Полиморфные итераторы и выполняющие очистку заместители находятся в контейнерных классах ET++ [WGM88]. Курсороподобные итераторы используются в классах каркаса графических редакторов Unidraw [VL90].

В системе ObjectWindows 2.0 [Bor94] имеется иерархия классов итераторов для контейнеров. Контейнеры разных типов можно обходить одним и тем же способом. Синтаксис итераторов в ObjectWindows основан на перегрузке постфиксного оператора инкремента ++ для перехода к следующему элементу.

Родственные паттерны

Компоновщик: итераторы довольно часто применяются для обхода рекурсивных структур, создаваемых компоновщиком.

Фабричный метод: полиморфные итераторы поручают фабричным методам инстанцировать подходящие подклассы класса `Iterator`.

Итератор может использовать хранитель для сохранения состояния итерации и при этом содержит его внутри себя.

Паттерн Mediator

Название и классификация паттерна

Посредник – паттерн поведения объектов.

Назначение

Определяет объект, инкапсулирующий способ взаимодействия множества объектов. Посредник обеспечивает слабую связанность системы, избавляя объекты от необходимости явно ссылаться друг на друга и позволяя тем самым независимо изменять взаимодействия между ними.

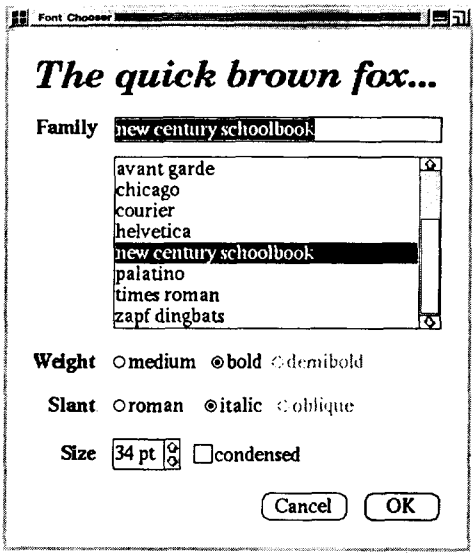
Мотивация

Объектно-ориентированное проектирование способствует распределению некоторого поведения между объектами. Но при этом в получившейся структуре объектов может возникнуть много связей или (в худшем случае) каждому объекту придется иметь информацию обо всех остальных.

Несмотря на то что разбиение системы на множество объектов в общем случае повышает степень повторного использования, однако изобилие взаимосвязей приводит к обратному эффекту. Если взаимосвязей слишком много, тогда система подобна монолиту и маловероятно, что объект сможет работать без поддержки других объектов. Более того, существенно изменить поведение системы практически невозможно, поскольку оно распределено между многими объектами. Если вы предпримете подобную попытку, то для настройки поведения системы вам придется определять множество подклассов.

Рассмотрим реализацию диалоговых окон в графическом интерфейсе пользователя. Здесь располагается ряд виджетов: кнопки, меню, поля ввода и т.д., как показано на рисунке.

Часто между разными виджетами в диалоговом окне существуют зависимости. Например, если одно из полей ввода пустое, то определенная кнопка недоступна. При выборе из списка может измениться содержимое поля ввода. И наоборот,

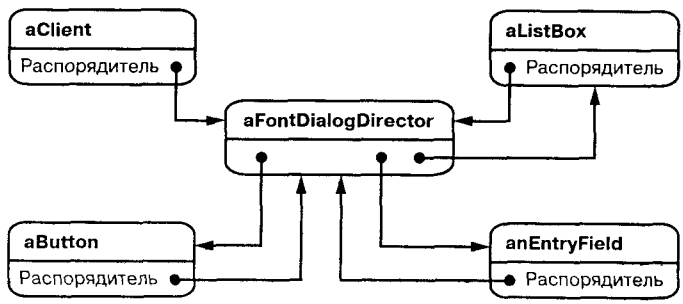


ввод текста в некоторое поле может автоматически привести к выбору одного или нескольких элементов списка. Если в поле ввода присутствует какой-то текст, то могут быть активизированы кнопки, позволяющие произвести определенное действие над этим текстом, например изменить либо удалить его.

