Код легко читается. Внесение изменений не составляет особого труда. Прогнозы необходимых изменений довольно точные.

Ещё 100 аспектов...

В зависимости от языка программирования, отрасли, стиля, привычек и корпоративной традиции.

Для кого мы программируем

Для конечных потребителей

Заказчики, посетилели сайта, пользователи продуктов.

Для тестировщиков QA

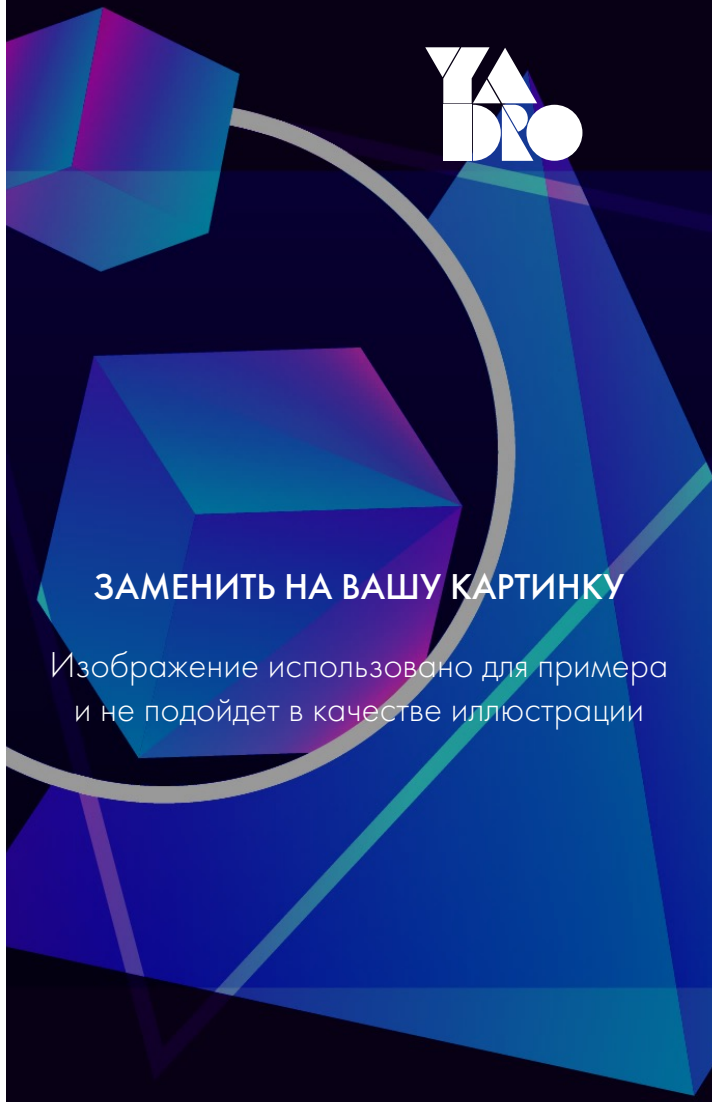
Иженеры отдела тестирования и контроля качества.

Для сервис-инженеров

Сотрудники технической поддержки, сервисного обслуживания и т.д.

