# Паттерн Посредник (Mediator)

**Назначение**: Определяет объект, инкапсулирующий способ взаимодействия множества объектов.

**Другими словами**: Посредник – это клей, связывающий несколько независимых классов между собой. Он избавляет от необходимости ссылаться классам друг на друга, позволяя тем самым их независимо изменять и анализировать

## Мотивация

Самая простая реализация импорта логов может выглядеть так. Класс LogFileReader читает лог-файлы и вызывает методы класса LogSaver, который сохраняет лог-файлы для последующего полнотекстового поиска (\*).

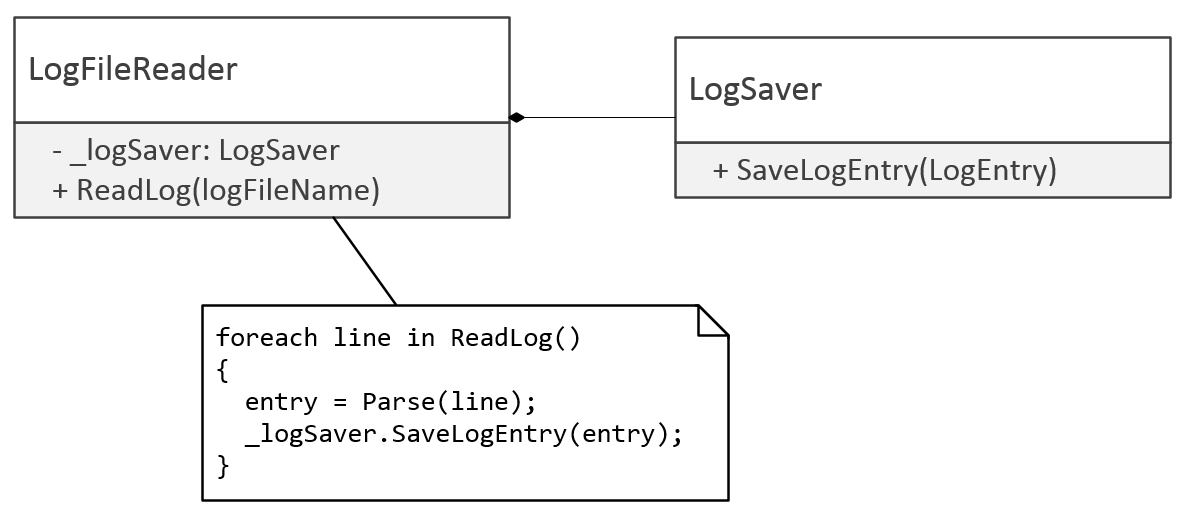


Рисунок 3.1 - Дизайн примитивного процесса импорта лог-файлов

(\*) Сноска. Рассмотренный здесь пример очень напоминает пример, приведенный Робертом Мартином в своей статье "Принцип инверсии зависимостей", которую можно найти по адресу <http://www.objectmentor.com/resources/articles/dip.pdf>. Разница лишь в том, что вместо читателя и писателя логов, Роберт использует классы Лампы (Lamp) и Кнопки (Button). Хотя задачи очень похожи, Роберт предлагает несколько иное решение, основанное на выделении интерфейса и "инверсии зависимостей". С моей точки зрения, данная задача идеально решается именно с помощью паттернов Наблюдатель и Посредник, и не требует выделения никаких дополнительных интерфейсов.

При таком подходе класс LogFileReader жестко связан с LogSaver, что не позволяет использовать класс LogFileReader повторно в другом контексте. Также это усложняет анализ и развитие класса LogFileReader, поскольку требует понимание работы его зависимостей.

Основная проблема этого дизайна в том, что он плохо моделирует задачу. Классы чтения и сохранения логов являются независимыми шагами импорта лог-файлов, и не должны знать друг о друге. Решение о "перекладывании" логов из одного источника в другой должно решаться не на уровне этих классов, а выше. И дело здесь не столько в отсутствии гибкости текущего решения, сколько в ненужной сложности и в плохом разделении ответственности. В системе можно четко выделить три аспекта: чтение логов, сохранение логов и связующее звено, которое знает, что нужно перекладывать логи именно таким образом.

Самое простое решение заключается в выделении еще одного класса: LogImporter, который будет знать о двух других классах, и заниматься импортом логов из одного источника в другой.