В разных диалоговых окнах зависимости между виджетами могут быть различными. Поэтому, несмотря на то что во всех окнах встречаются однотипные виджеты, просто взять и повторно использовать готовые классы виджетов не удастся, придется производить настройку с целью учета зависимостей. Индивидуальная настройка каждого виджета – утомительное занятие, ибо участвующих классов слишком много.

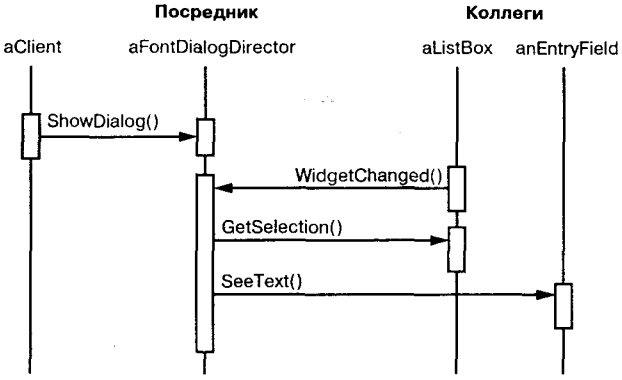
Всех этих проблем можно избежать, если инкапсулировать коллективное поведение в отдельном объекте-посреднике. Посредник отвечает за координацию взаимодействий между группой объектов. Он избавляет входящие в группу объекты от необходимости явно ссылаться друг на друга. Все объекты располагают информацией только о посреднике, поэтому количество взаимосвязей сокращается.

Так, класс `FontDialogDirector` может служить посредником между виджетами в диалоговом окне. Объект этого класса «знает» обо всех виджетах в окне



и координирует взаимодействие между ними, то есть выполняет функции центра коммуникаций.

На следующей диаграмме взаимодействий показано, как объекты кооперируются друг с другом, реагируя на изменение выбранного элемента списка.

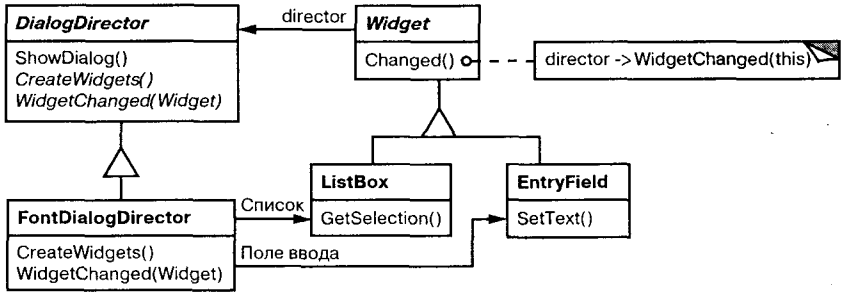


Последовательность событий, в результате которых информация о выбранном элемента списка передается в поле ввода, следующая:

1. Список информирует распорядителя о происшедших в нем изменениях.
2. Распорядитель получает от списка выбранный элемент.
3. Распорядитель передает выбранный элемент полю ввода.
4. Теперь, когда поле ввода содержит какую-то информацию, распорядитель активизирует кнопки, позволяющие выполнить определенное действие (например, изменить шрифт на полужирный или курсив).

Обратите внимание на то, как распорядитель осуществляет посредничество между списком и полем ввода. Виджеты общаются друг с другом не напрямую, а через распорядитель. Им вообще не нужно владеть информацией друг о друге, они осведомлены лишь о существовании распорядителя. А коль скоро поведение локализовано в одном классе, то его несложно модифицировать или сделать совершенно другим путем расширения или замены этого класса.

Абстракцию `FontDialogDirector` можно было бы интегрировать в библиотеку классов так, как показано на рисунке.



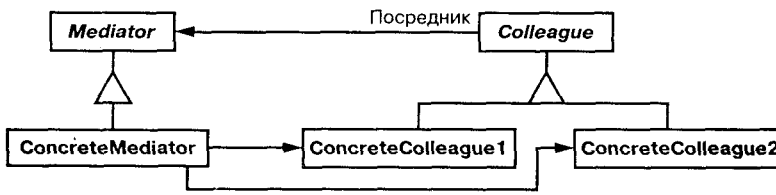
`DialogDirector` – это абстрактный класс, который определяет поведение диалогового окна в целом. Клиенты вызывают его операцию `ShowDialog` для отображения окна на экране. `CreateWidgets` – это абстрактная операция для создания виджетов в диалоговом окне. `WidgetChanged` – еще одна абстрактная операция; с ее помощью виджеты сообщают распорядителю об изменениях. Подклассы `DialogDirector` замещают операции `CreateWidgets` (для создания нужных виджетов) и `WidgetChanged` (для обработки извещений об изменениях).