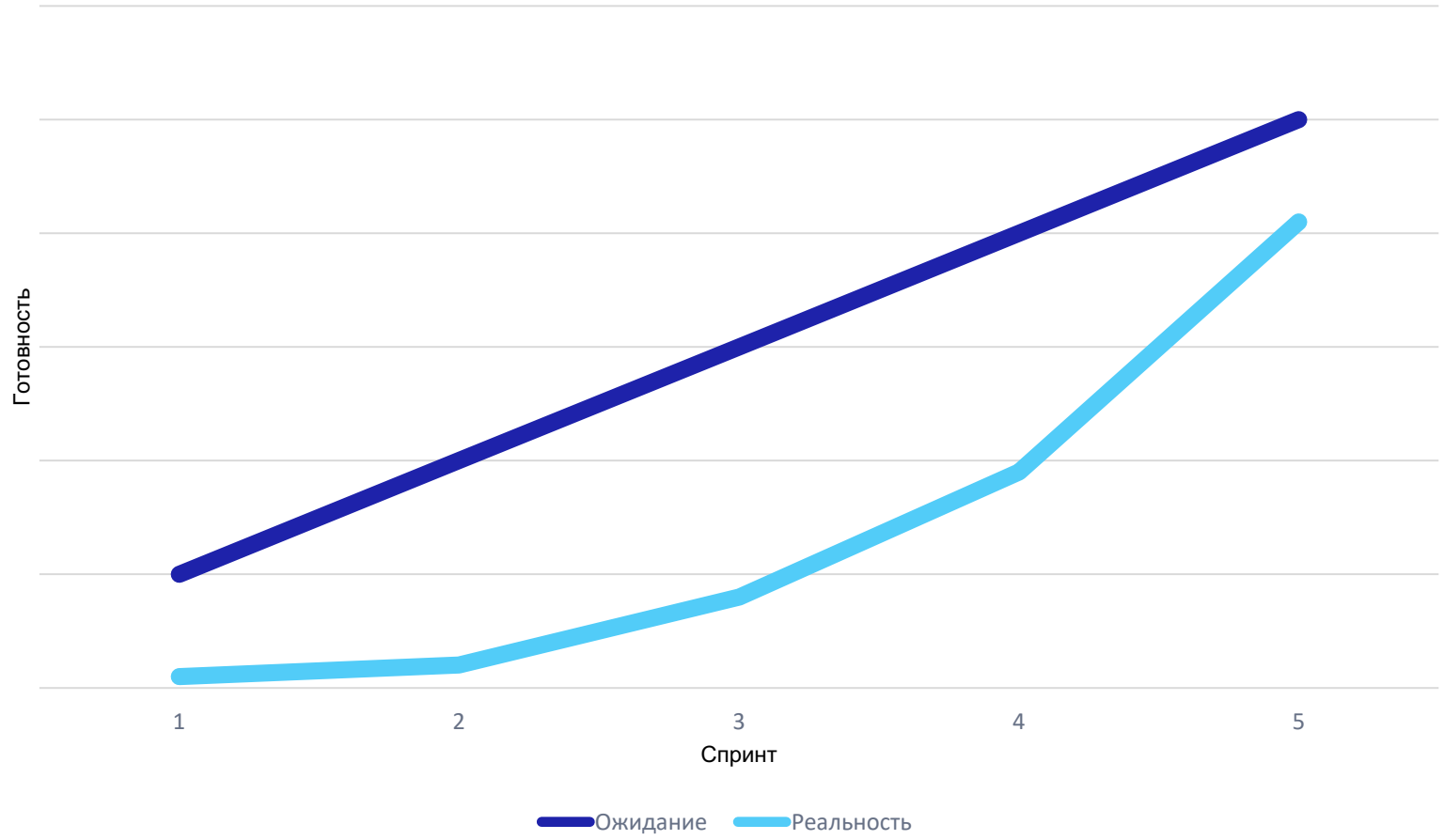
Для себя и коллег

Разработчики, которые разбираются в Вашем коде. Мы сами через некоторое время.



