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.

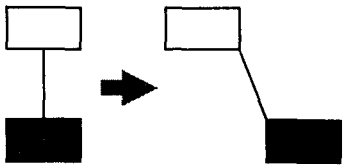
Известен также под именем

Token (лексема).

Мотивация

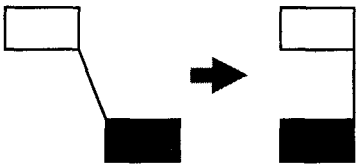
Иногда необходимо тем или иным способом зафиксировать внутреннее состояние объекта. Такая потребность возникает, например, при реализации контрольных точек и механизмов отката, позволяющих пользователю отменить пробную операцию или восстановить состояние после ошибки. Его необходимо где-то сохранить,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

свое состояние или хотя бы его часть, делая его недоступным для других объектов, так что сохранить состояние извне невозможно. Раскрытие же состояния явилось бы нарушением принципа инкапсуляции и поставило бы под угрозу надежность и расширяемость приложения.



Рассмотрим, например, графический редактор, который поддерживает связанность объектов. Пользователь может соединить два прямоугольника линией, и они останутся в таком положении при любых перемещениях. Редактор сам перерисовывает линию, сохраняя связанность конфигурации.

Система разрешения ограничений – хорошо известный способ поддержания связанности между объектами. Ее функции могут выполняться объектом класса `ConstraintSolver`, который регистрирует вновь создаваемые соединения и генерирует описывающие их математические уравнения. А когда пользователь каким-то образом модифицирует диаграмму, объект решает эти уравнения. Результаты вычислений объект `ConstraintSolver` использует для перерисовки графики так, чтобы были сохранены все соединения.



Поддержка отката операций в приложениях не так проста, как может показаться на первый взгляд. Очевидный способ откатить операцию перемещения – это сохранить расстояние между старым и новым положением, а затем переместить объект на такое же расстояние назад. Однако при этом не гарантируется, что все объекты окажутся там же, где находились. Предположим, что в способе расположения соединительной линии есть некоторая свобода. Тогда, переместив прямоугольник на прежнее место, мы можем не добиться желаемого эффекта.

Открытого интерфейса `ConstraintSolver` иногда не хватает для точного отката всех изменений смежных объектов. Механизм отката должен работать в тесном взаимодействии с `ConstraintSolver` для восстановления предыдущего состояния, но необходимо также позаботиться о том, чтобы внутренние детали `ConstraintSolver` не были доступны этому механизму.

Паттерн **хранитель** поможет решить данную проблему. **Хранитель** – это объект, в котором сохраняется внутреннее состояния другого объекта – **хозяина** хранителя. Для работы механизма отката нужно, чтобы хозяин предоставил хранитель, когда возникнет необходимость записать контрольную точку состояния хозяина. Только хозяину разрешено помещать в хранитель информацию и извлекать ее оттуда, для других объектов хранитель непрозрачен.

В примере графического редактора, который обсуждался выше, в роли хозяина может выступать объект `ConstraintSolver`. Процесс отката характеризуется такой последовательностью событий:

1. Редактор запрашивает хранитель у объекта `ConstraintSolver` в процессе выполнения операции перемещения.
2. `ConstraintSolver` создает и возвращает хранитель, в данном случае экземпляр класса `SolverState`. Хранитель `SolverState` содержит структуры

данных, описывающие текущее состояние внутренних уравнений и переменных `ConstraintSolver`.

3. Позже, когда пользователь отменяет операцию перемещения, редактор возвращает `SolverState` объекту `ConstraintSolver`.
4. Основываясь на информации, которая хранится в объекте `SolverState`, `ConstraintSolver` изменяет свои внутренние структуры, возвращая уравнения и переменные в первоначальное состояние.

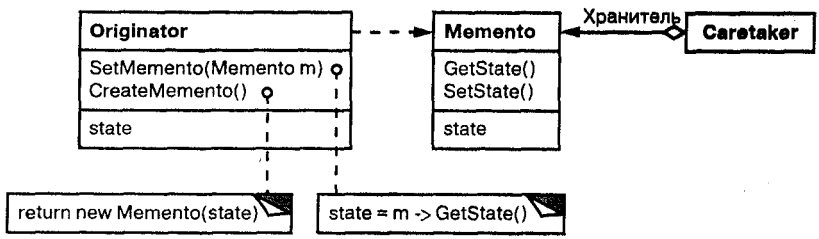
Такая организация позволяет объекту `ConstraintSolver` «знакомить» другие объекты с информацией, которая ему необходима для возврата в предыдущее состояние, не раскрывая в то же время свою структуру и представление.

