



HELP UKRAINE STOP RUSSIA

[🏠](#) / [Паттерны проектирования](#) / [Порождающие паттерны](#)

Фабричный метод

Также известен как: Виртуальный конструктор, Factory Method

Суть паттерна

Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.



Проблема

Представьте, что вы создаёте программу управления грузовыми перевозками. Сперва вы рассчитываете перевозить товары только на автомобилях. Поэтому весь ваш код работает с объектами класса `Грузовик`.

В какой-то момент ваша программа становится настолько известной, что морские перевозчики выстраиваются в очередь и просят добавить поддержку морской логистики в программу.



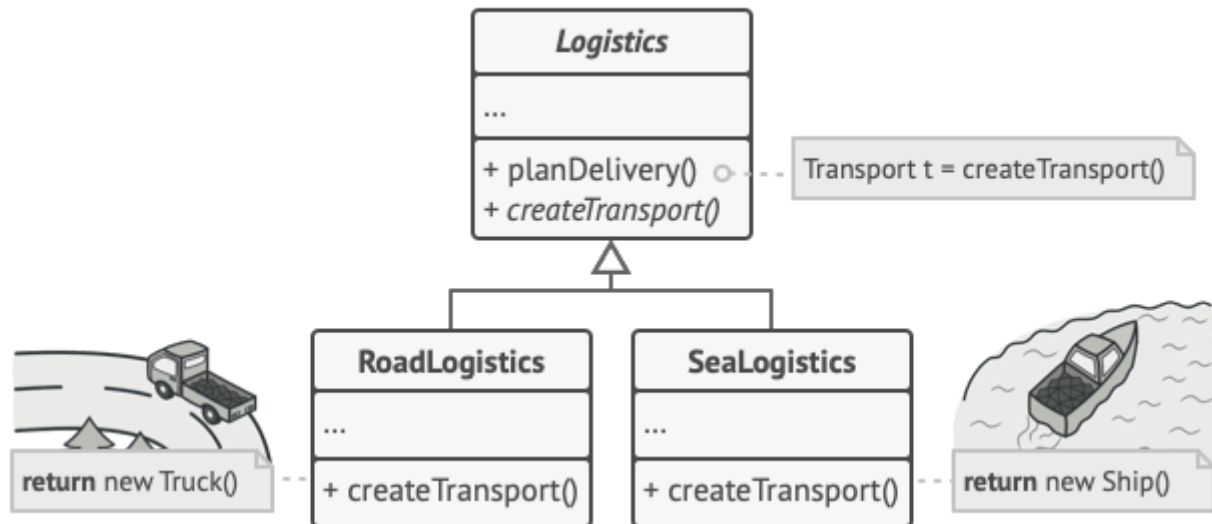
Добавить новый класс не так-то просто, если весь код уже завязан на конкретные классы.

Отличные новости, правда?! Но как насчёт кода? Большая часть существующего кода жёстко привязана к классам `Грузовиков`. Чтобы добавить в программу классы морских `Судов`, понадобится перелопатить всю программу. Более того, если вы потом решите добавить в программу ещё один вид транспорта, то всю эту работу придётся повторить.

В итоге вы получите ужасающий код, наполненный условными операторами, которые выполняют то или иное действие, в зависимости от класса транспорта.

😊 Решение

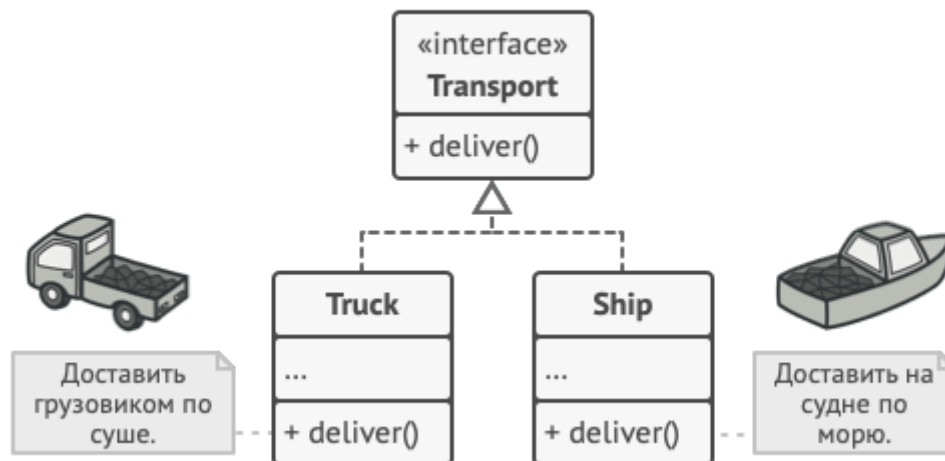
Паттерн Фабричный метод предлагает создавать объекты не напрямую, используя оператор `new`, а через вызов особого *фабричного* метода. Не пугайтесь, объекты всё равно будут создаваться при помощи `new`, но делать это будет фабричный метод.



