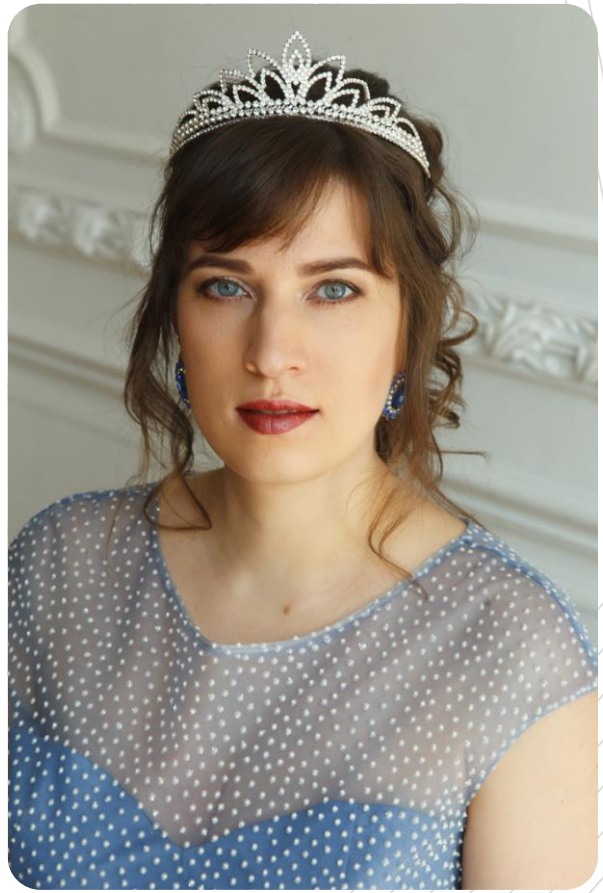
Практики и техники работы с Legacy-кодом