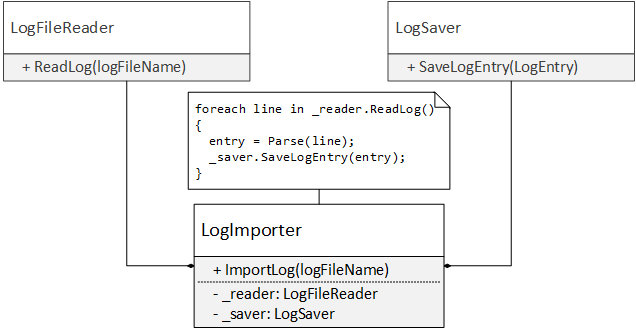


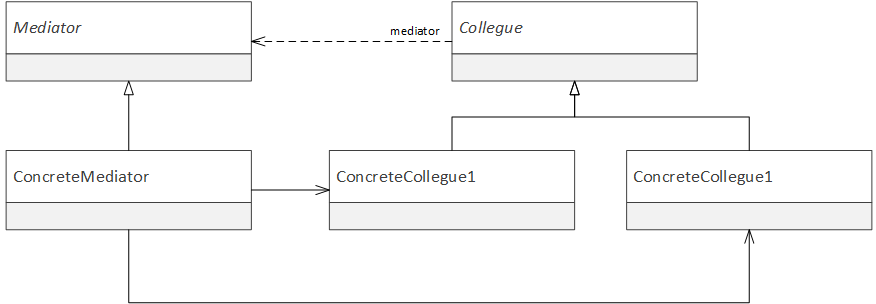
Рисунок 3.2 - Диаграмма классов импорта лог-файлов

Класс LogImporter выступает в роли посредника: он связывает воедино несколько низкоуровневых классов для обеспечения нового высокоуровневого поведения. Такой подход обеспечивает гибкость в развитии системы, хотя и не вводит полиморфного поведения. Посредник в этом случае выступает барьером, который гасит изменения в одной части системы, не давая им распространятся на другие части! Любые изменения в классе LogFileReader приведут к модификации LogImporter, но не потребует изменений класса LogFileSaver или его клиентов.

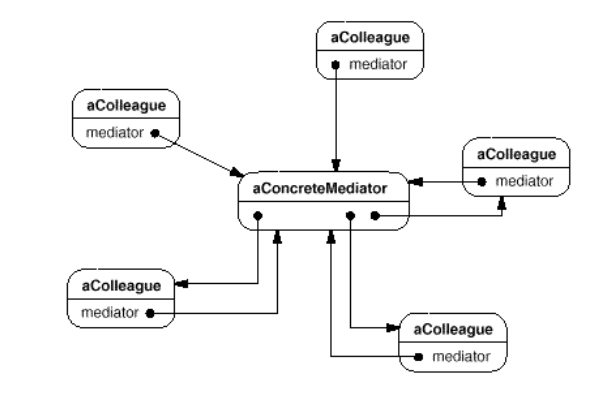
**ПРИМЕЧАНИЕ**  
Эрик Эванс в своей знаменитой книге "Domain-Driven Design. Tackling Complexity in the Heart of Software" четко разделяет понятия гибкого (flexible) и податливого (supple) дизайна. Гибкость обычно обеспечивается за счет дополнительных уровней абстракции (например, за счет полиморфизма и иерархий наследования), что неизбежно приводит к увеличению сложности. Податливый дизайн, с другой стороны, представляет собой самый простой способ решения поставленной задачи. Он не обеспечивает возможность адаптации системы к новым требования на лету, во время исполнения. Изменение поведения потребуют внесения изменений в код приложения, но поскольку дизайн прост и "податлив", то сделать это будет просто. Такой подход экономит силы и позволяет избежать одной из типичных ловушек современного проектирования: проблемы преждевременного обобщения (premature generalization).

## Классическая диаграмма классов паттерна Посредник

Как и большинство других классических паттернов, описание паттерна Посредник от банды четырех предполагает наличие наследования.



TODO: Вот диаграмма объектов из GoF-а: нужна ли она здесь?



Обычно взаимодействующие компоненты не содержат общего предка (если не считать класса System.Object), и совсем не обязательно знают о классе-посреднике. Иерархия посредников также применяется довольно редко.

**Участники**

* Mediator (LogFileImporter) определяет интерфейс Посредника. На практике базовый класс посредника выделяется редко, поэтому класс Mediator обычно содержит всю логику взаимодействия.
* ConcreteCollegue1, ConcreteCollegue2 (LogFileReader, LogFileSaver) классы одного уровня абстракции, которые взаимодействуют друг с другом косвенным образом через посредника.

