Объекты Layout создаются и управляются объектами класса Look. Класс Look – это абстрактная фабрика, которая производит объекты Layout с помощью таких операций, как GetButtonLayout, GetMenuBarLayout и т.д. Для каждого стандарта внешнего облика у класса Look есть соответствующий подкласс (MotifLook, OpenLook и т.д.).

Кстати говоря, объекты Layout – это, по существу, стратегии (см. описание паттерна стратегия). Таким образом, мы имеем пример объекта-стратегии, реализованный в виде приспособленца.

### Родственные паттерны

Паттерн приспособленец часто используется в сочетании с компоновщиком для реализации иерархической структуры в виде ациклического направленного графа с разделяемыми листовыми вершинами.

Часто наилучшим способом реализации объектов состояния и стратегии является паттерн приспособленец.

## Паттерн Proxy

### Название и классификация паттерна

Заместитель – паттерн, структурирующий объекты.

### Назначение

Является суррогатом другого объекта и контролирует доступ к нему.

### Известен также под именем

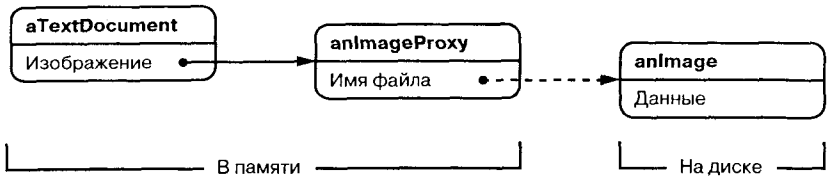
Surrogate (суррогат).

### Мотивация

Разумно управлять доступом к объекту, поскольку тогда можно отложить расходы на создание и инициализацию до момента, когда объект действительно понадобится. Рассмотрим редактор документов, который допускает встраивание в документ графических объектов. Затраты на создание некоторых таких объектов, например больших растровых изображений, могут быть весьма значительны. Но документ должен открываться быстро, поэтому следует избегать создания всех «тяжелых» объектов на стадии открытия (да и вообще это излишне, поскольку не все они будут видны одновременно).

В связи с такими ограничениями кажется разумным создавать «тяжелые» объекты *по требованию*. Это означает «когда изображение становится видимым». Но что поместить в документ вместо изображения? И как, не усложняя реализации редактора, скрыть то, что изображение создается по требованию? Например, оптимизация не должна отражаться на коде, отвечающем за рисование и форматирование.

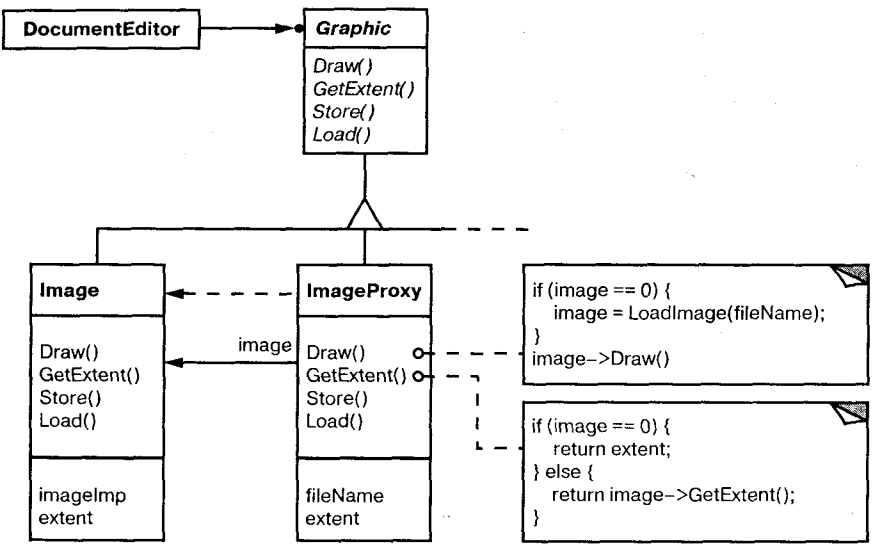
Решение состоит в том, чтобы использовать другой объект – *заместитель* изображения, который временно подставляется вместо реального изображения. Заместитель ведет себя точно так же, как само изображение, и выполняет при необходимости его инстанцирование.