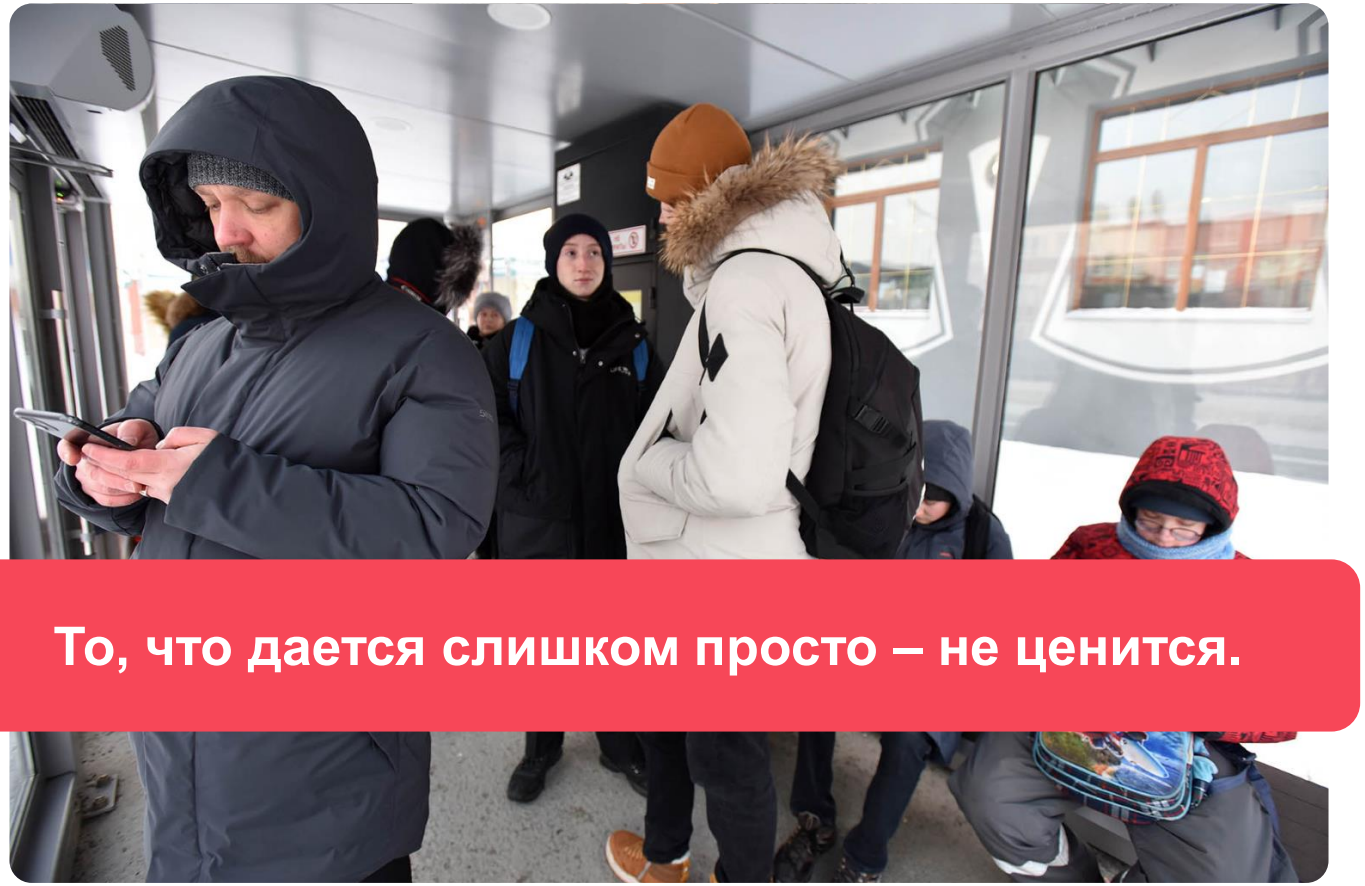
Елена Щелкунова

Обо мне

- Елена Щелкунова
- Работаю программистом с 2010 года
- Работала в 7 фирмах за это время
- Full-stack разработчик (C# / JavaScript – JQuery, ExtJs, React)
- Много работала с legacy-кодом в крупных проектах



Обещание себе



То, что дается слишком просто – не ценится.

Идеальный код

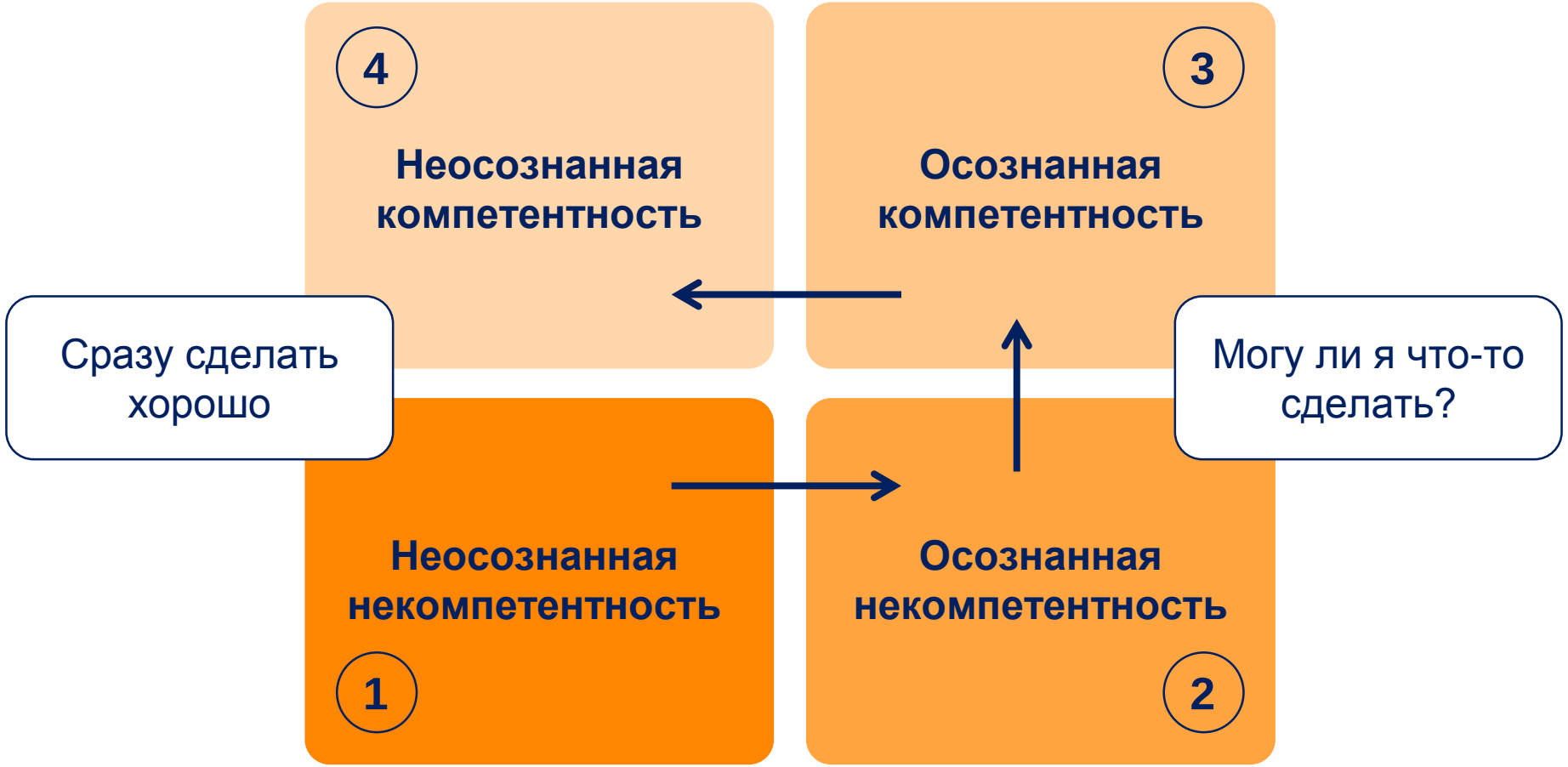
Другие просто
не знают, что делают

Со мной точно так
не будет

В реальности

- Сотни проектов
- Сотни тысяч - миллионы строк кода
- Десятки тысяч файлов с исходным кодом
- Богатая функциональность, которую целиком не знает ни один человек в фирме

Стадии развития специалиста



С какими сложностями я столкнулась?