Подклассы могут изменять класс создаваемых объектов.

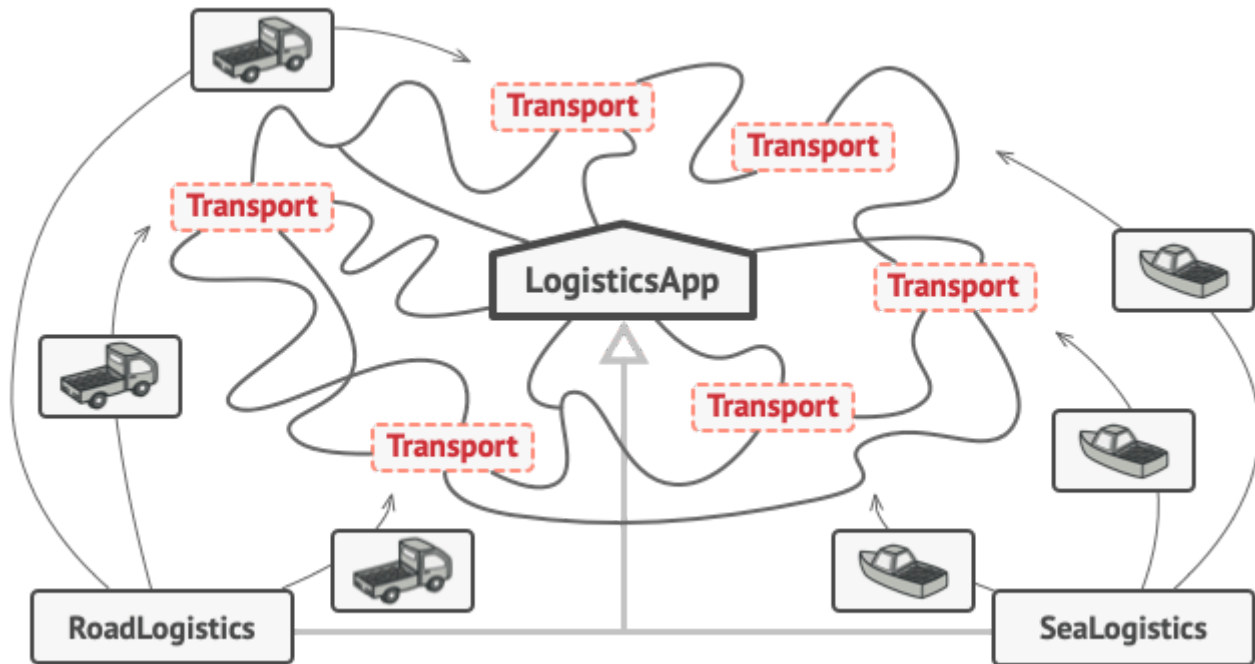
На первый взгляд, это может показаться бессмысленным: мы просто переместили вызов конструктора из одного конца программы в другой. Но теперь вы сможете переопределить фабричный метод в подклассе, чтобы изменить тип создаваемого продукта.

Чтобы эта система заработала, все возвращаемые объекты должны иметь общий интерфейс. Подклассы смогут производить объекты различных классов, следующих одному и тому же интерфейсу.



Все объекты-продукты должны иметь общий интерфейс.

