УДК 004.045 ***DOI: 10.46960/1816-210X_2022_1_17***

ОСОБЕННОСТИ ПРИМЕНЕНИЯ ПРИНЦИПА РАЗДЕЛЕНИЯ ИНТЕРФЕЙСОВ

С.В. Логанов
ORCID: **0000-0002-7302-4586** e-mail: **loganovserg@yandex.ru**
Нижегородский государственный технический университет им. Р.Е. Алексеева
Нижний Новгород, Россия

Проанализирован принцип разделения интерфейсов и его применение для классов, расположенных в одном или нескольких компонентах. Показано, что первоначальная формулировка принципа разделения интерфейсов справедлива только для классов, расположенных в различных компонентах. В качестве принципа проектирования классов принцип разделения интерфейсов необходимо рассматривать как поддержку принципа LSP. Разделение интерфейсов позволяет исключить из них методы, реализация которых не всегда возможна, тем самым предотвратив нарушение принципа LSP.

Ключевые слова: объектно-ориентированное программирование, принципы SOLID, принцип ISP, принцип LSP, иерархия классов, повторное использование, обобщение/специализация.

ДЛЯ ЦИТИРОВАНИЯ: Логанов, С.В. Особенности применения принципа разделения интерфейсов // Труды НГТУ им. Р.Е. Алексеева. 2022. № 1. С. 17-23. DOI: 10.46960/1816-210X_2022_1_17

APPLICATION FEATURES OF THE INTERFACE SEGREGATION PRINCIPLE

S.V. Loganov
ORCID: **0000-0002-7302-4586** e-mail: **loganovserg@yandex.ru**
Nizhny Novgorod state technical university n.a. R.E. Alekseev
Nizhny Novgorod, Russia

Abstract. Interfaces separation principle and its use for classes located in one or several components, are analyzed. It is shown that the initial formulation of the interfaces separation principle is true only for classes located in different components. As a class design principle, the interfaces separation principle should be considered as support for the LSP principle. Separation of interfaces makes it possible to exclude methods, the implementation of which is not always possible, and thereby to prevent the breach of LSP principle.

Key words: object-oriented programming, SOLID principles, ISP principle, LSP principle, class hierarchy, reuse, generalization/specialization**.**

FOR CITATION: Loganov S.V. Application features of the interface segregation principle. Transactions of NNSTU n.a. R.E. Alekseev. 2022. №1. Pp. 17-23. DOI: 10.46960/1816-210X_2022_1_17

Введение

В настоящее время принципы SOLID являются основой для разработки структуры классов любой информационной системы. Одним них является принцип разделения интерфейсов (Interface Segregation Principle ISP), сформулированный Р. Мартином следующим образом: клиенты не должны вынужденно зависеть от методов, которыми не пользуются [1]. Принципы SOLID позиционируются как принципы гибкого проектирования классов, поскольку принципы проектирования компонентов выделены в отдельную группу [2-4]. Принцип разделения интерфейсов был обозначен Р. Мартином при консультировании фирмы Xerox по разработке широкой функциональности программных компонентов принтеров [5]. Таким образом, будучи изначально применен для классов, расположенных в различных ком-

понентах, он был сформулирован как принцип гибкого проектирования для любых классов. Отсутствие указаний о местоположении классов приводит к неоднозначному пониманию принципа разделения интерфейсов.

Особенности применения

Для демонстрации принципа разделения интерфейсов для классов безотносительно их местоположения в [1, 2] приведен достаточно противоречивый пример с дверями системы безопасности, в котором используется наследование интерфейса Door от интерфейса Timer-Client, что явно противоречит основному принципу наследования. Кроме этого, совершенно надуманный пример демонстрации принципа ISP с точки зрения классов, располагаемых в рамках одного компонента, приведен в [3]. Согласно ему, три класса User1, User2 и User3 пользуются операциями некоторого класса OPS. При этом класс User1 использует только операцию op1, User2 — только op2 и User3 — только op3 (рис. 1) [3].