- Очень много кода
- Мало специалистов, которые хорошо знают, что делает этот код
- Постоянно появляются новые требования, «чтобы не сломать старое, но сделать еще что-то»
- «старое» не задокументировано → рефакторинги сложны и опасны

Первое решение

А давайте просто это выкинем, и новое напишем с нуля

Идея на миллион (и хорошо если только один).

Что по итогу?

- Я видела, как эта идея «взлетела».

По итогу:

- Более 2х лет разработки, много хороших специалистов было задействовано, поддержка «старого» продукта встала. Сроки релиза нового все оттягивались.
- Новый продукт вышел, но по функциональности был беднее «старого», задержка выхода – более чем в 4 раза относительно изначальной самой оптимистичной оценки.
- Возможно, это не единственно возможный исход?

А если не переписывать?

- Придется работать (разбираться в том, что есть)
- Придется понимать логику автора, искать требования, собирать по крупицам рассказы, передающиеся из уст в уста.
- Именно этим путем я и шла и этим опытом
  и хочу поделиться сегодня

Эквивалентные преобразования

$$\frac{3\bullet\left(\sin\left(\frac{\pi}{2}+x\right)+\cos(\pi+x)+\sin(\pi-x)\right)}{\sin x}=$$

$$=\frac{3\bullet\left(\cos(x)-\cos(x)+\sin(x)\right)}{\sin x}=$$

$$=\frac{3\bullet\sin x}{\sin x}=3$$

Legacy-код - это

- Достался «в наследство»
- Не покрыт тестами
- Сложная логика, которую возможно никто досконально не знает, а документации нет или крайне мало

```
public static class ModuleMetadataHelper
{
  public static bool IsHidden(ModuleMetadata metadata)
  {
    if (!WebLicenseHelper.IsVisibleByLicense(metadata.GetOriginal()) || !metadata.IsVisible
                      || metadata.IsSolutionMetadata())
      return true;

    var foldersMetadata = metadata.GetAllSpecialFolderMetadata().Union(metadata.ExternalFolders)
      .Where(f =>
        f.Kind == SpecialFolderKind.Computable &&
        f.IsShow &&
        f.DisplayType == FolderDisplayType.Module)
      .ToList();

    var folders = SpecialFoldersCache.GetComputableFolders(foldersMetadata)
              .Where(f => f.SpecialFolderType.HasValue);
    var entitiesMetadata = ModuleItemVisibilityChecker.GetVisibleEntitiesMetadata(metadata);
    return !(folders.Any() || entitiesMetadata.Any());
  }
.....
```