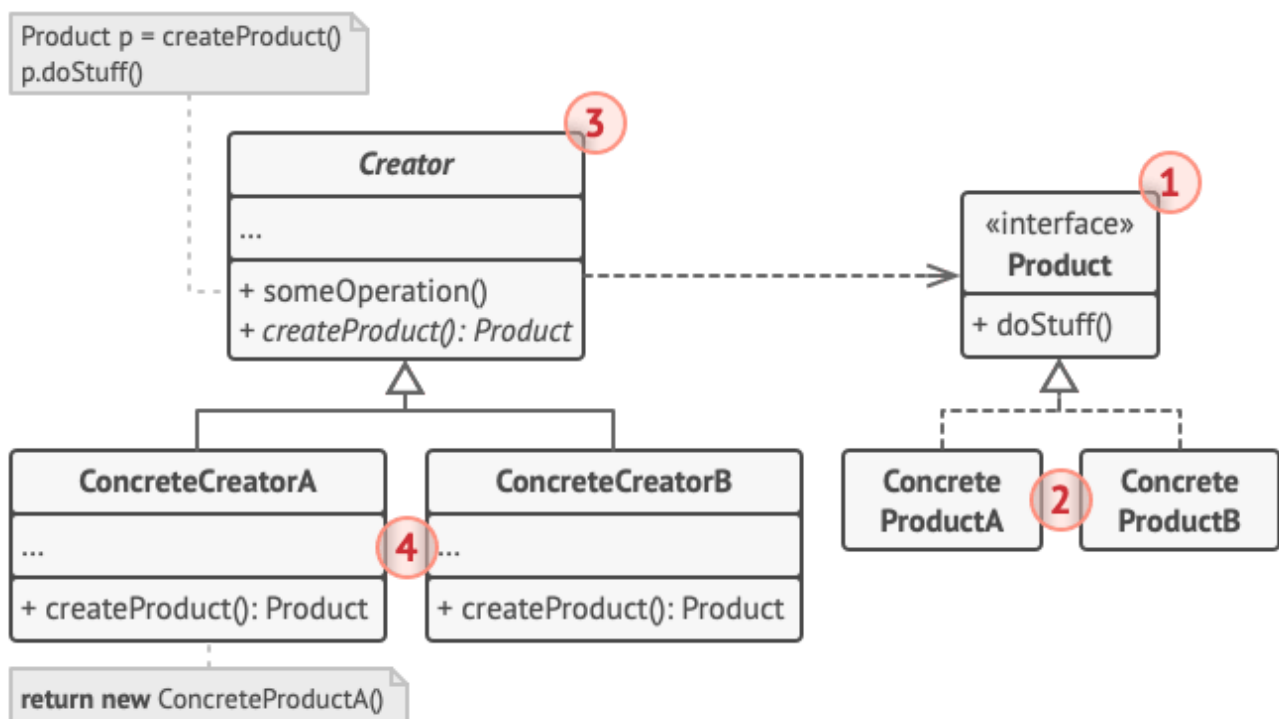
Например, классы `Грузовик` и `Судно` реализуют интерфейс `Транспорт` с методом `доставить`. Каждый из этих классов реализует метод по-своему: грузовики везут грузы по земле, а суда — по морю. Фабричный метод в классе `ДорожнойЛогистики` вернёт объект-грузовик, а класс `МорскойЛогистики` — объект-судно.



Пока все продукты реализуют общий интерфейс, их объекты можно взаимозаменять в клиентском коде.

Для клиента фабричного метода нет разницы между этими объектами, так как он будет трактовать их как некий абстрактный `Транспорт`. Для него будет важно, чтобы объект имел метод `доставить`, а как конкретно он работает — не важно.

Структура



1. **Продукт** определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.
2. **Конкретные продукты** содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.
3. **Создатель** объявляет фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов.

Зачастую фабричный метод объявляют абстрактным, чтобы заставить все подклассы реализовать его по-своему. Но он может возвращать и некий стандартный продукт.