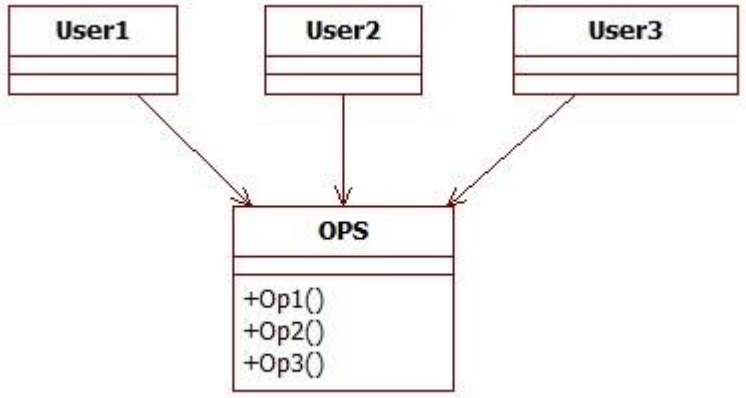

Рис. 1. Структура классов без использования интерфейсов

Fig. 1. Class structure without using interfaces

«Очевидно, что в такой ситуации исходный код User1 непреднамеренно будет зависеть от op2 и op3 даже при том, что он не пользуется ими. Эта зависимость означает, что изменения в исходном коде метода op2 в классе OPS потребуют повторной компиляции и развертывания класса User1, несмотря на то, что для него ничего не изменилось» [3]. Совершенно очевидно и то, что если все указанные классы расположены в одном компоненте, то они компилируются и развертываются совместно и требование повторной компиляции здесь является не применимым. Также очевидно, что все клиенты данного класса непреднамеренно зависят не только от неиспользуемых публичных методов, но и от частных методов их реализации, которые используют публичные методы. Ввод дополнительных раздельных интерфейсов (рис. 2), как рекомендовано в [3], никоим образом не изменяет данные зависимости.

Классы User1, User2, User3 в данной ситуации по-прежнему зависят от состояния объекта класса OPS, только данная зависимость становится еще более завуалированной. Использование различных объектов класса OPS позволяет существенно уменьшить зависимость от текущего состояния конкретного объекта OPS, однако данное действие не возбраняется и для исходной структуры классов. Кроме этого, если класс OPS не требует сложных методов инициализации, то классы в первоначальном варианте для снижения влияния текущего состояния сами могут решить вопрос создания нового экземпляра класса, что принципиально невозможно для варианта с интерфейсами. Таким образом, введение дополнительного слоя интерфейсов лишь усложняет и затуманивает восприятие программного кода при условии расположения всех классов в одном компоненте.

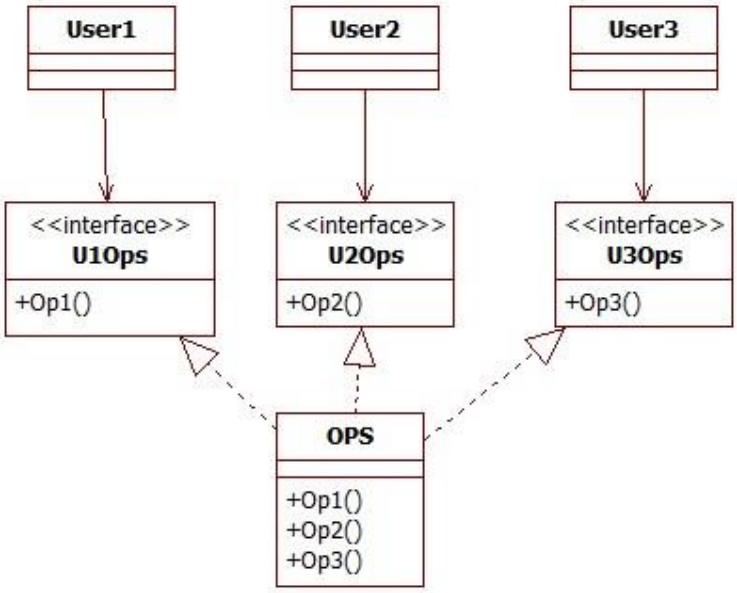

Рис. 2. Структура классов с использованием разделения интерфейсов

Fig. 2. Class structure using interface segregation

Наличие у класса OPS в том же самом компоненте трех клиентов, которые независимо используют лишь им необходимые методы, как правило, свидетельствует о нарушении однородности (низком зацеплении) методов класса и нарушении им принципа единственности ответственности. Таким образом, класс OPS необходимо просто разделить на три класса с соответствующими интерфейсами, а при необходимости для них общей информации предоставить им один экземпляр совместно используемого класса (рис. 3).