Чего хочет менеджмент

Добавить график ожидание/реальность

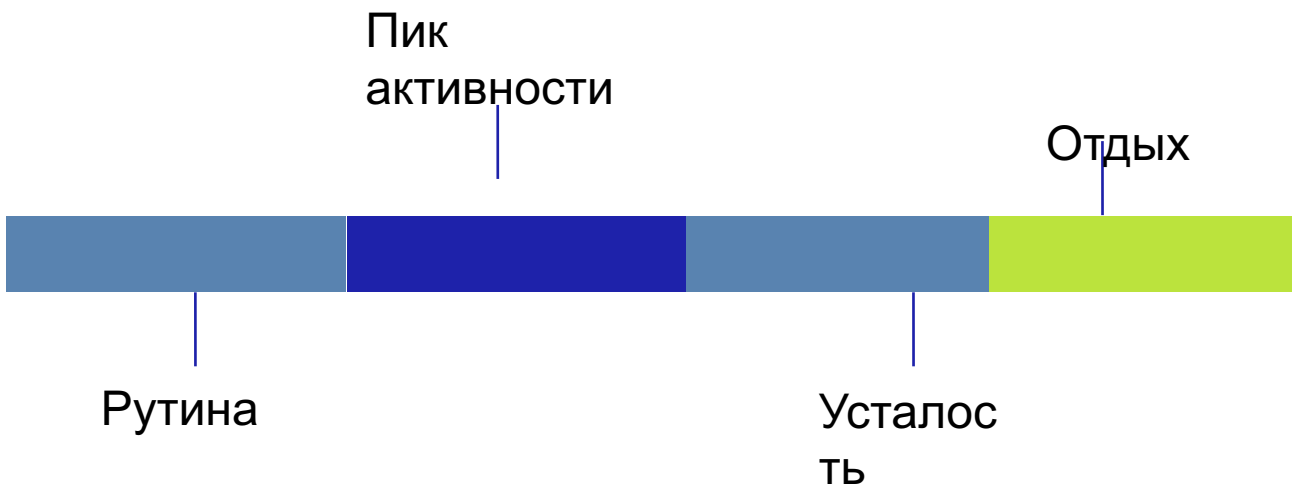


Предсказуемые сроки разработки. Сбывающиеся прогнозы.

Быстро, дёшево и качественно!

Сохранить кадры, уменьшить текучесть.

Что ест время разработчика



Время не линейно

Ценность времени разная. Отвлекаться во время самой активной разработки не тоже самое, что отвлекаться во время релакса.