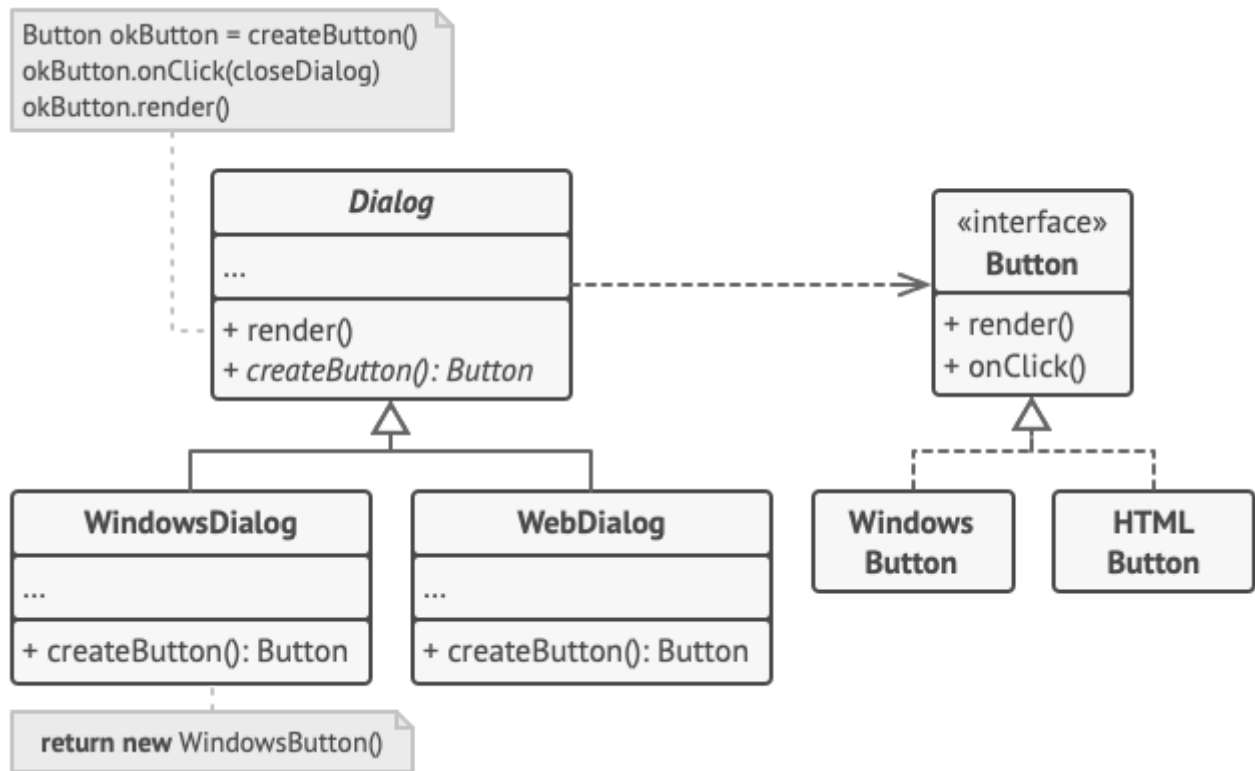
Несмотря на название, важно понимать, что создание продуктов **не является** единственной функцией создателя. Обычно он содержит и другой полезный код работы с продуктом. Аналогия: большая софтверная компания может иметь центр подготовки программистов, но основная задача компании — создавать программные продукты, а не готовить программистов.

4. **Конкретные создатели** по-своему реализуют фабричный метод, производя те или иные конкретные продукты.

Фабричный метод не обязан всё время создавать новые объекты. Его можно переписать так, чтобы возвращать существующие объекты из какого-то хранилища или кэша.

Псевдокод

В этом примере **Фабричный метод** помогает создавать кросс-платформенные элементы интерфейса, не привязывая основной код программы к конкретным классам элементов.



Пример кросс-платформенного диалога.

Фабричный метод объявлен в классе диалогов. Его подклассы относятся к различным операционным системам. Благодаря фабричному методу, вам не нужно переписывать логику диалогов под каждую систему. Подклассы могут наследовать почти весь код из базового диалога, изменяя типы кнопок и других элементов, из которых базовый код строит окна графического пользовательского интерфейса.

Базовый класс диалогов работает с кнопками через их общий программный интерфейс. Поэтому, какую вариацию кнопок ни вернул бы фабричный метод, диалог останется рабочим. Базовый класс не зависит от конкретных классов кнопок, оставляя подклассам решение о том, какой тип кнопок создавать.