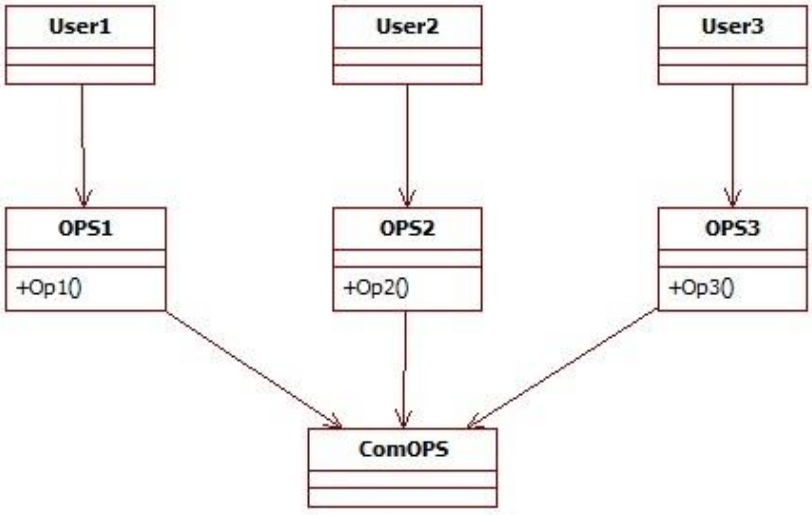

Рис. 3. Структура классов, соответствующих принципу единственности ответственности

Fig. 3. The structure of classes corresponding to the single responsibility principle

Если же разнести указанные классы по различным компонентам (рис. 4), ситуация кардинально меняется, и основным требованием становится обеспечение независимости напрямую не связанных между собою компонентов. Изменение же требований, например, к

классу User3 означает, что требуются изменения в исходном коде метода op3 класса OPS, которые потребуют повторной компиляции и развертывания компонентов CompU1 и CompU2, несмотря на то, что для классов User1 и User2 ничего не изменилось.

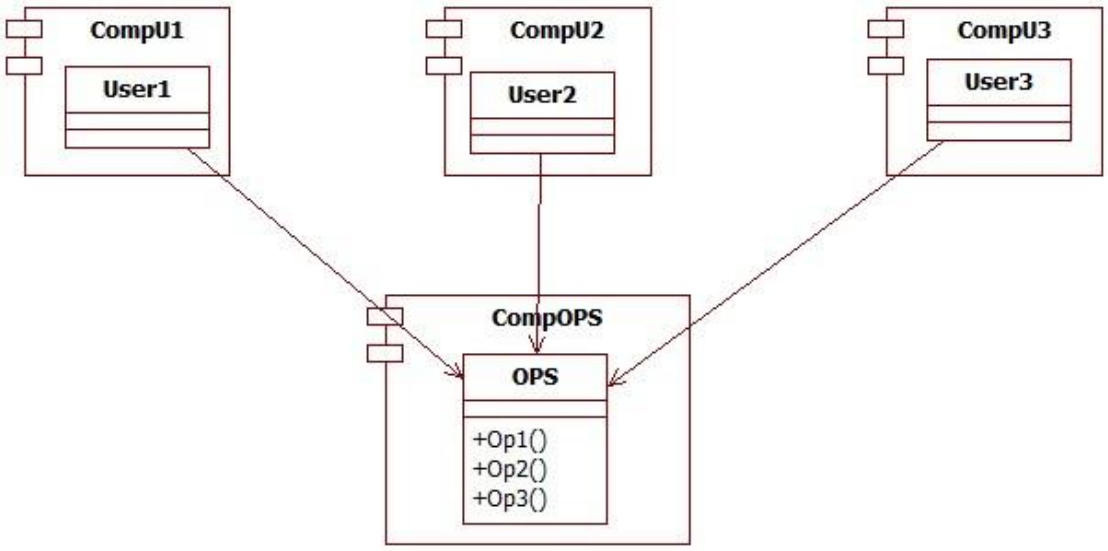

Рис. 4. Отношения классов, расположенных по различным компонентам

Fig. 4. Relationships of classes located on various components

Эта проблема действительно решается разделением операций по различным интерфейсам (рис. 5). Однако разделение интерфейсов лишь на основе заранее неизвестного количества внешних компонентов, которые будут использовать разрабатываемый компонент, невозможно и просто бессмысленно. Поэтому для компонентов необходимо разрабатывать интерфейсы, обеспечивающие выполнение каждой значимой извне узкоспециализированной задачи. Примером такого разделения могут служить узкоспециализированные интерфейсы IDbConnection, IDbCommand, IDataReader библиотеки ADO.NET, обеспечивающие независимость ее клиентов от конкретного провайдера данных. Таким образом, если класс обладает сфокусированными и однородными обязанностями и имеет единственную ответственность, то его клиентам, расположенным в том же компоненте, как правило, необходимы большинство его методов или же такая необходимость с высокой долей вероятностей появится при совершенствовании и развитии данного компонента. И введение промежуточного слоя в виде интерфейсов лишь усложняет разработку, затуманивает понимание структуры и затрудняет поддержку и развитие компонента. Проектирование класса как некого набора независимых групп операций, а не как цельной однородной сущности, обладающей единственной ответственностью, требует проведения рефакторинга, а не ввода дополнительных сущностей, пытающихся скрыть этот факт. Далее в [1, 2] приводится пример с применением пользовательского интерфейса банкомата, реализованного с помощью экрана, брайлевского планшета или синтезатора речи, которые очевидно располагаются в различных компонентах. Кроме того, достоинства приведенного решения в полной мере проявляются при размещении транзакций банкомата по внесению, снятию и переводу наличных денег в различных компонентах. Таким образом, эффективность принципа разделения интерфейсов достаточно наглядно продемонстрирована для классов, расположенных в различных компонентах. Следовательно, данный принцип логичнее использовать при распределении классов по различным компонентам.