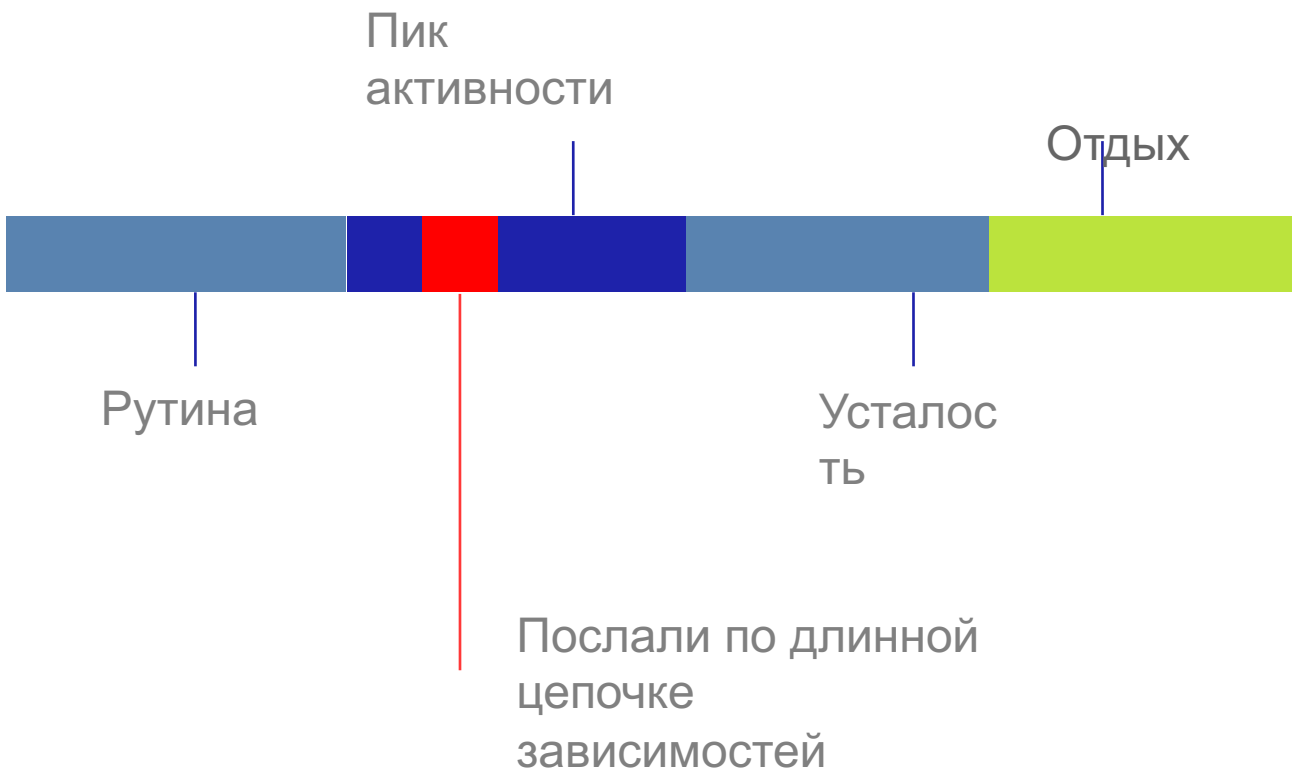
Внешние факторы

- Встречи, совещания
- Вопросы коллег
- ...

Факторы кода

- Найден баг
- Длинные цепочки зависимостей
- Невозможность создать юнит-тест
- Плохой стиль
- Острая жажда рефакторинга
- Ожидание CI, прогонов тестов и т. д.
- Необходимость согласовать дизайн
- Узнать почему так сделано

Что ест время разработчика



Время не линейно

Ценность времени разная. Отвлекаться во время самой активной разработки не тоже самое, что отвлекаться во время релакса.

Внешние факторы

- Встречи, совещания
- Вопросы коллег
- ...

Факторы кода

- Найден баг
- Длинные цепочки зависимостей
- Невозможность создать юнит-тест
- Плохой стиль
- Острая жажда рефакторинга
- Ожидание CI, прогонов тестов и т. д.
- Необходимость согласовать дизайн
- Узнать почему так сделано

Ну а чего же Go?

Синтаксис

Нейминг

Документация

Тестирование

Архитектура

Убран холивар на тему скобочек ;)

```
#include <stdio.h>

// Main func.
int main()
{
  printf("Hello world\n");
  return 0;
}
```

VS

```
#include <stdio.h>

/* Main func */
int main() {
  printf("Hello world\n");
  return 0;
}
```

gofmt

Gofmt's style is no one's favorite, yet gofmt is everyone's favorite.

Rob Pike

Gofmt должен не кому-то нравиться, gofmt обязан нравиться всем.

Роб Пайк

gofmt

> - I want to move a braces...
> - Who care? Shut up!
>
> Rob Pike.

- Я хочу передвинуть скобки...
- Давай поговорим об этом в другой раз.

Роб Пайк.

Синтаксис

Нейминг

Документация

Тестирование

Архитектура

Нейминг

Наименование переменной состоит из двух слов: названия пакета и самого имени.

```
|- namespace/
|
|-- namespace.go
|
|-stor/
|
|-- storage.go
|
|- go.mod
```

```
// CreateNS создаёт новое пространство имён.
func CreateNS(name string) (*munn.Namespace, error) {
ns, err := namespace.New(name)
if err != nil {
                    return nil, err
}
return ns, nil
}
```

Нейминг

Существует ряд договорённостей по поводу наименования переменных.

```
// Суффикс -er.
type Reader interface {
Read()
}
```

```
// Префикс Get не ставится.
type Interactor interface {
Namespace()
SetNamespace()
DelNamespace()
}
```

Нейминг

Старайтесь в разнык местах модуля одинаковые сущности называть одинаково.

```
// New создаёт новое пространство имён
// с уникальным идентификатором.
func New(name string) (*munn.Namespace, error) {
x := munn.Namespace{
                Name: name,
                ID:   uuid.New(),
}
return &x, nil
}
```

```
// CreateNS создаёт новое пространство имён.
func CreateNS(name string) (*munn.Namespace, error) {
ns, err := namespace.New(name)
if err != nil {
                return nil, err
}
return ns, nil
}
```

Синтаксис

Нейминг

Документация

Тестирование

Архитектура

Документация

Короткое предложение с документацией сильно экономит время.

```
// CreateNS создаёт новое пространство имён.
func CreateNS(name string) (*munn.Namespace, error) {
ns, err := namespace.New(name)
if err != nil {
        return nil, err
}
return ns, nil
}
```

```
func New(name string) (*munn.Namespace, error)

New создаёт новое пространство имён с уникальным идентификатором.
```