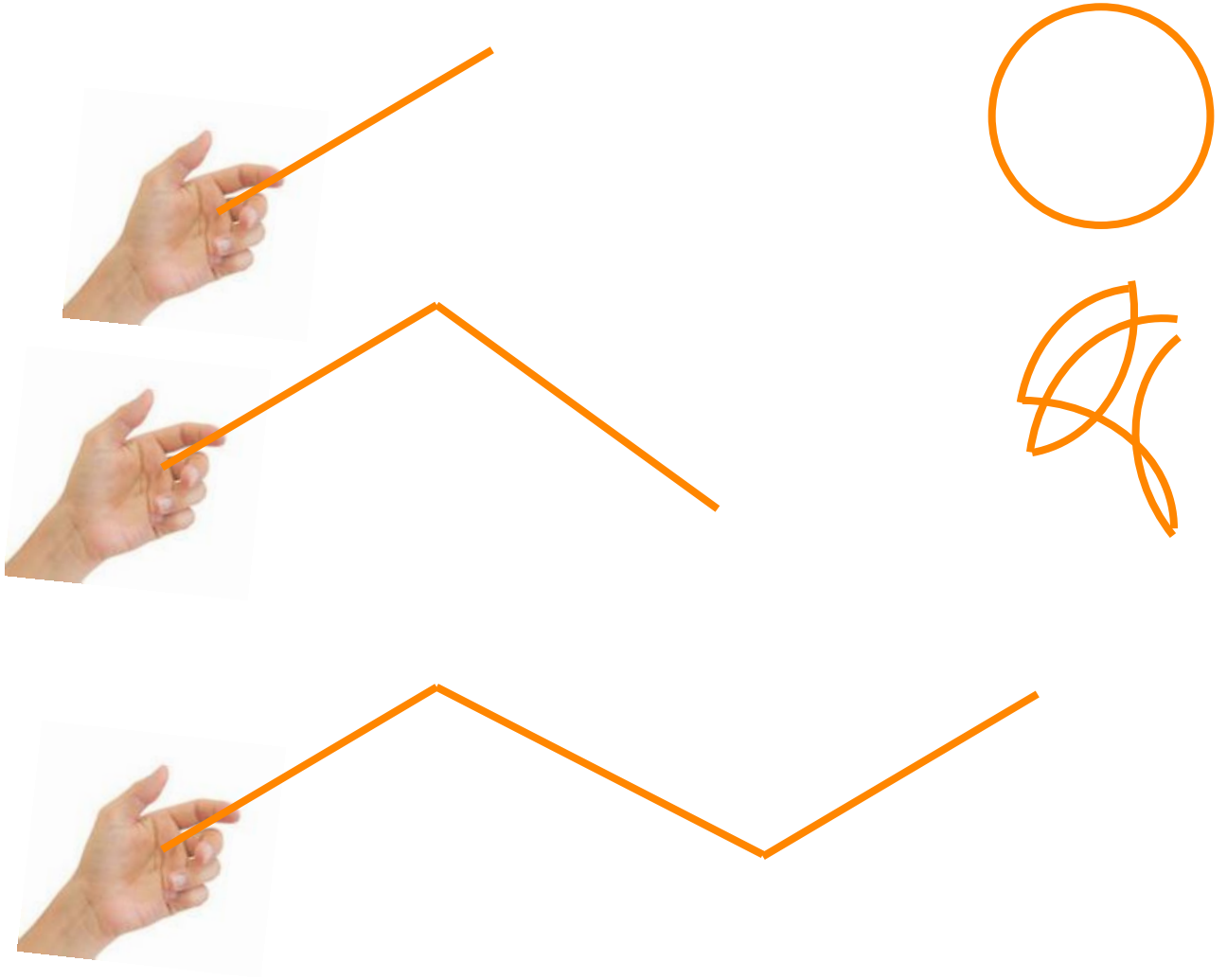
Проблемы

- Сильно связанный код
- Нет понимания зависимостей
- Невозможность написать тесты
- Нерасширяемость
- Хрупкость кода в случае изменений
- «Процедурный» подход к разработке

SOLID

- Единственная ответственность
- Открытость для расширения, закрытость для модификации
- Принцип подстановки Лисков
- Принцип разделения интерфейсов
- Принцип инверсии зависимостей

Предсказуемость



...?

```
 public static class ModuleMetadataHelper
{
  public static bool IsHidden(ModuleMetadata metadata)
  {
    if (!WebLicenseHelper.IsVisibleByLicense(metadata.GetOriginal()) || !metadata.IsVisible
                        || metadata.IsSolutionMetadata())
      return true;

    var foldersMetadata = metadata.GetAllSpecialFolderMetadata().Union(metadata.ExternalFolders)
      .Where(f =>
        f.Kind == SpecialFolderKind.Computable &&
        f.IsShow &&
        f.DisplayType == FolderDisplayType.Module)
      .ToList();

    var folders = SpecialFoldersCache.GetComputableFolders(foldersMetadata)
              .Where(f => f.SpecialFolderType.HasValue);
    var entitiesMetadata = ModuleItemVisibilityChecker.GetVisibleEntitiesMetadata(metadata);
    return !(folders.Any() || entitiesMetadata.Any());
  }
......
```

```
public sealed class ModuleMetadataHelper : IModuleMetadataHelper
{
    private readonly IWebLicenseChecker webLicenseChecker;
    private readonly ISpecialFoldersCacheImplementer specialFoldersCacheImplementer;
    private readonly IModuleItemVisibilityChecker moduleItemVisibilityChecker;
    private readonly IFolderFactoryImplementer folderFactoryImplementer;

    public bool IsHidden(ModuleMetadata metadata)
    {
        if (!this.webLicenseChecker.IsVisibleByLicense(metadata.GetOriginal())
            || !metadata.IsVisible || metadata.IsSolutionMetadata())
            return true;
        var foldersMetadata = metadata.GetAllSpecialFolderMetadata()
        .Union(metadata.ExternalFolders)
        .Where(f => f.Kind == SpecialFolderKind.Computable && f.IsShow &&
            f.DisplayType == FolderDisplayType.Module).ToList();
        var folders = this.specialFoldersCacheImplementer
            .GetOrAdd(null, foldersMetadata, (u, fm) =>
            this.folderFactoryImplementer.GetSpecialFolders(fm))
            .Where(f => f.SpecialFolderType.HasValue);
        var entitiesMetadata = this.moduleItemVisibilityChecker
            .GetVisibleEntitiesMetadata(metadata);
        return !(folders.Any() || entitiesMetadata.Any());
    }
```

.....