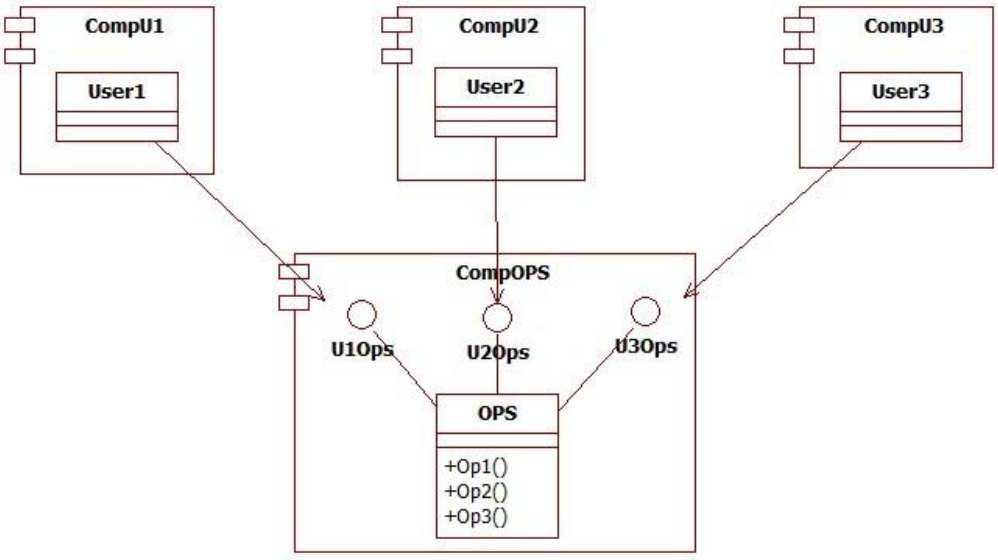

Рис. 5. Взаимодействие классов через раздельные интерфейсы

Fig. 5. Interaction of classes through segregate interfaces

Однако, с другой стороны, если класс выполняет достаточно сложную функциональность, а для полноценной ее реализации требуется предоставление ряда услуг, для независимой работы такого класса может быть определен достаточно «жирный» интерфейс. Реализацию такого интерфейса должен предоставить клиент, пользующийся услугами данного класса. При этом возможна ситуация, когда какой-либо клиент частично использует функциональность такого класса и принципиально не может обеспечить реализацию некоторых методов его «жирного» интерфейса. Таким образом, если классом возможно частичное предоставление услуг, но для этого требуется реализация «жирного» интерфейса, которую невозможно выполнить целиком всем клиентам, то такой интерфейс должен быть разделен.

Например, если в целях повышения безопасности требуются операции для переименования и (или) изменения владельца файлов различных операционных систем, то создается следующий интерфейс [4]:

```
public interface FileInterface
{
   void rename(string OldName, string NewName);
   void changeOwner(string user, string group);
}
```

Может показаться очевидным, что у файла всегда есть имя и владелец, и их можно изменить. Однако, если потребуется использовать данный интерфейс для изменения владельцев файлов облачного хранилища, что принципиально невозможно, то реализация функции change Owner в классе потомке становится также невозможной. Поэтому данный интерфейс необходимо разделить на два более узкоспециализированных интерфейса:

```
public interface FileInterface
{
   void rename(string OldName, string NewName);
}
public interface FileOwnerInterface : FileInterface
{
   void changeOwner(string user, string group);
}
```

Вместе эти интерфейсы образуют иерархию файловых типов (рис. 6). Сначала идет универсальный файловый тип, определяемый с помощью FileInterface, и содержащий только метод для переименования. Затем идет подтип файлов, владельца которого можно изменять.