Такой подход можно применить и для создания других элементов интерфейса. Хотя каждый новый тип элементов будет приближать вас к **Абстрактной фабрике**.

```
// Паттерн Фабричный метод применим тогда, когда в программе
// есть иерархия классов продуктов.
```

```
interface Button is
```

```
    method render()
```

```
    method onClick(f)
```

```
class WindowsButton implements Button is
```

```
    method render(a, b) is
```

```
        // Отрисовать кнопку в стиле Windows.
```

```
    method onClick(f) is
```

```
// Навесить на кнопку обработчик событий Windows.

class HTMLButton implements Button is
    method render(a, b) is
        // Вернуть HTML-код кнопки.
    method onClick(f) is
        // Навесить на кнопку обработчик события браузера.

// Базовый класс фабрики. Заметьте, что "фабрика" — это всего
// лишь дополнительная роль для класса. Скорее всего, он уже
// имеет какую-то бизнес-логику, в которой требуется создание
// разнообразных продуктов.
class Dialog is
    method render() is
        // Чтобы использовать фабричный метод, вы должны
        // убедиться в том, что эта бизнес-логика не зависит от
        // конкретных классов продуктов. Button — это общий
        // интерфейс кнопок, поэтому все хорошо.
        Button okButton = createButton()
        okButton.onClick(closeDialog)
        okButton.render()

// Мы выносим весь код создания продуктов в особый метод,
// который называют "фабричным".
    abstract method createButton():Button

// Конкретные фабрики переопределяют фабричный метод и
// возвращают из него собственные продукты.
class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()

class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()

class Application is
    field dialog: Dialog

// Приложение создаёт определённую фабрику в зависимости от
// конфигурации или окружения.
    method initialize() is
        config = readApplicationConfigFile()


        if (config.OS == "Windows") then
            dialog = new WindowsDialog()
        else if (config.OS == "Web") then
            dialog = new WebDialog()
        else
```

```
throw new Exception("Error! Unknown operating system.")
```


```
// Если весь остальной клиентский код работает с фабриками и  
// продуктами только через общий интерфейс, то для него  
// будет не важно, какая фабрика была создана изначально.  
method main() is  
    this.initialize()  
    dialog.render()
```


Применимость

 Когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код.

 Фабричный метод отделяет код производства продуктов от остального кода, который эти продукты использует.

Благодаря этому, код производства можно расширять, не трогая основной. Так, чтобы добавить поддержку нового продукта, вам нужно создать новый подкласс и определить в нём фабричный метод, возвращая оттуда экземпляр нового продукта.

 Когда вы хотите дать возможность пользователям расширять части вашего фреймворка или библиотеки.


 Пользователи могут расширять классы вашего фреймворка через наследование. Но как сделать так, чтобы фреймворк создавал объекты из этих новых классов, а не из стандартных?


Решением будет дать пользователям возможность расширять не только желаемые компоненты, но и классы, которые создают эти компоненты. А для этого создающие классы должны иметь конкретные создающие методы, которые можно определить.

Например, вы используете готовый UI-фреймворк для своего приложения. Но вот беда — требуется иметь круглые кнопки, вместо стандартных прямоугольных. Вы создаёте класс `RoundButton`. Но как сказать главному классу фреймворка `UIFramework`, чтобы он теперь создавал круглые кнопки, вместо стандартных?

Для этого вы создаёте подкласс `UIWithRoundButtons` из базового класса фреймворка, переопределяете в нём метод создания кнопки (а-ля `createButton`) и вписываете туда

создание своего класса кнопок. Затем используете `UIWithRoundButtons` вместо стандартного `UIFramework`.

 Когда вы хотите экономить системные ресурсы, повторно используя уже созданные объекты, вместо порождения новых.

 Такая проблема обычно возникает при работе с тяжёлыми ресурсоёмкими объектами, такими, как подключение к базе данных, файловой системе и т. д.

Представьте, сколько действий вам нужно совершить, чтобы повторно использовать существующие объекты:

1. Сначала вам следует создать общее хранилище, чтобы хранить в нём все создаваемые объекты.
2. При запросе нового объекта нужно будет заглянуть в хранилище и проверить, есть ли там неиспользуемый объект.
3. А затем вернуть его клиентскому коду.
4. Но если свободных объектов нет — создать новый, не забыв добавить его в хранилище.