## Обсуждение паттерна Посредник

Основная суть борьбы со сложность заключается в изоляции информации, чтобы у разработчика была возможность сосредоточиться на одной проблеме и игнорировать другие. Именно по этой причине мы стараемся объединять логически связанные куски системы в отдельные модули (отсюда сильная связность – high cohesion), максимально изолируя эти модули друг от друга (отсюда слабая связанность – low coupling).

При этом любая сложная система вырастает на основе более простых компонентов. Мы получаем иерархичную систему, объединяя независимые строительные блоки в более высокоуровневые абстракции. Любой класс или модуль строится на основе автономных классов/модулей более низкого уровня, обеспечивая передачу управления между ними. Любой такой класс/модуль выполняет роль посредника.

Посредник – это один из самых распространенных паттернов проектирования. Он десятками используется в любом приложении, хотя имена классов практически никогда не отражают это в своих названиях.

Если же подходить к вариантам реализации взаимодействия, то тут можно выделить две разновидности паттерна Посредник: посредник может быть явным, и о нем могут знать "объединяемые компоненты", или же он может быть невидимым для этих компонентов и объединять их между собой без их ведома.

### Явный и неявный посредник

В классической реализации паттерна Посредник, независимые классы не знают друг о друге, но знают о существовании Посредника и все взаимодействие происходит через него явным образом.

Такой подход довольно распространен, и активно применяется в паттернах Поставщик/Потребитель (Producer/Consumer), Агрегатор событий (Event Aggregator) или в других случаях, когда классы знают о существовании "общей шины взаимодействия". При этом посредник может содержать ссылки на взаимодействующие классы и "маршрутизировать" вызовы явно. Или же Посредник может быть наблюдаемым объектом и предоставлять набор событий, на которые будут подписываться коллеги для взаимодействия между собой.

С другой стороны, классы низкого уровня могут и не знать о существовании Посредника. Классы LogFileReader иLogFileSaver не знают о существовании посредника - LogImporter. Это делает их более автономными, а дизайн - более естественным. Если же одному из участников понадобиться активно управлять "процессом обмена сообщениями" (т.е. использовать push-модель взаимодействия(\*)), то достаточно сделать его наблюдаемым. В этом случае участник останется автономным и не будет знать о посреднике.

(\*) Сноска: push и pull модели взаимодействия более подробно будет рассмотрены в главе о паттерне Наблюдатель.

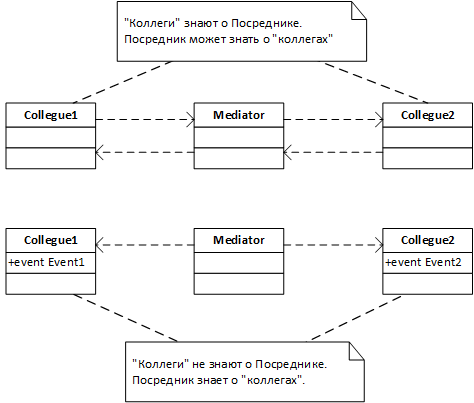


Рисунок 3.3 - Явная и неявная реализация посредника

Класс LogFileReader вполне может быть активным, следить за содержимым лог-файла и сохранять новые фрагменты по мере их поступления. В этом случае, наиболее естественное решение заключается в использовании событий или интерфейса IObservable, для активного уведомления посредника о прочитанных записях:

**ПРИМЕЧАНИЕ**  
Наличие в дизайне системы Наблюдателей говорит о наличии Посредников. Паттерн Наблюдатель будет рассмотрен в одной из последующих глав.

Именно такой подход используется в Windows Forms и других UI платформах. Форма выступает в роли Посредника: следит за событиями компонентов формы и передает управление другим компонентам или бизнес-объектам.

При наличии формы (CustomForm) с двумя элементами управления: TextBox и Button, мы можем добавить логику разрешения кнопки сохранения при вводе пользователем данных в TextBox:

public CustomForm()

{

InitializeComponent();

buttonSave.Enabled = false;

textBoxName.TextChanged += (s,ea) =>

{

buttonSave.Enabled = true;

};

}

Листинг 3.1 – Пример паттерна Наблюдатель в Windows Forms

### Явные и неявные связи

Одной из главных целей многих паттернов проектирования является получение слабосвязанного (loosely coupled) дизайна. Это позволяет развивать части системы независимо, бороться со сложность и выпускать новые версии систем со скоростью света. Однако, как и все в дизайне, слабая связанность имеет свою цену, особенно, когда с ней начинают перегибать палку. Дизайн системы, в которой никто ни о ком не знает, так же утопичен, как и дизайн, в котором каждый класс знает обо всех остальных.