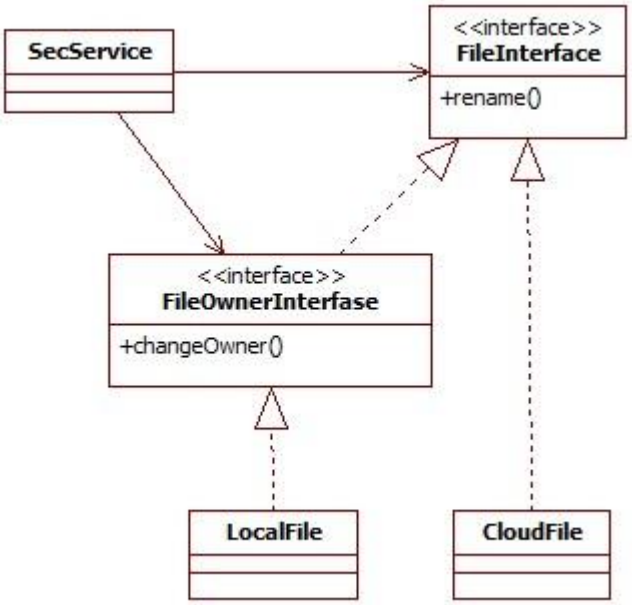

Рис. 6. Раздельные интерфейсы иерархии файловых типов

Fig. 6. Segregate interfaces of the file type hierarchy

В данном случае интерфейс FileOwnerInterface обязывает исполнять и FileInterface, поскольку предполагается, что все без исключений файлы могут быть переименованы. Однако возможна ситуация, когда части исходного интерфейса достаточно независимы, тогда разделенные интерфейсы также могут быть независимыми. В данном случае классы-реализаторы могут выбирать наследовать либо один, либо сразу два интерфейса в зависимости от их ответственности.

Первоначально идея FileInterface заключалась в том, чтобы представить некие действия («переименовать» и «изменить владельца»), которые подойдут для любого файла независимо от места его расположения, в виде единого интерфейса. Пытаясь абстрагироваться от подробностей выполнения этих операций для конкретных файлов, сделали неверное предположение, согласно которому любая реализация FileInterface сможет обеспечить значимые реализации всех методов этого интерфейса. Однако, когда потребовалась реализация FileInterface для файлов, расположенных в облаке, это предположение оказалось неверным. Таким образом, FileInterface – это неправильное обобщение понятия «файл» и его необходимо модернизировать. Поэтому, если при определении интерфейса существует вероятность, что некоторые из его методов смогут реализовать не все клиенты, такой интерфейс необходимо разделить на более специализированные интерфейсы.

Таким образом, принцип разделения интерфейсов для классов следует рассматривать как противоположность принципа LSP (Liskov Substitution Principle) в том смысле, что в LSP классы-потомки не должны преподносить сюрпризы клиентам базового класса, а согласно принципу ISP – определение интерфейсов клиентами не должно преподносить сюрпризы в виде «жирных» интерфейсов, которые смогут реализовать не все классы-потомки. Наличие такого «жирного» интерфейса неизбежно приводит к нарушению принципа LSP теми классами-реализаторами, которые принципиально не могут предложить значимую реализацию для всех его методов.

Выводы

Принцип разделения интерфейсов следует относить как к принципам проектирования классов, так и к принципам проектирования компонентов. Первоначальная формулировка наглядно и убедительно продемонстрирована на примерах при расположении классов по различным компонентам.

Как принцип проектирования классов принцип разделения интерфейсов направлен на поддержание принципа LSP. Он позволяет предотвратить нарушение принципа LSP, когда класс выполняет некоторые обязанности, для независимой реализации которых требуется выполнение дополнительных сопутствующих услуг. Разработка нескольких специализированных интерфейсов позволяет полностью или частично использовать функциональность данного класса, выбирая реализацию соответствующих интерфейсов.

Библиографический список

1. **Martin**, **R.C.** Agile software development: principles, patterns, and practices: / R.C. Martin. – Person Education, Inc, 2003. – 529 p.
2. **Мартин, Р.** Принципы, паттерны и методики гибкой разработки на языке C#: / Р. Мартин, М. Мартин – СПб.: Символ-Плюс, 2011. – 768 с.
3. **Мартин, Р.** Чистая архитектура. Искусство разработки программного обеспечения / Р. Мартин. – СПб.: Питер, 2018. — 352 с.
4. **Нобак, М.** Принципы разработки программных пакетов: Проектирование повторно используемых компонентов / М. Нобак. − М.: ДМК Пресс, 2020. – 274 с.
5. **Janssen, T.** SOLID Design Principles Explained: Interface Segregation with Code Examples [Электронный ресурс] // Режим доступа: https://stackify.com/interface-segregation-principle (дата обращения: 29.09.2021).

Дата поступления
в редакцию: 23.11.2021