**Заместитель** создает настоящее изображение, только если редактор документа вызовет операцию `Draw`. Все последующие запросы заместитель переадресует непосредственно изображению. Поэтому после создания изображения он должен сохранить ссылку на него.

Предположим, что изображения хранятся в отдельных файлах. В таком случае мы можем использовать имя файла как ссылку на реальный объект. **Заместитель** хранит также размер изображения, то есть длину и ширину. «Зная» ее, заместитель может отвечать на запросы форматера о своем размере, не инстанцируя изображение.

На следующей диаграмме классов этот пример показан более подробно.




Редактор документов получает доступ к встроенным изображениям только через интерфейс, определенный в абстрактном классе `Graphic`. `ImageProxy` – это класс для представления изображений, создаваемых по требованию. В `ImageProxy` хранится имя файла, играющее роль ссылки на изображение, которое находится на диске. Имя файла передается конструктору класса `ImageProxy`.

В объекте `ImageProxy` находятся также ограничивающий прямоугольник изображения и ссылка на экземпляр реального объекта `Image`. Ссылка остается недействительной, пока заместитель не инстанцирует реальное изображение. Операцией `Draw` гарантируется, что изображение будет создано до того, как заместитель переадресует ему запрос. Операция `GetExtent` переадресует запрос

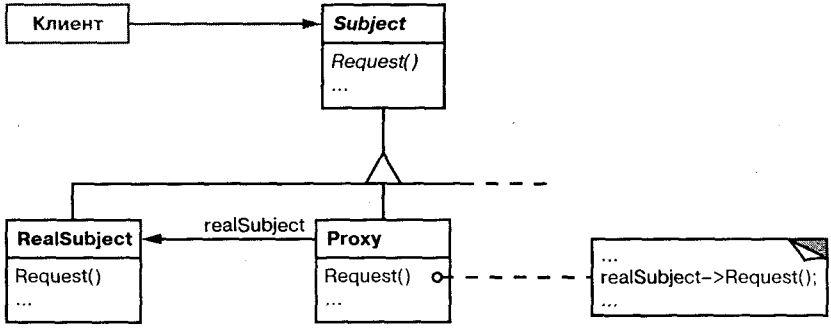
изображению, только если оно уже инстанцировано; в противном случае `ImageProxy` возвращает размеры, которые хранит сам.

### Применимость

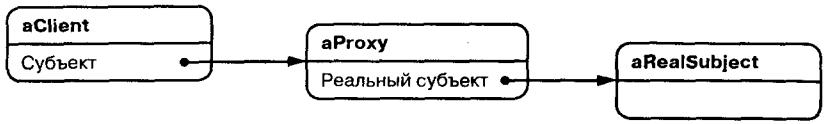
Паттерн **заместитель** применим во всех случаях, когда возникает необходимость сослаться на объект более изощренно, чем это возможно, если использовать простой указатель. Вот несколько типичных ситуаций, где **заместитель** оказывается полезным:

- *удаленный заместитель* предоставляет локального представителя вместо объекта, находящегося в другом адресном пространстве. В системе NEXTSTEP [Add94] для этой цели применяется класс `NXProxy`. Заместителя такого рода Джеймс Коплиен [Cop92] называет «послом»;
- *виртуальный заместитель* создает «тяжелые» объекты по требованию. Примером может служить класс `ImageProxy`, описанный в разделе «Мотивация»;
- *защищающий заместитель* контролирует доступ к исходному объекту. Такие заместители полезны, когда для разных объектов определены различные права доступа. Например, в операционной системе `Choices` [CIRM93] объекты `KernelProxy` ограничивают права доступа к объектам операционной системы;
- *«умная» ссылка* – это замена обычного указателя. Она позволяет выполнить дополнительные действия при доступе к объекту. К типичным применениям такой ссылки можно отнести:
  - подсчет числа ссылок на реальный объект, с тем чтобы занимаемую им память можно было освободить автоматически, когда не останется ни одной ссылки (такие ссылки называют еще «умными» указателями [Ede92]);
  - загрузку объекта в память при первом обращении к нему;
  - проверку и установку блокировки на реальный объект при обращении к нему, чтобы никакой другой объект не смог в это время изменить его.