Применимость

Используйте паттерн хранитель, когда:

- необходимо сохранить мгновенный снимок состояния объекта (или его части), чтобы впоследствии объект можно было восстановить в том же состоянии;
- прямое получение этого состояния раскрывает детали реализации и нарушает инкапсуляцию объекта.

Структура



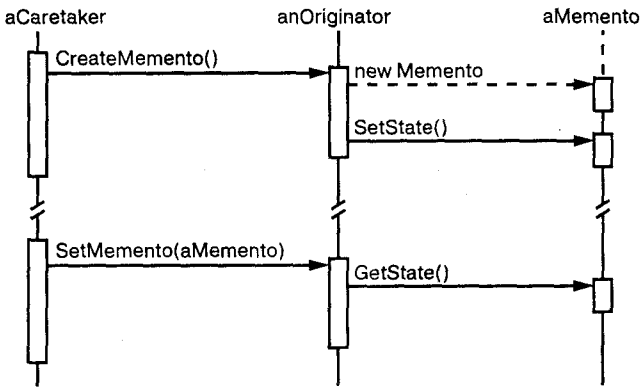
Участники

- **Memento** (`SolverState`) – хранитель:
  - сохраняет внутреннее состояние объекта `Originator`. Объем сохраняемой информации может быть различным и определяется потребностями хозяина;
  - запрещает доступ всем другим объектам, кроме хозяина. По существу, у хранителей есть два интерфейса. «Посыльный» `Caretaker` «видит» лишь *«узкий»* интерфейс хранителя – он может только передавать хранителя другим объектам. Напротив, хозяину доступен «широкий» интерфейс, который обеспечивает доступ ко всем данным, необходимым для восстановления в прежнем состоянии. Идеальный вариант – когда только хозяину, создавшему хранитель, открыт доступ к внутреннему состоянию последнего;
- **Originator** (`ConstraintSolver`) – хозяин:
  - создает хранитель, содержащего снимок текущего внутреннего состояния;
  - использует хранитель для восстановления внутреннего состояния;

- **Caretaker** (механизм отката) – посыльный:
  - отвечает за сохранение хранителя;
  - не производит никаких операций над хранителем и не исследует его внутреннее содержимое.

Отношения

- посыльный запрашивает хранитель у хозяина, некоторое время держит его у себя, а затем возвращает хозяину, как видно на представленной диаграмме взаимодействий.



Иногда этого не происходит, так как последнему не нужно восстанавливать прежнее состояние;

- хранители пассивны. Только хозяин, создавший хранитель, имеет доступ к информации о состоянии.

Результаты

Характерные особенности паттерна хранитель:

- *сохранение границ инкапсуляции.* Хранитель позволяет избежать раскрытия информации, которой должен распоряжаться только хозяин, но которую тем не менее необходимо хранить вне последнего. Этот паттерн экранирует объекты от потенциально сложного внутреннего устройства хозяина, не изменяя границы инкапсуляции;
- *упрощение структуры хозяина.* При других вариантах дизайна, направленного на сохранение границ инкапсуляции, хозяин хранит внутри себя версии внутреннего состояния, которое запрашивали клиенты. Таким образом, вся ответственность за управление памятью лежит на хозяине. При перекладывании заботы о запрошенном состоянии на клиентов упрощается структура хозяина, а клиентам дается возможность не информировать хозяина о том, что они закончили работу;
- *значительные издержки при использовании хранителей.* С хранителями могут быть связаны заметные издержки, если хозяин должен копировать большой

объем информации для занесения в память хранителя или если клиенты создают и возвращают хранителей достаточно часто. Если плата за инкапсуляцию и восстановление состояния хозяина велика, то этот паттерн не всегда подходит (см. также обсуждение инкрементности в разделе «Реализация»);

- *определение «узкого» и «широкого» интерфейсов*. В некоторых языках сложно гарантировать, что только хозяин имеет доступ к состоянию хранителя;
- *скрытая плата за содержание хранителя*. Посыльный отвечает за удаление хранителя, однако не располагает информацией о том, какой объем информации о состоянии скрыт в нем. Поэтому нетребовательный к ресурсам посыльный может расходовать очень много памяти при работе с хранителем.