Синтаксис

Нейминг

Документация

Тестирование

Архитектура

Тестирование

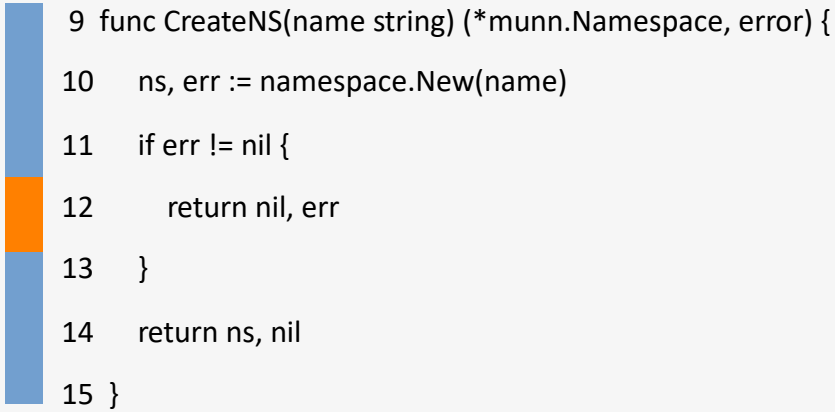
Юнит-тесты и интеграционные/e2e тесты лучше разделять.

```
|- test/
|
|-- namespace_test.go
|
|-stor/
|
|-- storage.go
|
|-- storage_test.go
|
```

```
func TestCreateNS(t *testing.T) {
_, err := CreateNS("along")
if err != nil {
                t.Errorf("Что-то пошло не так: %s.", err.Error())
}
}
```

Покрытие кода

При сохранении файла можно сразу запускать тесты и смотреть покрытие.



```
 8  // CreateNS создаёт новое пространство имён.
 9  func CreateNS(name string) (*munn.Namespace, error) {
10      ns, err := namespace.New(name)
11      if err != nil {
12          return nil, err
13      }
14      return ns, nil
15  }
```

Синтаксис

Нейминг

Документация

Тестирование

Архитектура

Что ожидаем при сопровождении?

Взгляда сверху

Необходимо понимать общую картинку, в которой работает сервис. Какой элемент для чего нужен. Из документации достаточно крупной схемки, например контекста и контейнеров из нотации C4.

Легко найти точку входа

Для начала выполнения задачи здорово было бы здорово найти точку, от которой начать раскручивать задачу, а также найти цепочку выполнения.