Инъекция зависимостей

```
public ModuleMetadataHelper(IWebLicenseChecker webLicenseChecker,
        ISpecialFoldersCacheImplementer specialFoldersCacheImplementer,
        IModuleItemVisibilityChecker moduleItemVisibilityChecker,
        IFolderFactoryImplementer folderFactoryImplementer)
{
        this.webLicenseChecker = webLicenseChecker;
        this.specialFoldersCacheImplementer = specialFoldersCacheImplementer;
        this.moduleItemVisibilityChecker = moduleItemVisibilityChecker;
        this.folderFactoryImplementer = folderFactoryImplementer;
}
```

Регистрация в DI

```
services
        .AddSingleton<IWebLicenseChecker, WebLicenseChecker>()
        .AddSingleton<IModuleMetadataHelper, ModuleMetadataHelper>()
        .AddSingleton<ISpecialFoldersCacheImplementer, SpecialFoldersCacheImplementer>()
        .AddSingleton<IModuleItemVisibilityChecker, ModuleItemVisibilityChecker>()
        .AddSingleton<IFolderFactoryImplementer, FolderFactoryImplementer>()
```

Способы

- **Явное вынесение зависимостей, использование DI**

Бизнес-кейсы

```
public sealed class ModuleMetadataHelper : IModuleMetadataHelper
{
        private readonly IWebLicenseChecker webLicenseChecker;
        private readonly ISpecialFoldersCacheImplementer specialFoldersCacheImplementer;
        private readonly IModuleItemVisibilityChecker moduleItemVisibilityChecker;
        private readonly IFolderFactoryImplementer folderFactoryImplementer;

        public bool IsHidden(ModuleMetadata metadata)
        {
                if (!this.webLicenseChecker.IsVisibleByLicense(metadata.GetOriginal())
                        || !metadata.IsVisible || metadata.IsSolutionMetadata())
                        return true;
                var foldersMetadata = metadata.GetAllSpecialFolderMetadata()
                .Union(metadata.ExternalFolders)
                .Where(f => f.Kind == SpecialFolderKind.Computable && f.IsShow &&
                        f.DisplayType == FolderDisplayType.Module).ToList();
                var folders = this.specialFoldersCacheImplementer
                        .GetOrAdd(null, foldersMetadata, (u, fm) =>
                        this.folderFactoryImplementer.GetSpecialFolders(fm))
                        .Where(f => f.SpecialFolderType.HasValue);
                var entitiesMetadata = this.moduleItemVisibilityChecker
                        .GetVisibleEntitiesMetadata(metadata);
                return !(folders.Any() || entitiesMetadata.Any());
        }
```

Позволяет лицензия

Установлен флаг видимости

Не солюшен

Есть папки, либо сущности

.....

Тесты

- Позволяет лицензия
- Установлен флаг видимости
- Не солюшен
- ~~Есть папки, либо сущности~~

```
[Test]
public void IsHidden_ForInvisibleByLicense_ReturnsTrue()
{
        // Arrange
        this.metadata.IsVisible = true;
        this.VisibleByLicense(true);
        // Act
        var result = this.hepler.IsHidden(this.metadata);
        // Assert
        result.Should().BeTrue();
}

[Test]
public void IsHidden_ForInvisibleMetadata_ReturnsTrue()
{
        // Arrange
        this.metadata.IsVisible = false;
        this.VisibleByLicense(true);
        // Act var result =
        this.hepler.IsHidden(this.metadata);
        // Assert result.Should().BeTrue();
}

[Test]
public void IsHidden_ForSolutionMetadata_ReturnsTrue()
{
        // Arrange
        var metadata = new SolutionMetadata();
        // Act
        var result = this.hepler.IsHidden(metadata);
        // Assert
        result.Should().BeTrue();
}
```

Способы

- Явное вынесение зависимостей, использование DI
- **Повышение тестового покрытия**

А если сущность создается в методе?

```
public class ModuleNode : ModuleNodeBase
{ ...
        public static IEnumerable<ModuleNode> CreateModuleNodeCollection(NodeBase parent)
        {
                var moduleNodes = ModuleManager.Instance.Modules
                        .Where(m => m.HasPresenter)
                        .Select(m => new ModuleNode(m, parent))
                        .ToList(); var computableFolders = GetComputableFolders(moduleNodes);
                var allVisibleEntities = moduleNodes.ToDictionary(m => m, m =>
                                moduleItemVisibilityChecker.GetVisibleEntitiesMetadata(m.Metadata));
                foreach (var moduleNode in moduleNodes)
                {
                         var children = moduleNode.CreateChildContexts(computableFolders,
                                allVisibleEntities[moduleNode]);
                        moduleNode.Children.AddRange(children);
                }

                return moduleNodes;
        }
```