Слабая связанность должна всегда идти рука об руку с сильной связностью. Хороший дизайн разрезает систему на модули, которые решают совместно некоторую задачу. При этом внутри модуля связей может быть достаточно много и вполне нормально если эти связи будут явными. А это значит, что слабую связанность нужно обеспечивать на границах модуля, а не внутри него. При этом польза отсутствия явных связей между классами может быть обманчивой.

Вместо явного и конкретного посредника (класса LogImporter), который контролирует логическое взаимодействие классов LogFileReader и LogSaver, можно было бы использовать универсальный посредник - агрегатор событий (Event Aggregator):

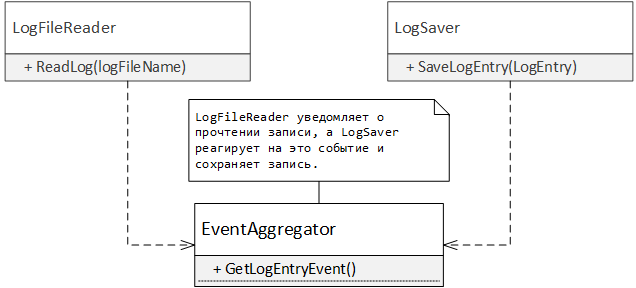


Рисунок 3.4 - Импорт логов на основе агрегатора событий

В этом случае, класс LogFileReader публикует событие о чтении/записи лог-файла, и не будет знать кто его обрабатывает. LogSaver "слушает" это событие и сохраняет записи, не зная, откуда они были прочитаны.

Обобщение простого решения, в котором нет никакой необходимости, убило не один проект. Смысл посредника в том, что он инкапсулирует в себе процесс взаимодействия объектов. Агрегатор событий тоже может выполнять эту роль, но при этом логика взаимодействия начинает расплываться.

Агрегаторы событий не устраняют связи между классами, они лишь делают их неявными. Когда в системе взаимодействуют десятки объектов, возможно, это и оправдано. Но значительно лучше начинать с явного решения и обобщать его лишь тогда, когда стало понятно, в чем заключается "обобщение" и что оно действительно нужно.

Слабосвязанный дизайн не означает то, что класс A не имеет ссылки на класс Б. Слабосвязанный дизайн говорит о том, что класс А может успешно функционировать, развиваться и изменяться никак не завися от класса Б. Наличие же неявных связей через события, интерфейсы или глобальные переменные не устраняет связи между этими классами, он лишь прячет их и делает эти связи неявными.

Если предметная область говорит о наличии связи между понятием А и понятием Б, то в дизайне системы лучше всего отразить эти отношения явным образом.

### Тестировать или не тестировать? Вот в чем вопрос!

При наличии в системах довольно большого количества Посредников может возникнуть логичный вопрос: стоит ли покрывать их юнит-тестами? У сторонников TDD (Test-Driven Development) будет готов ответ на этот вопрос, но что делать всем остальным?

С одной стороны, любая сложная логика, которая является важной с точки зрения работы приложения должна быть покрыта тестами. С другой стороны, в случае простого Посредника, вполне достаточным будет покрыть его логику интеграционными тестами и не тратить время на модульное тестирование.

Здесь, как и во многих других случаях, следует попытаться найти компромисс между трудозатратами на разработку и поддержку тестов, и выгодой от их наличия. Любая критически важная логика должна быть покрыта тестами. Точка. Но должны ли посредники содержать критически важную логику? Вполне возможно, что сам факт наличия сложной логики в посреднике говорит о том, что он делает слишком многое. Если же его "посредническая" логика сложная и важная, то стоит подумать о том, нельзя ли ее упростить.

Тестирование поведения осуществляется с помощью моков (mocks) – специального вида тестовых подделок, которые поддерживаются большинством современных тестовых фреймворков. Благодаря им, обычно не составляет особого труда написать набор тестов, которые будут проверять, что в определенных условиях тестируемый класс (CUT – Class Under Test) вызывает те или иные методы своих зависимостей. Однако с такими тестами нужно быть осторожными и уделять особое внимание тому, чтобы проверять лишь ключевую логику (например, лишь факт вызова метода зависимости, без проверки точных аргументов вызова), а также стараться избегать проверок деталей реализации.