Весь этот код нужно куда-то поместить, чтобы не засорять клиентский код.

Самым удобным местом был бы конструктор объекта, ведь все эти проверки нужны только при создании объектов. Но, увы, конструктор всегда создаёт **новые** объекты, он не может вернуть существующий экземпляр.

Значит, нужен другой метод, который бы отдавал как существующие, так и новые объекты. Им и станет фабричный метод.

Шаги реализации

1. Приведите все создаваемые продукты к общему интерфейсу.
2. В классе, который производит продукты, создайте пустой фабричный метод. В качестве возвращаемого типа укажите общий интерфейс продукта.
3. Затем пройдитесь по коду класса и найдите все участки, создающие продукты. Поочерёдно замените эти участки вызовами фабричного метода, перенося в него код создания различных продуктов.

В фабричный метод, возможно, придётся добавить несколько параметров, контролирующих, какой из продуктов нужно создать.

На этом этапе фабричный метод, скорее всего, будет выглядеть удручающе. В нём будет жить большой условный оператор, выбирающий класс создаваемого продукта. Но не волнуйтесь, мы вот-вот исправим это.

4. Для каждого типа продуктов заведите подкласс и переопределите в нём фабричный метод. Переместите туда код создания соответствующего продукта из суперкласса.
5. Если создаваемых продуктов слишком много для существующих подклассов создателя, вы можете подумать о введении параметров в фабричный метод, которые позволят возвращать различные продукты в пределах одного подкласса.

Например, у вас есть класс `Почта` с подклассами `АвиаПочта` и `НаземнаяПочта`, а также классы продуктов `Самолёт`, `Грузовик` и `Поезд`. `Авиа` соответствует `Самолётам`, но для `НаземнойПочты` есть сразу два продукта. Вы могли бы создать новый подкласс почты для поездов, но проблему можно решить и по-другому. Клиентский код может передавать в фабричный метод `НаземнойПочты` аргумент, контролирующий тип создаваемого продукта.

6. Если после всех перемещений фабричный метод стал пустым, можете сделать его абстрактным. Если в нём что-то осталось — не беда, это будет его реализацией по умолчанию.

⚖️ Преимущества и недостатки

- ✓ Избавляет класс от привязки к конкретным классам продуктов.
- ✓ Выделяет код производства продуктов в одно место, упрощая поддержку кода.
- ✓ Упрощает добавление новых продуктов в программу.
- ✓ Реализует *принцип открытости/закрытости*.
- ✗ Может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.

↔️ Отношения с другими паттернами

- Многие архитектуры начинаются с применения Фабричного метода (более простого и расширяемого через подклассы) и эволюционируют в сторону Абстрактной фабрики, Прототипа или Строителя (более гибких, но и более сложных).

- Классы Абстрактной фабрики чаще всего реализуются с помощью Фабричного метода, хотя они могут быть построены и на основе Прототипа.
- Фабричный метод можно использовать вместе с Итератором, чтобы подклассы коллекций могли создавать подходящие им итераторы.
- Прототип не опирается на наследование, но ему нужна сложная операция инициализации. Фабричный метод, наоборот, построен на наследовании, но не требует сложной инициализации.
- Фабричный метод можно рассматривать как частный случай Шаблонного метода. Кроме того, *Фабричный метод* нередко бывает частью большого класса с *Шаблонными методами*.

</> Примеры реализации паттерна



📖 Дополнительные материалы

- Если вы уже слышали о *Фабрике*, *Фабричном методе* или *Абстрактной фабрике*, но с трудом их различаете — почитайте нашу статью Сравнение фабрик.



Книга всегда под рукой

В отличие от обычной бумажной книги о программировании...

...в нашей есть поиск, и её невозможно физически где-то забыть. Она всегда готова к чтению, в телефоне или на рабочем компе.

 Узнать больше...

Главная



Рефакторинг



Паттерны



Премиум контент

Форум

Связаться

© 2014-2023 Refactoring.Guru. Все права защищены.

 Иллюстрации нарисовал Дмитрий Жарт

 Хмельницкое шоссе 19 / 27, Каменец-Подольский, Украина, 32305

 Email: support@refactoring.guru

Условия использования

Политика конфиденциальности

Использование контента