Фабрика

```
public class ModuleNode : ModuleNodeBase
{ ...
        public IEnumerable<ModuleNode> CreateModuleNodeCollection(NodeBase parent)
        {
                var moduleNodes = ModuleManager.Instance.Modules
                        .Where(m => m.HasPresenter)
                        .Select(m => moduleNodeFactory.Create(m, parent))
                        .ToList();
                var computableFolders = GetComputableFolders(moduleNodes);
                var allVisibleEntities = moduleNodes.ToDictionary(m => m, m =>
                        moduleItemVisibilityChecker.GetVisibleEntitiesMetadata(m.Metadata));
                foreach (var moduleNode in moduleNodes)
                {
                        var children = moduleNode.CreateChildContexts(computableFolders,
                                allVisibleEntities[moduleNode]);
                        moduleNode.Children.AddRange(children);
                }

                return moduleNodes;
}
```

Фабрика реализация

```
internal class ModuleNodeCollectionFactory : IModuleNodeCollectionFactory
{
        public IEnumerable<ModuleNode> Create(NodeBase parent)
        {
                var moduleNodes = ModuleManager.Instance.Modules
                        .Where(m => m.HasPresenter)
                        .Select(m => this.moduleNodeFactory.Create(m, parent))
                        .ToList();
```

```
public ModuleNodeCollectionFactory(
        IFactory<ICompositeModule, NodeBase, ModuleNode> moduleNodeFactory,
```

Фабрика универсальная реализация

```
public class Factory<T> : IFactory<T>
{
        public T Create()
        {
                return ServiceProvider.Instance.GetService<T>();
        }
}

public class Factory<TParam, TResult> : IFactory<TParam, TResult>
{
        public TResult Create(TParam param)
        {
                return ActivatorUtilities.CreateInstance<TResult>(
                        ServiceProvider.Instance, new object[] { param });
        }
}
```

Фабрика – универсальный интерфейс

```
/// <summary>
/// Обобщенная фабрика.
/// </summary>
/// <typeparam name="T">Тип возвращаемого объекта.</typeparam>
public interface IFactory<out T>
{
        /// <summary>
        /// Создать объект.
        /// </summary>
        /// <returns>Объект указанного типа.</returns>
        T Create();
}
/// <summary>
/// Обобщенная фабрика с параметром.
/// </summary>
/// <typeparam name="TParam">Тип принимаемого параметра.</typeparam>
/// <typeparam name="TResult">Тип возвращаемого объекта.</typeparam>
public interface IFactory<in TParam, out TResult>
{
        /// <summary>
        /// Создать объект.
        /// </summary>
        /// <param name="param">Аргумент.</param>
        /// <returns>Объект указанного типа.</returns>
        TResult Create(TParam param);
}
```

Примеры с разными DI контейнерами

Microsoft.Extensions.DependencyInjection

```
return ActivatorUtilities.CreateInstance<TResult>(
ServiceProvider.Instance, new object[] { param });
```

Только класс (не абстрактный)

Untity

```
var dependency = new DependencyOverride(typeof(TParam), param);
return unityContainer.Resolve<TResult>(dependency);
```

Autofac

```
var reader = Container.Resolve<TParam>(new TypedParameter(typeof(TParam), param));
```

Способы

- Явное вынесение зависимостей, использование DI
- Повышение тестового покрытия
- **Использование фабрик вместо явного вызова конструктора**

Фабрика вместо статического метода Create

```
public class ModuleNode : ModuleNodeBase
{ ...
/// <summary>
/// Создать коллекцию контекстов модулей с инициализацией всех её элементов.
/// </summary>
/// <param name="parent">Родительский контекст.</param>
/// <returns>Контексты модулей.</returns>
public static IEnumerable<ModuleNode> CreateModuleNodeCollection(NodeBase parent)
{
```



```
internal interface IModuleNodeCollectionFactory
{
        IEnumerable<ModuleNode> Create(NodeBase parent);
}
```

- 100 строк кода из файла ModuleNode
- Несколько зависимостей класса ModuleNode
- Оказалось, что ModuleNode и ModuleNodeCollectionFactory абсолютно независимы и вызывались для разных целей

Способы

- Явное вынесение зависимостей, использование DI
- Повышение тестового покрытия
- Использование фабрик вместо явного вызова конструктора
- **Следование принципу единственной ответственности, разбиение на несколько классов**

Рефакторинг влечет изменения вызывающего кода

- Нужно сделать классы, которые были статические, не-статическими
- Переписать вызовы их методов на не-статические
- Заинжектить интерфейсы классов в те классы, где они используются как зависимости
- Возможно, разделить статический класс на несколько в соответствии с принципом единственности ответственности

Лавина изменений

Нужно уметь вовремя остановиться

- Фокус на цели рефакторинга
- Планирование и выделение шагов для дальнейших изменений
- Придерживаться одного стиля во всех дальнейших рефакторингах

Причины «остановки»

- Код в таком виде (например, статический вызов) используется в большом количестве мест (>4-5)
- Объекты не зарегистрированы в DI (вызывающие код), а создаются через конструктор, вызовов много (>4-5)
- Объекты сложные (вызывающие код), содержат большое количество зависимостей

Способы «остановки»

- Использовать «стандартные» для вашего кода паттерны
- Сделать экземплярную обертку над статическим классом или методом, чтобы можно было вызывать и так, и так, статическую потом удалить
- Если в коде используется ServiceLocator, то за один рефакторинг от него не избавиться (ServiceProvider.Resolve(…))

Способы

- Явное вынесение зависимостей, использование DI
- Повышение тестового покрытия
- Использование фабрик вместо явного вызова конструктора
- Следование принципу единственной ответственности, разбиение на несколько классов
- **Умение вовремя остановиться и ограничить скоуп изменений**

Управление видимостью

Как быть с internal-методами?