### Структура




Вот как может выглядеть диаграмма объектов для структуры с заместителем во время выполнения.




### Участники

- **Proxy** (`ImageProxy`) – заместитель:
  - хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту. Объект класса `Proxy` может обращаться к объекту класса `Subject`, если интерфейсы классов `RealSubject` и `Subject` одинаковы;
  - предоставляет интерфейс, идентичный интерфейсу `Subject`, так что заместитель всегда может быть подставлен вместо реального субъекта;
  - контролирует доступ к реальному субъекту и может отвечать за его создание и удаление;
  - прочие обязанности зависят от вида заместителя:
    - *удаленный заместитель* отвечает за кодирование запроса и его аргументов и отправление закодированного запроса реальному субъекту в другом адресном пространстве;
    - *виртуальный заместитель* может кэшировать дополнительную информацию о реальном субъекте, чтобы отложить его создание. Например, класс `ImageProxy` из раздела «Мотивация» кэширует размеры реального изображения;
    - *защищающий заместитель* проверяет, имеет ли вызывающий объект необходимые для выполнения запроса права;
- **Subject** (`Graphic`) – субъект:
  - определяет общий для `RealSubject` и `Proxy` интерфейс, так что класс `Proxy` можно использовать везде, где ожидается `RealSubject`;
- **RealSubject** (`Image`) – реальный субъект:
  - определяет реальный объект, представленный заместителем.

### Отношения

`Proxy` при необходимости переадресует запросы объекту `RealSubject`. Детали зависят от вида заместителя.

### Результаты

С помощью паттерна **заместитель** при доступе к объекту вводится дополнительный уровень косвенности. У этого подхода есть много вариантов в зависимости от вида заместителя:

- удаленный заместитель может скрыть тот факт, что объект находится в другом адресном пространстве;
- виртуальный заместитель может выполнять оптимизацию, например создание объекта по требованию;
- защищающий заместитель и «умная» ссылка позволяют решать дополнительные задачи при доступе к объекту.

Есть еще одна оптимизация, которую паттерн **заместитель** иногда скрывает от клиента. Она называется *копированием при записи* (copy-on-write) и имеет много общего с созданием объекта по требованию. Копирование большого и сложного объекта – очень дорогая операция. Если копия не модифицировалась, то нет смысла эту цену платить. Если отложить процесс копирования, применив заместитель, то можно быть уверенным, что эта операция произойдет только тогда, когда он действительно был изменен.

Чтобы во время записи можно было копировать, необходимо подсчитывать ссылки на субъект. Копирование заместителя просто увеличивает счетчик ссылок. И только тогда, когда клиент запрашивает операцию, изменяющую субъект, заместитель действительно выполняет копирование. Одновременно заместитель должен уменьшить счетчик ссылок. Когда счетчик ссылок становится равным нулю, субъект уничтожается.

Копирование при записи может существенно уменьшить плату за копирование «тяжелых» субъектов.

## Реализация

При реализации паттерна **заместитель** можно использовать следующие возможности языка:

- *перегрузку оператора доступа к членам в C++*. Язык C++ поддерживает перегрузку оператора доступа к членам класса `->`. Это позволяет производить дополнительные действия при любом разыменовании указателя на объект. Для реализации некоторых видов заместителей это оказывается полезно, поскольку заместитель ведет себя аналогично указателю.

  В следующем примере показано, как воспользоваться данным приемом для реализации виртуального заместителя `ImagePtr`:

```
class Image;
extern Image* LoadAnImageFile(const char*);
    // внешняя функция

class ImagePtr {
public:
    ImagePtr(const char* imageFile);
    virtual ~ImagePtr();

    virtual Image* operator->();
    virtual Image& operator*();
private:
    Image* LoadImage();
private:
    Image* _image;
    const char* _imageFile;
};

ImagePtr::ImagePtr (const char* theImageFile) {
    _imageFile = theImageFile;
    _image = 0;
}
```

```
Image* ImagePtr::LoadImage () {
    if (_image == 0) {
        _image = LoadAnImageFile(_imageFile);
    }
    return _image;
}
```