Реализация

При реализации паттерна хранитель следует иметь в виду:

- *языковую поддержку*. У хранителей есть два интерфейса: «широкий» для хозяев и «узкий» для всех остальных объектов. В идеале язык реализации должен поддерживать два уровня статического контроля доступа. В C++ это возможно, если объявить хозяина другом хранителя и сделать закрытым «широкий» интерфейс последнего (с помощью ключевого слова `private`). Открытым (`public`) остается только «узкий» интерфейс. Например:

```
class State;

class Originator {
public:
    Memento* CreateMemento();
    void SetMemento(const Memento*);
    // ...
private:
    State* _state; // внутренние структуры данных
    // ...
};

class Memento {
public:
    // узкий открытый интерфейс
    virtual ~Memento();
private:
    // закрытые члены доступны только хозяину Originator
    friend class Originator;
    Memento();

    void SetState(State*);
    State* GetState();
    // ...
private:
    State* _state;
    // ...
};
```

- *сохранение инкрементных изменений.* Если хранители создаются и возвращаются своему хозяину в предсказуемой последовательности, то хранитель может сохранить лишь *изменения* во внутреннем состоянии хозяина. Например, допускающие отмену команды в списке истории могут пользоваться хранителями для восстановления первоначального состояния (см. описание паттерна команда). Список истории предназначен только для отмены и повтора команд. Это означает, что хранители могут работать лишь с изменениями, сделанными командой, а не с полным состоянием объекта. В примере из раздела «Мотивация» объект, отменяющий ограничения, может содержать только такие внутренние структуры, которые изменяются с целью сохранить линию, соединяющую прямоугольники, а не абсолютные позиции всех объектов.

Пример кода

Приведенный пример кода на языке C++ иллюстрирует рассмотренный выше пример класса `ConstraintSolver` для разрешения ограничений. Мы используем объекты `MoveCommand` (см. паттерн команда) для выполнения и отмены переноса графического объекта из одного места в другое. Графический редактор вызывает операцию `Execute` объекта-команды, чтобы переместить объект, и команду `Unexecute`, чтобы отменить перемещение. В команде хранятся координаты места назначения, величина перемещения и экземпляр класса `ConstraintSolverMemento` – хранителя, содержащего состояние объекта `ConstraintSolver`:

```
class Graphic;
    // базовый класс графических объектов

class MoveCommand {
public:
    MoveCommand(Graphic* target, const Point& delta);
    void Execute();
    void Unexecute();
private:
    ConstraintSolverMemento* _state;
    Point _delta;
    Graphic* _target;
};
```

Ограничения связанности устанавливаются классом `ConstraintSolver`. Его основная функция-член, называемая `Solve`, отменяет ограничения, регистрируемые операцией `AddConstraint`. Для поддержки отмены действий состояние объекта `ConstraintSolver` можно разместить в экземпляре класса `ConstraintSolverMemento` с помощью операции `CreateMemento`. В предыдущее состояние объект `ConstraintSolver` возвращается посредством операции `SetMemento`. `ConstraintSolver` является примером паттерна одиночка:

```
class ConstraintSolver {
public:
    static ConstraintSolver* Instance();

    void Solve();
```

```
    void AddConstraint(
        Graphic* startConnection, Graphic* endConnection
    );
    void RemoveConstraint(
        Graphic* startConnection, Graphic* endConnection
    );

    ConstraintSolverMemento* CreateMemento();
    void SetMemento(ConstraintSolverMemento*);
private:
    // нетривиальное состояние и операции
    // для поддержки семантики связанности
};

class ConstraintSolverMemento {
public:
    virtual ~ConstraintSolverMemento();
private:
    friend class ConstraintSolver;
    ConstraintSolverMemento();

    // закрытое состояние Solver
};
```

С такими интерфейсами мы можем реализовать функции-члены `Execute` и `Unexecute` в классе `MoveCommand` следующим образом:

```
void MoveCommand::Execute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _state = solver->CreateMemento(); // создание хранителя
    _target->Move(_delta);
    solver->Solve();
}

void MoveCommand::Unexecute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _target->Move(-_delta);
    solver->SetMemento(_state); // восстановление состояния хозяина
    solver->Solve();
}
```

`Execute` запрашивает хранитель `ConstraintSolverMemento` перед началом перемещения графического объекта. `Unexecute` возвращает объект на прежнее место, восстанавливает состояние `Solver` и обращается к последнему с целью отменить ограничения.