Применимость

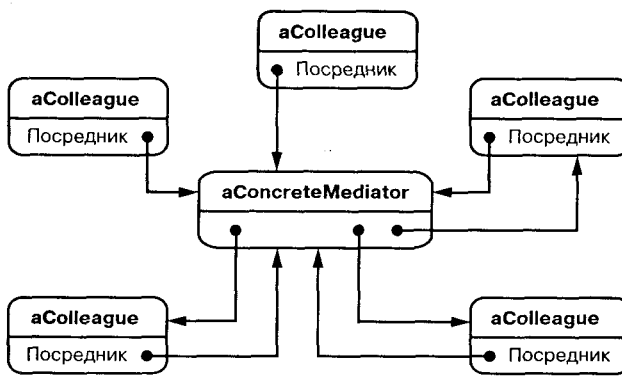
Используйте паттерн посредник, когда

- имеются объекты, связи между которыми сложны и четко определены. Получающиеся при этом взаимозависимости не структурированы и трудны для понимания;
- нельзя повторно использовать объект, поскольку он обменивается информацией со многими другими объектами;
- поведение, распределенное между несколькими классами, должно поддаваться настройке без порождения множества подклассов.

Структура



Типичная структура объектов.



Участники

- **Mediator** (`DialogDirector`) – посредник:
  - определяет интерфейс для обмена информацией с объектами `Colleague`;

- **ConcreteMediator** (`FontDialogDirector`) – конкретный посредник:
  - реализует кооперативное поведение, координируя действия объектов `Colleague`;
  - владеет информацией о коллегах и подсчитывает их;
- **Классы Colleague** (`ListBox`, `EntryField`) – коллеги:
  - каждый класс `Colleague` «знает» о своем объекте `Mediator`;
  - все коллеги обмениваются информацией только с посредником, так как при его отсутствии им пришлось бы общаться между собой напрямую.

Отношения

Коллеги посылают запросы посреднику и получают запросы от него. Посредник реализует кооперативное поведение путем переадресации каждого запроса подходящему коллеге (или нескольким коллегам).

Результаты

У паттерна посредник есть следующие достоинства и недостатки:

- *снижает число порождаемых подклассов.* Посредник локализует поведение, которое в противном случае пришлось бы распределять между несколькими объектами. Для изменения поведения нужно породить подклассы только от класса посредника `Mediator`, классы коллег `Colleague` можно использовать повторно без каких бы то ни было изменений;
- *устраняет связанность между коллегами.* Посредник обеспечивает слабую связанность коллег. Изменять классы `Colleague` и `Mediator` можно независимо друг от друга;
- *упрощает протоколы взаимодействия объектов.* Посредник заменяет дисциплину взаимодействия «все со всеми» дисциплиной «один со всеми», то есть один посредник взаимодействует со всеми коллегами. Отношения вида «один ко многим» проще для понимания, сопровождения и расширения;
- *абстрагирует способ кооперирования объектов.* Выделение механизма посредничества в отдельную концепцию и инкапсуляция ее в одном объекте позволяет сосредоточиться именно на взаимодействии объектов, а не на их индивидуальном поведении. Это дает возможность прояснить имеющиеся в системе взаимодействия;
- *централизует управление.* Паттерн посредник переносит сложность взаимодействия в класс-посредник. Поскольку посредник инкапсулирует протоколы, то он может быть сложнее отдельных коллег. В результате сам посредник становится монолитом, который трудно сопровождать.

Реализация

Имейте в виду, что при реализации паттерна посредник может происходить:

- *избавление от абстрактного класса Mediator.* Если коллеги работают только с одним посредником, то нет необходимости определять абстрактный класс `Mediator`. Обеспечиваемая классом `Mediator` абстракция позволяет коллегам работать с разными подклассами класса `Mediator` и наоборот;

- *обмен информацией между коллегами и посредником.* Коллеги должны обмениваться информацией со своим посредником только тогда, когда возникает представляющее интерес событие. Одним из подходов к реализации посредника является применение паттерна наблюдатель. Тогда классы коллег действуют как субъекты, посылающие извещения посреднику о любом изменении своего состояния. Посредник реагирует на них, сообщая об этом другим коллегам.