Делать их public, а сам класс делать internal

```
internal class ChildNodeHelper
{
        internal bool TryCreateChildContext
                (IEntity entity, out NodeBase context)
        { …
```

```
internal class ChildNodeHelper : IChildNodeHelper
{
        public bool TryCreateChildContext
                (IEntity entity, out NodeBase context)
        { …
```



```
internal /*public*/ interface IChildNodeHelper
{
        bool TryCreateChildContext(IEntity entity, out NodeBase context);
}
```

Управление видимостью

Как быть с internal-методами?

Делать их public, а сам класс делать internal

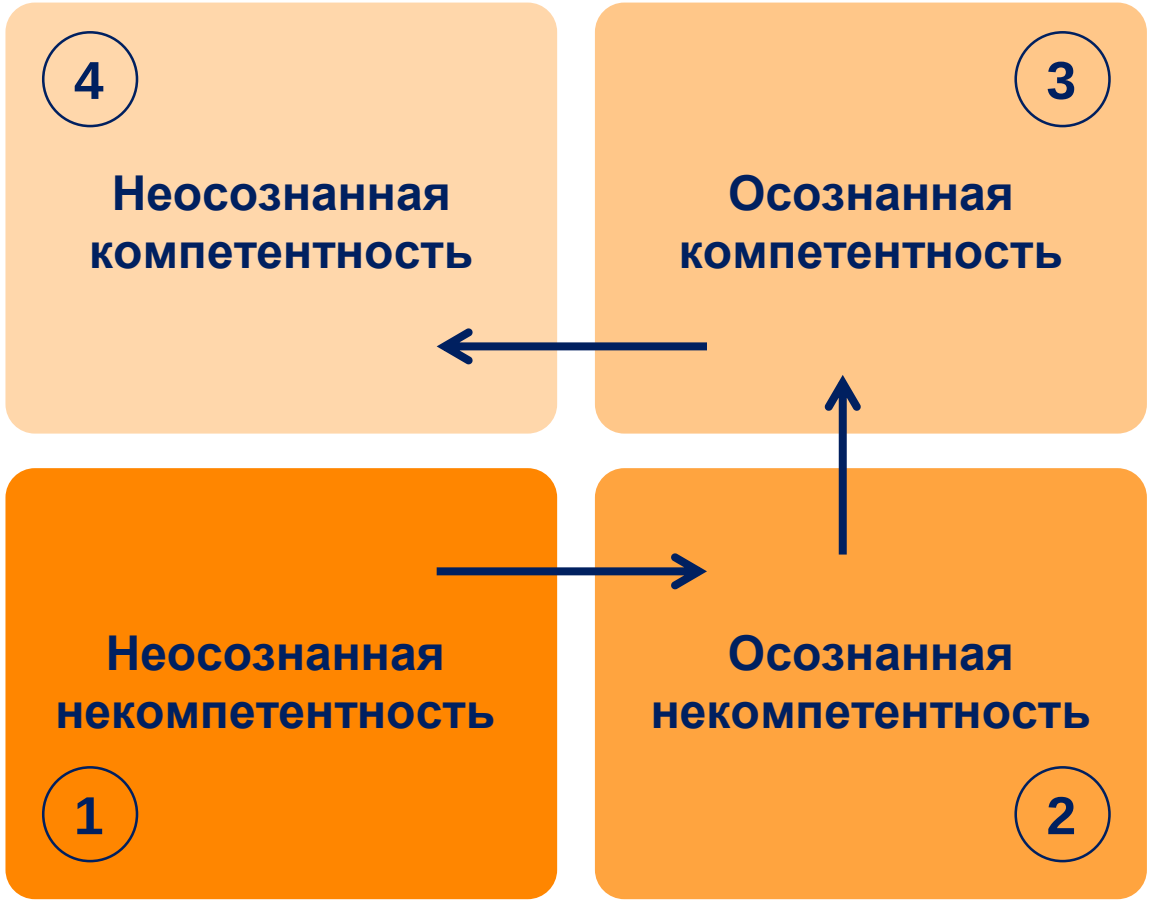
```
internal class ChildNodeHelper
{
        internal ChildNodeHelper(NodeBase parentContext,
                …)
        {
```

```
internal class ChildNodeHelper
{
        public ChildNodeHelper(NodeBase parentContext,
                …)
        {
```