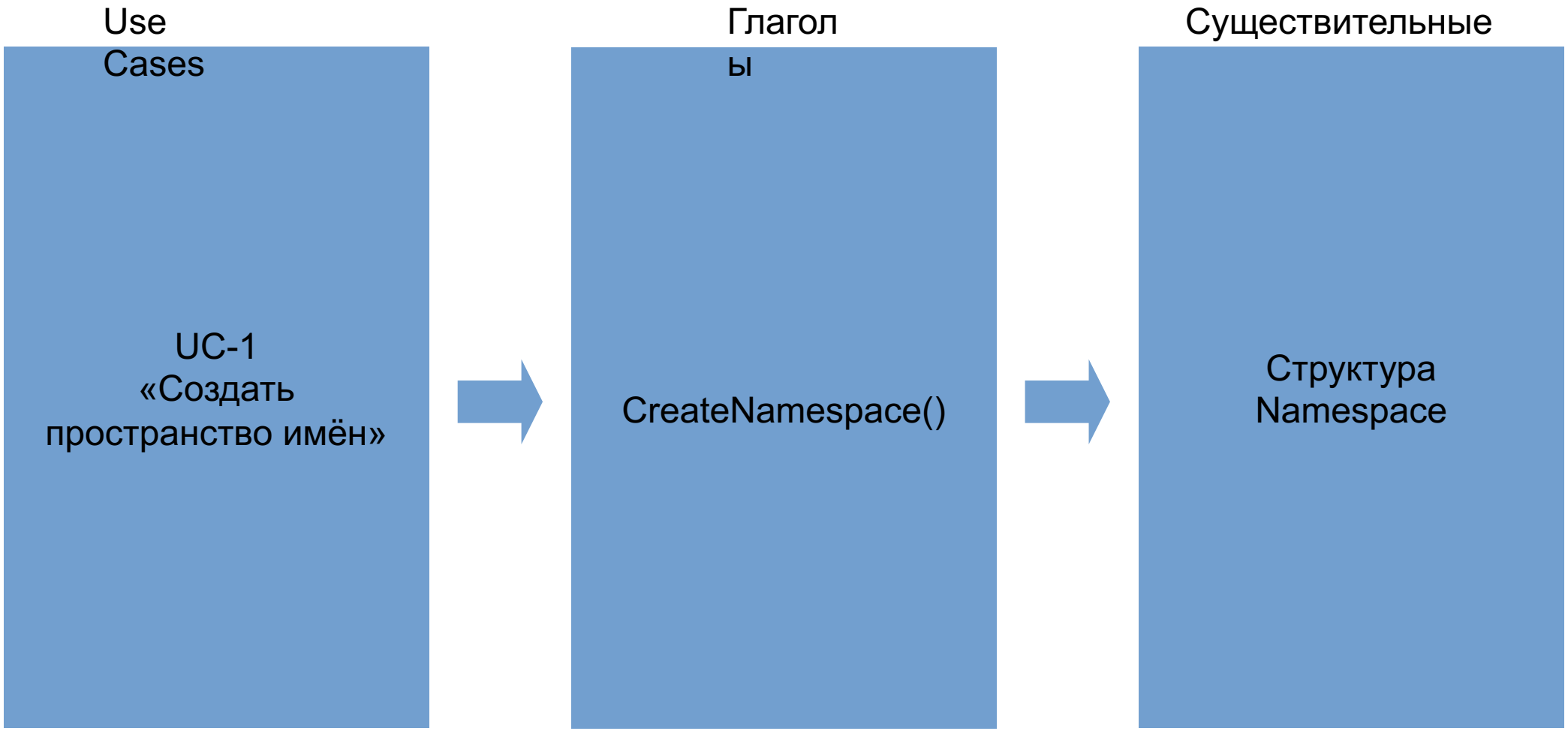
Короткой цепочки зависимостей

При отладке задачи необходимо

Быстрого исполнения тестов

Sed

Элементы архитектуры



Элементы архитектуры

Пример. Use case — создать пространство имён. Запрос попадает в handler, в нём нам легко найти функцию CreateNS, затем в сервис, там также легко найти функцию CreateNS.


```
|- namespace/
|
|-- namespace.go
|
|-storage/
|   |- handler.go
|   |- service.go
|
|- namespace.go
|- go.mod
```

```
// CreateNS обрабатывает запрос создания нового пространства имён.
func (hdlr *Handler) CreateNS(w http.ResponseWriter, r *http.Request) {
ns, err := hdlr.storSvc.CreateNS(name)
if err != nil {
                w.WriteHeader(http.StatusInternalServerError)
}
w.Write([]byte(“Ok”))
}
```

Элементы архитектуры

При ревью очень внимательно надо смотреть с точки зрения изменения существительных. Их рефактирить сложно.

Глаголы же рефакторить легко.

ГЛАГОЛЫ

Существительные

Интерфейсы

Интерфейс описывает то, что нужно здесь и сейчас, а не то, что предоставляется. Пример: здесь нам нужен только CreateNS().

```
type NSClient struct {

}

// NS возвращает...
// CreateNS создаёт...
// UpdateNS редактирует...
// DeleteNS удаляет.
```

```
type NSClient interface {
CreateNS(name string) (*Namespace, error)
}
func TestCreateNS(t *testing.T) {
_, err := cli.CreateNS("along")
if err != nil {
        t.Errorf("Что-то пошло не так: %s.", err.Error())
}
}
```

Итого: рабочее пространство

Стиль привычный

Минимум документации прямо в IDE

Зависимость разорвали

```
type DB interface {
Find()
Store()
}
```

```
func ToDebug() {
   // ...
}
```

Продуманная архитектура

Спасибо!

go.dev/doc/effective_go

Yadro.com

@GennadyKovalev - телеграм