  Другой подход: в классе `Mediator` определяется специализированный интерфейс уведомления, который позволяет коллегам обмениваться информацией более свободно. В Smalltalk/V для Windows применяется некоторая форма делегирования: общаясь с посредником, коллега передает себя в качестве аргумента, давая посреднику возможность идентифицировать отправителя. Об этом подходе рассказывается в разделе «Пример кода», а о реализации в Smalltalk/V – в разделе «Известные применения».

Пример кода

Для создания диалогового окна, обсуждавшегося в разделе «Мотивация», воспользуемся классом `DialogDirector`. Абстрактный класс `DialogDirector` определяет интерфейс распорядителей:

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

    virtual void ShowDialog();
    virtual void WidgetChanged(Widget*) = 0;

protected:
    DialogDirector();
    virtual void CreateWidgets() = 0;
};
```

`Widget` – это абстрактный базовый класс для всех виджетов. Он располагает информацией о своем распорядителе:

```
class Widget {
public:
    Widget(DialogDirector*);
    virtual void Changed();

    virtual void HandleMouse(MouseEvent& event);
    // ...
private:
    DialogDirector* _director;
};
```

`Changed` вызывает операцию распорядителя `WidgetChanged`. С ее помощью виджеты информируют своего распорядителя о происшедших с ними изменениях:

```
void Widget::Changed () {
    _director->WidgetChanged(this);
}
```

В подклассах DialogDirector переопределена операция WidgetChanged для воздействия на нужные виджеты. Виджет передает ссылку на самого себя в качестве аргумента WidgetChanged, чтобы распорядитель имел информацию об изменившемся виджете. Подклассы DialogDirector переопределяют исключительно виртуальную функцию CreateWidgets для размещения в диалоговом окне нужных виджетов.

ListBox, EntryField и Button – это подклассы Widget для специализированных элементов интерфейса. В классе ListBox есть операция GetSelection для получения текущего множества выделенных элементов, а в классе EntryField – операция SetText для помещения текста в поле ввода:

```
class ListBox : public Widget {
public:
    ListBox(DialogDirector*);

    virtual const char* GetSelection();
    virtual void SetList(List<char*>* listItems);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

class EntryField : public Widget {
public:
    EntryField(DialogDirector*);

    virtual void SetText(const char* text);
    virtual const char* GetText();
    virtual void HandleMouse(MouseEvent& event);
    // ...
};
```

Операция Changed вызывается при нажатии кнопки Button (простой виджет). Это происходит в операции обработки событий мыши HandleMouse:

```
class Button : public Widget {
public:
    Button(DialogDirector*);

    virtual void SetText(const char* text);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

void Button::HandleMouse (MouseEvent& event) {
    // ...
    Changed();
}
```

Класс FontDialogDirector является посредником между всеми виджетами в диалоговом окне. FontDialogDirector – это подкласс класса DialogDirector:

```
class FontDialogDirector : public DialogDirector {
public:
    FontDialogDirector();
    virtual ~FontDialogDirector();
    virtual void WidgetChanged(Widget*);

protected:
    virtual void CreateWidgets();

private:
    Button* _ok;
    Button* _cancel;
    ListBox* _fontList;
    EntryField* _fontName;
};
```

FontDialogDirector отслеживает все виджеты, которые ранее поместил в диалоговое окно. Переопределенная в нем операция CreateWidgets создает виджеты и инициализирует ссылки на них:

```
void FontDialogDirector::CreateWidgets () {
    _ok = new Button(this);
    _cancel = new Button(this);
    _fontList = new ListBox(this);
    _fontName = new EntryField(this);

    // поместить в список названия шрифтов

    // разместить все виджеты в диалоговом окне
}
```

Операция WidgetChanged обеспечивает правильную совместную работу виджетов:

```
void FontDialogDirector::WidgetChanged (
    Widget* theChangedWidget
) {
    if (theChangedWidget == _fontList) {
        _fontName->SetText(_fontList->GetSelection());

    } else if (theChangedWidget == _ok) {
        // изменить шрифт и уничтожить диалоговое окно
        // ...