Способы

- Явное вынесение зависимостей, использование DI
- Повышение тестового покрытия
- Использование фабрик вместо явного вызова конструктора
- Следование принципу единственной ответственности, разбиение на несколько классов
- Умение вовремя остановиться и ограничить скоуп изменений
- **Управление видимостью**

Стадии развития специалиста



Эмоциональное выгорание

- Излишки стресса
- Переработки
- Непонимание, как это все работает и что мне с этим делать
- Банальная лень 

Способы

- Явное вынесение зависимостей, использование DI
- Повышение тестового покрытия
- Использование фабрик вместо явного вызова конструктора
- Следование принципу единственной ответственности, разбиение на несколько классов
- Умение вовремя остановиться и ограничить скоуп изменений
- Управление видимостью
- …

Эмоциональная стабильность

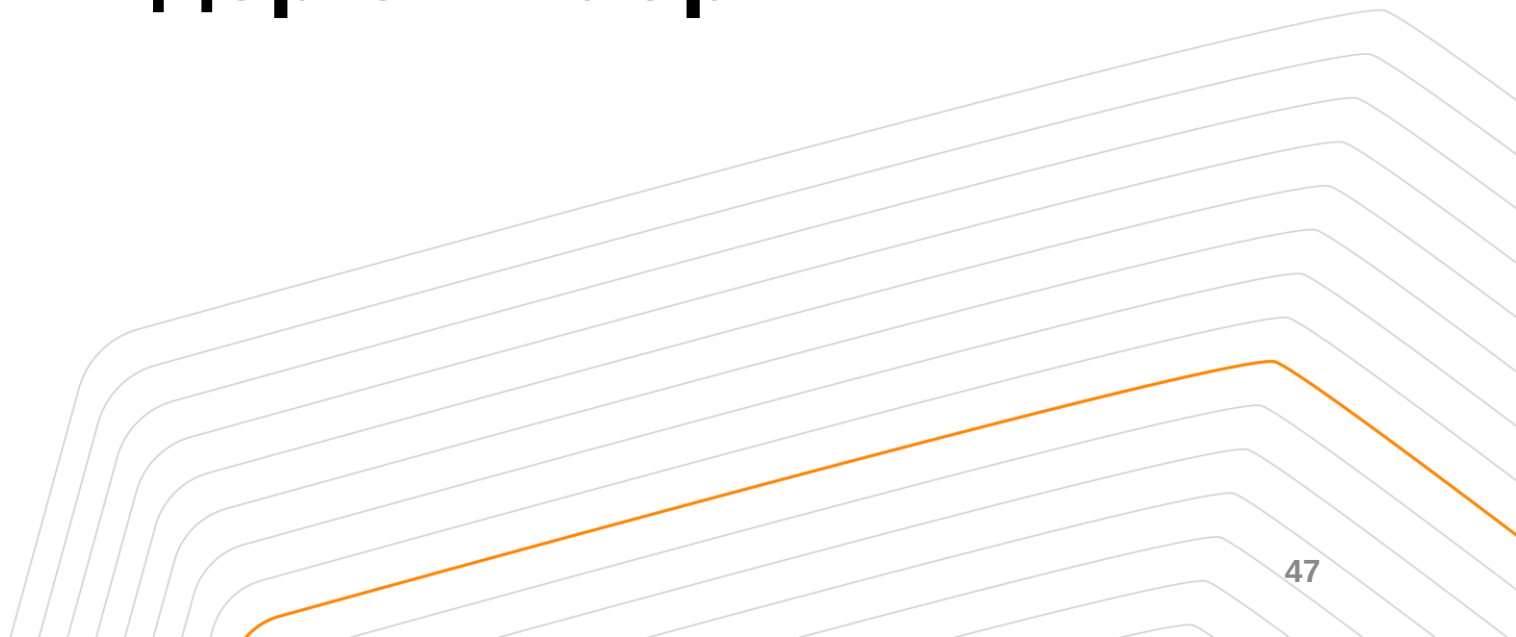
- Повышение понимания, как это работает
- Специалист из одной сферы переходит в другую более «спокойно» достигает тех же вершин (уровня зар.платы)
- Терпение и труд (ну, и конечно немного удачи)
- Возможно, это история не про вас, и вы любите «американские горки». Говорят, это болезнь молодости. Сколько людей так сорвалось, а сколько добилось успеха, и все равно пришли к тому же самому.


О, зарплата пришла!
@RABOTARZD
как всё достало! пора увольняться.

Литература

- Working Effectively with Legacy Code - Michael C. FeathersMichael C. Feathers https://www.amazon.com/Working-Effectively-Legacy-Michael-Feathers/dp/0131177052
- Чистый Код – Роберт Мартин
- Чистая Архитектура – Роберт Мартин
- Паттерны объектно-ориентированного проектирования – Гамма, Хелм
- Паттерны проектирования на языке C# - Тепляков

В чем основное отличие паттерна Билдер от Фабрики?



Вопросы

Елена Щелкунова