Известные применения

Предыдущий пример основан на поддержке связанности в каркасе `Unidraw` с помощью класса `CSolver` [VL90].

В коллекциях языка Dylan [App92] для итерации предусмотрен интерфейс, напоминающий паттерн хранитель. Для этих коллекций существует понятие состояния объекта, которое является хранителем, представляющим состояние

итерации. Представление текущего состояния каждой коллекции может быть любым, но оно полностью скрыто от клиентов. Решение, используемое в языке Dylan, можно написать на C++ следующим образом:

```
template <class Item>
class Collection {
public:
    Collection();

    IterationState* CreateInitialState();
    void Next(IterationState*);
    bool IsDone(const IterationState*) const;
    Item CurrentItem(const IterationState*) const;
    IterationState* Copy(const IterationState*) const;

    void Append(const Item&);
    void Remove(const Item&);
    // ...
};
```

Операция `CreateInitialState` возвращает инициализированный объект `IterationState` для коллекции. Операция `Next` переходит к следующему объекту в порядке итерации, по сути дела, она увеличивает на единицу индекс итерации. Операция `IsDone` возвращает `true`, если в результате выполнения `Next` мы оказались за последним элементом коллекции. Операция `CurrentItem` разыменовывает объект состояния и возвращает тот элемент коллекции, на который он ссылается. `Copy` возвращает копию данного объекта состояния. Это имеет смысл, когда необходимо оставить закладку в некотором месте, пройденном во время итерации.

Если есть класс `ItemType`, то обойти коллекцию, составленную из его экземпляров, можно так:[1]

```
class ItemType {
public:
    void Process();
    // ...
};

Collection<ItemType*> aCollection;
IterationState* state;

state = aCollection.CreateInitialState();

while (!aCollection.IsDone(state)) {
    aCollection.CurrentItem(state)->Process();
    aCollection.Next(state);
}
delete state;
```

[1] Отметим, что в нашем примере объект состояния удаляется по завершении итерации. Но оператор `delete` не будет вызван, если `ProcessItem` возбудит исключение, поэтому в памяти остается мусор. Это проблема в языке C++, но не в Dylan, где есть сборщик мусора. Решение проблемы обсуждается на стр. 258.

У интерфейса итерации, основанного на паттерне хранитель, есть два преимущества:

- с одной коллекцией может быть связано несколько активных состояний (то же самое верно и для паттерна итератор);
- не требуется нарушать инкапсуляцию коллекции для поддержки итерации. Хранитель интерпретируется только самой коллекцией, больше никто к нему доступа не имеет. При других подходах приходится нарушать инкапсуляцию, объявляя классы итераторов друзьями классов коллекций (см. описание паттерна итератор). В случае с хранителем ситуация противоположная: класс коллекции `Collection` является другом класса `IteratorState`.

В библиотеке QOCA для разрешения ограничений в хранителях содержится информация об изменениях. Клиент может получить хранитель, характеризующий текущее решение системы ограничений. В хранителе находятся только те переменные ограничений, которые были преобразованы со времени последнего решения. Обычно при каждом новом решении изменяется лишь небольшое подмножество переменных `Solver`. Но этого достаточно, чтобы вернуть `Solver` к предыдущему решению; для отката к более ранним решениям необходимо иметь все промежуточные хранители. Поэтому передавать хранители в произвольном порядке нельзя. QOCA использует механизм ведения истории для возврата к прежним решениям.

Родственные паттерны

Команда: команды помещают информацию о состоянии, необходимую для отмены выполненных действий, в хранители.

Итератор: хранители можно использовать для итераций, как было показано выше.

Паттерн Observer

Название и классификация паттерна

Наблюдатель – паттерн поведения объектов.

Назначение

Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

Известен также под именем

Dependents (подчиненные), Publish-Subscribe (издатель-подписчик).

Мотивация

В результате разбиения системы на множество совместно работающих классов появляется необходимость поддерживать согласованное состояние взаимосвязанных