Перегруженные операторы -> и * используют операцию LoadImage для возврата клиенту изображения, хранящегося в переменной _image (при необходимости загрузив его):

```
Image* ImagePtr::operator-> () {
    return LoadImage();
}

Image& ImagePtr::operator* () {
    return *LoadImage();
}
```

Такой подход позволяет вызывать операции объекта Image через объекты ImagePtr, не заботясь о том, что они не являются частью интерфейса данного класса:

```
ImagePtr image = ImagePtr("anImageFileName");
image->Draw(Point(50, 100));
    // (image.operator->())->Draw(Point(50, 100))
```

Обратите внимание, что заместитель изображения ведет себя подобно указателю, но не объявлен как указатель на Image. Это означает, что использовать его в точности как настоящий указатель на Image нельзя. Поэтому при таком подходе клиентам следует трактовать объекты Image и ImagePtr поразному.

Перегрузка оператора доступа – лучшее решение далеко не для всех видов заместителей. Некоторым из них должно быть точно известно, *какая* операция вызывается, а в таких случаях перегрузка оператора доступа не работает. Рассмотрим пример виртуального заместителя, обсуждавшийся в разделе «Мотивация». Изображение нужно загружать в точно определенное время – при вызове операции Draw, а не при каждом обращении к нему. Перегрузка оператора доступа не позволяет различить подобные случаи. В такой ситуации придется вручную реализовать каждую операцию заместителя, переадресующую запрос субъекту.

Обычно все эти операции очень похожи друг на друга, как видно из примера кода в одноименном разделе. Они проверяют, что запрос корректен, что объект-адресат существует и т.д., а потом уже перенаправляют ему запрос. Писать этот код снова и снова надоедает. Поэтому нередко для его автоматической генерации используют препроцессор;

- *метод doesNotUnderstand в Smalltalk.* В языке Smalltalk есть возможность, позволяющая автоматически поддержать переадресацию запросов. При отправлении клиентом сообщения, для которого у получателя нет соответствующего метода, Smalltalk вызывает метод doesNotUnderstand: aMessage.

Заместитель может переопределить `doesNotUnderstand` так, что сообщение будет переадресовано субъекту.

Дабы гарантировать, что запрос будет перенаправлен субъекту, а не просто тихо поглощен заместителем, класс `Proxy` можно определить так, что он не станет понимать *никаких* сообщений. Smalltalk позволяет это сделать, надо лишь, чтобы у `Proxy` не было суперкласса[1].

Главный недостаток метода `doesNotUnderstand:` в том, что в большинстве Smalltalk-систем имеется несколько специальных сообщений, обрабатываемых непосредственно виртуальной машиной, а в этом случае стандартный механизм поиска методов обходится. Правда, единственной такой операцией, написанной в классе `Object` (следовательно, могущей затронуть заместителей), является тождество `==`.

Если вы собираетесь применять `doesNotUnderstand:` для реализация заместителя, то должны как-то решить вышеописанную проблему. Нельзя же ожидать, что совпадение заместителей – это то же самое, что и совпадение реальных субъектов. К сожалению, `doesNotUnderstand:` изначально создавался для обработки ошибок, а не для построения заместителей, поэтому его быстродействие оставляет желать лучшего;

- *заместителю не всегда должен быть известен тип реального объекта.* Если класс `Proxy` может работать с субъектом только через его абстрактный интерфейс, то не нужно создавать `Proxy` для каждого класса реального субъекта `RealSubject`; заместитель может обращаться к *любому* из них единообразно. Но если заместитель должен инстанцировать реальных субъектов (как обстоит дело в случае виртуальных заместителей), то знание конкретного класса обязательно.

К проблемам реализации можно отнести и решение вопроса о том, как обращаться к еще не инстанцированному субъекту. Некоторые заместители должны обращаться к своим субъектам вне зависимости от того, где они находятся – диске или в памяти. Это означает, что нужно использовать какую-то форму не зависящих от адресного пространства идентификаторов объектов. В разделе «Мотивация» для этой цели использовалось имя файла.