**ПРИМЕЧАНИЕ**  
Подробнее о разнице между тестированием состояния и тестированием поведения можно прочитать в моей статье "Моки и стабы" (bit.ly/StubsVsMocks) или "Stubs are not Mocks" Мартина Фаулера (<http://martinfowler.com/articles/mocksArentStubs.html>).

### Архитектурные посредники

Паттерн Посредник может применяться на разных уровнях приложения. Существуют классы-посредники, компоненты-посредники, но есть даже целые модули или слои приложения, выполняющие эту же роль.

Очень часто слой приложения выполняет роль посредника: он объединяет воедино доменные и сервисные слои и обеспечивает поведение, уникальное для этого конкретного приложения.

Например, модули импорта/экспорта логов могут использоваться по-разному в разных приложениях. Можно создать консольное приложение, которое будет импортировать логи лишь определенных типов и запускаться по расписанию. Можно создать сервис, который будет принимать логи по сети, для последующего сохранения.

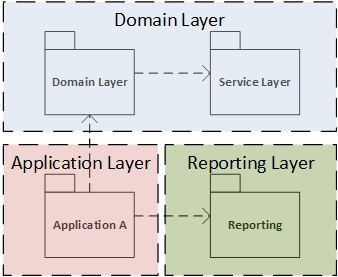


Рисунок 3.5 - Посредник на уровне слоев приложения

Основная логика по связке компонентов в готовое приложение обычно находится в точке входа приложения или недалеко от нее: в функции Main, ServiceMain или global.asax. Такая точка называется узлом композиции (Composition Root), и уникальным образом отличает одно приложение от другого. В ней может находиться логика конфигурирования IoC-контейнера или же создание явного класса-посредника.

## Применимость

Паттерн Посредник идеально подходит для объединения воедино нескольких автономных классов или компонентов. Каждый раз, когда вы задаетесь вопросом о том, как же изолировать классы А и Б, чтобы они могли жить независимо, подумайте об использовании Посредника.

При этом остается несколько вариантов решения: классы А и Б могут знать о посреднике или же один из классов может быть наблюдаемым. Выбор того или иного варианта зависит от "толщины" интерфейса взаимодействия между этими классами. Если "протокол общения" прост, то использование наблюдателя будет вполне оправданным, если же он достаточно сложен, то проще будет использовать посредник явно.

### Когда третий лишний

Как и любой другой паттерн, Посредник нужно использовать с умом. Этот паттерн изолирует изменения в одной части системы, не давая им распространяться на другие части. В некоторых случаях это упрощает внесение изменений, а в некоторых, наоборот, усложняет.

Не нужно разделять с помощью посредника тесно связанные вещи. Если процессы импорта и экспорта всегда изменяются совместно, то возможно, что они должны знать друг о друге. Наличие посредника между классами, которые всегда изменяются совместно, лишь усложнит внесение изменений: вместо изменения двух классов, придется изменять три.

## Примеры в .NET Framework

Посредник – это паттерн, который не проявляется в открытом интерфейсе модуля или библиотеки, поэтому примеры использования нужно искать в реализации .NET Framework или в пользовательском коде. Тем не менее, примеров использования паттерна Посредник в .NET приложениях довольно много.

* В Windows Forms любая форма по своей сути представляет собой Посредник, объединяющий элементы управления между собой: форма подписывается на события одного элемента управления и в случае изменения состояния уведомляет бизнес-объекты или другие элементы управления.
* Класс EventAggregator активно используемый в WPF и за его пределами и является примером "глобального" посредника для связи разных независимых компонентов между собой.
* В паттернах MVC (Model-View Controller, Модель-Представление Контроллер) и MVP (Model-View Presenter, Модель-Представление Презентер), Controller и Presenter выступают в роли посредника между представлением и моделью (\*).

(\*) Сноска: немного подробнее MVx паттерны будут рассмотрены в заключительной части книги, в главе, посвященной Принципу инверсии зависимостей.

* Паттерн Producer/Consumer (Поставщик/Потребитель) является еще одним примером паттерна Посредник. В этом паттерне Потребитель и Поставщик не знают друг о друге и общаются между собой за счет общей очереди, которая является Посредником. В случае .NET Framework таким посредником может служить BlockingCollection.