    } else if (theChangedWidget == _cancel) {
        // уничтожить диалоговое окно
    }
}
```

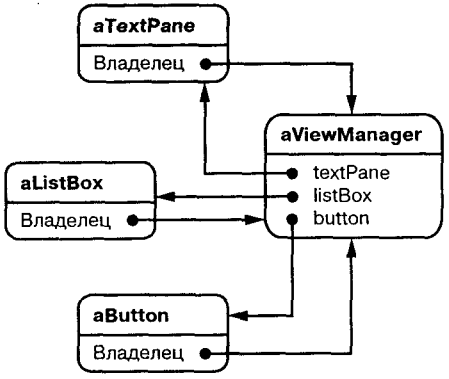
Сложность операции `WidgetChanged` возрастает пропорционально сложности окна диалога. Создание очень больших диалоговых окон нежелательно и по другим причинам, но в других приложениях сложность посредника может свести на нет его преимущества.

Известные применения

И в ET++ [WGM88], и в библиотеке классов THINK C [Sym93b] применяются похожие на нашего распорядителя объекты для осуществления посредничества между виджетами в диалоговых окнах.

Архитектура приложения в Smalltalk/V для Windows основана на структуре посредника [LaL94]. В этой среде приложение состоит из окна `Window`, которое содержит набор панелей. В библиотеке есть несколько предопределенных объектов-панелей `Pane`, например: `TextPane`, `ListBox`, `Button` и т.д. Их можно использовать без подклассов. Разработчик приложения порождает подклассы только от класса `ViewManager` (диспетчер видов), отвечающего за обмен информацией между панелями. `ViewManager` – это посредник, каждая панель «знает» своего диспетчера, который считается «владельцем» панели. Панели не ссылаются друг на друга напрямую.

На изображенной диаграмме объектов показан мгновенный снимок работающего приложения.



В Smalltalk/V для обмена информацией между объектами `Pane` и `ViewManager` используется механизм событий. Панель генерирует событие для получения данных от своего посредника или для информирования его о чем-то важном. С каждым событием связан символ (например, `#select`), который однозначно его идентифицирует. Диспетчер видов регистрирует вместе с панелью селектор метода, который является обработчиком события. Из следующего фрагмента кода видно, как объект `ListPane` создается внутри подкласса `ViewManager` и как `ViewManager` регистрирует обработчик события `#select`:

```
self addSubpane: (ListPane new
    paneName: 'myListPane';
    owner: self;
    when: #select perform: #listSelect:).
```

При координации сложных обновлений также требуется паттерн посредник. Примером может служить класс `ChangeManager`, упомянутый в описании паттерна наблюдатель. Этот класс осуществляет посредничество между субъектами и наблюдателями, чтобы не делать лишних обновлений. Когда объект изменяется, он извещает `ChangeManager`, который координирует обновление и информирует все необходимые объекты.

Аналогичным образом посредник применяется в графических редакторах Unidraw [VL90], где используется класс `CSolver`, следящий за соблюдением ограничений связанности между коннекторами. Объекты в графических редакторах могут быть визуально соединены между собой различными способами. Коннекторы полезны в приложениях, которые автоматически поддерживают связанность, например в редакторах диаграмм и в системах проектирования электронных схем. Класс `CSolver` является посредником между коннекторами. Он разрешает ограничения связанности и обновляет позиции коннекторов так, чтобы отразить изменения.

Родственные паттерны

Фасад отличается от посредника тем, что абстрагирует некоторую подсистему объектов для предоставления более удобного интерфейса. Его протокол однонаправленный, то есть объекты фасада направляют запросы классам подсистемы, но не наоборот. Посредник же обеспечивает совместное поведение, которое объекты-коллеги не могут или не «хотят» реализовывать, и его протокол двунаправленный.

Коллеги могут обмениваться информацией с посредником посредством паттерна наблюдатель.

Паттерн Memento

Название и классификация паттерна

Хранитель – паттерн поведения объектов.

Назначение

Не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние так, чтобы позднее можно было восстановить в нем объект.

Известен также под именем

Token (лексема).

Мотивация

Иногда необходимо тем или иным способом зафиксировать внутреннее состояние объекта. Такая потребность возникает, например, при реализации контрольных точек и механизмов отката, позволяющих пользователю отменить пробную операцию или восстановить состояние после ошибки. Его необходимо где-то сохранить, чтобы позднее восстановить в нем объект. Но обычно объекты инкапсулируют все