### Пример кода

В коде реализовано два вида заместителей: виртуальный, описанный в разделе «Мотивация», и реализованный с помощью метода `doesNotUnderstand:`.[2]

- *виртуальный заместитель.* В классе `Graphic` определен интерфейс для графических объектов:

```
class Graphic {
public:
    virtual ~Graphic();
```

[1] Эта техника используется при реализации распределенных объектов в системе NEXTSTEP [Add94] (точнее, в классе `NXProxy`). Только там переопределяется метод `forward` – эквивалент описанного только что приема в Smalltalk.

[2] Еще один вид заместителя дает паттерн **итератор**.

```
    virtual void Draw(const Point& at) = 0;
    virtual void HandleMouse(Event& event) = 0;

    virtual const Point& GetExtent() = 0;

    virtual void Load(istream& from) = 0;
    virtual void Save(ostream& to) = 0;
protected:
    Graphic();
};
```

Класс `Image` реализует интерфейс `Graphic` для отображения файлов изображений. В нем замещена операция `HandleMouse`, посредством которой пользователь может интерактивно изменять размер изображения:

```
class Image : public Graphic {
public:
    Image(const char* file); // загрузка изображения из файла
    virtual ~Image();

    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Point& GetExtent();

    virtual void Load(istream& from);
    virtual void Save(ostream& to);
private:
    // ...
};
```

Класс `ImageProxy` имеет тот же интерфейс, что и `Image`:

```
class ImageProxy : public Graphic {
public:
    ImageProxy(const char* imageFile);
    virtual ~ImageProxy();

    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Point& GetExtent();

    virtual void Load(istream& from);
    virtual void Save(ostream& to);
protected:
    Image* GetImage();
private:
    Image* _image;
    Point _extent;
    char* _fileName;
};
```

Конструктор сохраняет локальную копию имени файла, в котором хранится изображение, и инициализирует члены _extent и _image:

```
ImageProxy::ImageProxy (const char* fileName) {
    _fileName = strdup(fileName);
    _extent = Point::Zero; // размеры пока не известны
    _image = 0;
}

Image* ImageProxy::GetImage() {
    if (_image == 0) {
        _image = new Image(_fileName);
    }
    return _image;
}
```

Реализация операции GetExtent возвращает кэшированный размер, если это возможно. В противном случае изображение загружается из файла. Операция Draw загружает изображение, а HandleMouse перенаправляет событие реальному изображению:

```
const Point& ImageProxy::GetExtent () {
    if (_extent == Point::Zero) {
        _extent = GetImage()->GetExtent();
    }
    return _extent;
}
void ImageProxy::Draw (const Point& at) {
    GetImage()->Draw(at);
}

void ImageProxy::HandleMouse (Event& event) {
    GetImage()->HandleMouse(event);
}
```

Операция Save записывает кэшированный размер изображения и имя файла в поток, а Load считывает эту информацию и инициализирует соответствующие члены:

```
void ImageProxy::Save (ostream& to) {
    to << _extent << _fileName;
}

void ImageProxy::Load (istream& from) {
    from >> _extent >> _fileName;
}
```

Наконец, предположим, что есть класс TextDocument для представления документа, который может содержать объекты класса Graphic:

```
class TextDocument {
public:
    TextDocument();
```

```
    void Insert(Graphic*);
    // ...
};
```

Мы можем вставить объект ImageProxy в документ следующим образом:

```
TextDocument* text = new TextDocument;
// ...
text->Insert(new ImageProxy("anImageFileName"));
```

- *заместители, использующие метод doesNotUnderstand.* В языке Smalltalk можно создавать обобщенных заместителей, определяя классы, для которых нет суперкласса[1], а в них – метод doesNotUnderstand: для обработки сообщений.

  В показанном ниже фрагменте предполагается, что у заместителя есть метод realSubject, возвращающий связанный с ним реальный субъект. При использовании ImageProxy этот метод должен был бы проверить, создан ли объект Image, при необходимости создать его и затем вернуть. Для обработки перехваченного сообщения, которое было адресовано реальному субъекту, используется метод perform:withArguments:.

```
doesNotUnderstand: aMessage
    ^ self realSubject
        perform: aMessage selector
        withArguments: aMessage arguments
```

  Аргументом doesNotUnderstand: является экземпляр класса Message, представляющий сообщение, не понятое заместителем. Таким образом, при ответе на любое сообщение заместитель сначала проверяет, что реальный субъект существует, а потом уже переадресует ему сообщение.

  Одно из преимуществ метода doesNotUnderstand: – он способен выполнить произвольную обработку. Например, можно было бы создать защищающего заместителя, определив набор legalMessages-сообщений, которые следует принимать, и снабдив заместителя следующим методом:

```
doesNotUnderstand: aMessage
    ^ (legalMessages includes: aMessage selector)
        ifTrue: [self realSubject
            perform: aMessage selector
            withArguments: aMessage arguments]
        ifFalse: [self error: 'Illegal operator']
```

  Прежде чем переадресовать сообщение реальному субъекту, указанный метод проверяет, что оно допустимо. Если это не так, doesNotUnderstand: посылает сообщение error: самому себе, что приведет к зацикливанию, если в заместителе не определен метод error:. Следовательно, определение error: должно быть скопировано из класса Object вместе со всеми методами, которые в нем используются.

[1] Практически для любого класса Object является суперклассом самого верхнего уровня. Поэтому выражение «нет суперкласса» означает то же самое, что «Object не является суперклассом».

### Известные применения

Пример виртуального заместителя из раздела «Мотивация» заимствован из классов строительного блока текста, определенных в каркасе ET++.

В системе NEXTSTEP [Add94] заместители (экземпляры класса `NXProxy`) используются как локальные представители объектов, которые могут быть распределенными. Сервер создает заместителей для удаленных объектов, когда клиент их запрашивает. Заместитель кодирует полученное сообщение вместе со всеми аргументами, после чего отправляет его удаленному субъекту. Аналогично субъект кодирует возвращенные результаты и посылает их обратно объекту `NXProxy`.

В работе McCullough [McC87] обсуждается применение заместителей в Smalltalk для доступа к удаленным объектам. Джефри Пэско (Geoffrey Pascoe) [Pas86] описывает, как обеспечить побочные эффекты при вызове методов и реализовать контроль доступа с помощью «инкапсуляторов».

### Родственные паттерны

Паттерн адаптер предоставляет другой интерфейс к адаптируемому объекту. Напротив, заместитель в точности повторяет интерфейс своего субъекта. Однако, если заместитель используется для ограничения доступа, он может отказаться выполнять операцию, которую субъект выполнил бы, поэтому на самом деле интерфейс заместителя может быть и подмножеством интерфейса субъекта.

Несколько замечаний относительно декоратора. Хотя его реализация и похожа на реализацию заместителя, но назначение совершенно иное. Декоратор добавляет объекту новые обязанности, а заместитель контролирует доступ к объекту.

Степень схожести реализации заместителей и декораторов может быть различной. Защищающий заместитель мог бы быть реализован в точности как декоратор. С другой стороны, удаленный заместитель не содержит прямых ссылок на реальный субъект, а лишь косвенную ссылку, что-то вроде «идентификатор хоста и локальный адрес на этом хосте». Вначале виртуальный заместитель имеет только косвенную ссылку (скажем, имя файла), но в конечном итоге получает и использует прямую ссылку.

## Обсуждение структурных паттернов

Возможно, вы обратили внимание на то, что структурные паттерны похожи между собой, особенно когда речь идет об их участниках и взаимодействиях. Вероятное объяснение такому явлению: все структурные паттерны основаны на небольшом множестве языковых механизмов структурирования кода и объектов (одиночном и множественном наследовании для паттернов уровня класса и композиции для паттернов уровня объектов). Но имеющееся сходство может быть обманчиво, ибо с помощью разных паттернов можно решать совершенно разные задачи. В этом разделе сопоставлены группы структурных паттернов, и вы сможете яснее почувствовать их сравнительные достоинства и недостатки.

### Адаптер и мост

У паттернов адаптер и мост есть несколько общих атрибутов. Тот и другой повышают гибкость, вводя дополнительный уровень косвенности при обращении