Специализированные объектные модели

Денис С. Мигинский

Задача

Требуется разработать настраиваемую систему документооборота (упрощенный аналог Lotus Domino/SAP/1C).

Требования:

- Возможность задавать произвольный набор видов документов
- Возможность задавать жизненный цикл документов (бизнес-процессы)
- Структура документов и бизнес-процессы могут меняться независимо
- Для использования вышеперечисленных механизмов программист не должен знать деталей реализации системы
- Система обслуживается одним сервером
- Многопользовательский Web-доступ для просмотра и редактирования

Подзадача

Требуется разработать представление документа.

Требования:

- Обеспечение ACID (включая объектную персистентность)
- Авторизация доступа
- Визуальное представление документа (визуализация + редактирование)
- Возможность задавать произвольный набор видов документов; при этом разработчик этих видов документов не должен заботиться о ACID, а также о разграничении доступа и визуализации, если его устраивают умолчательные механизмы
- Расширяемость набора операций над документами

ООП-решение «в лоб»

Берем наш любимый ОО-язык, определяем базовый класс Document со следующими операциями:

- save (abstract/template)
- load (abstract/template)
- isActionAuthorized
- startTransaction (abstract/template)
- endTransaction (abstract/template)
- display (abstract/template)

Каждый подкласс должен определить/доопределить большинство операций, каждая бизнес-операция должна заботиться о транзакциях, авторизации, загрузке объекта.

Анализ решения

Недостатки:

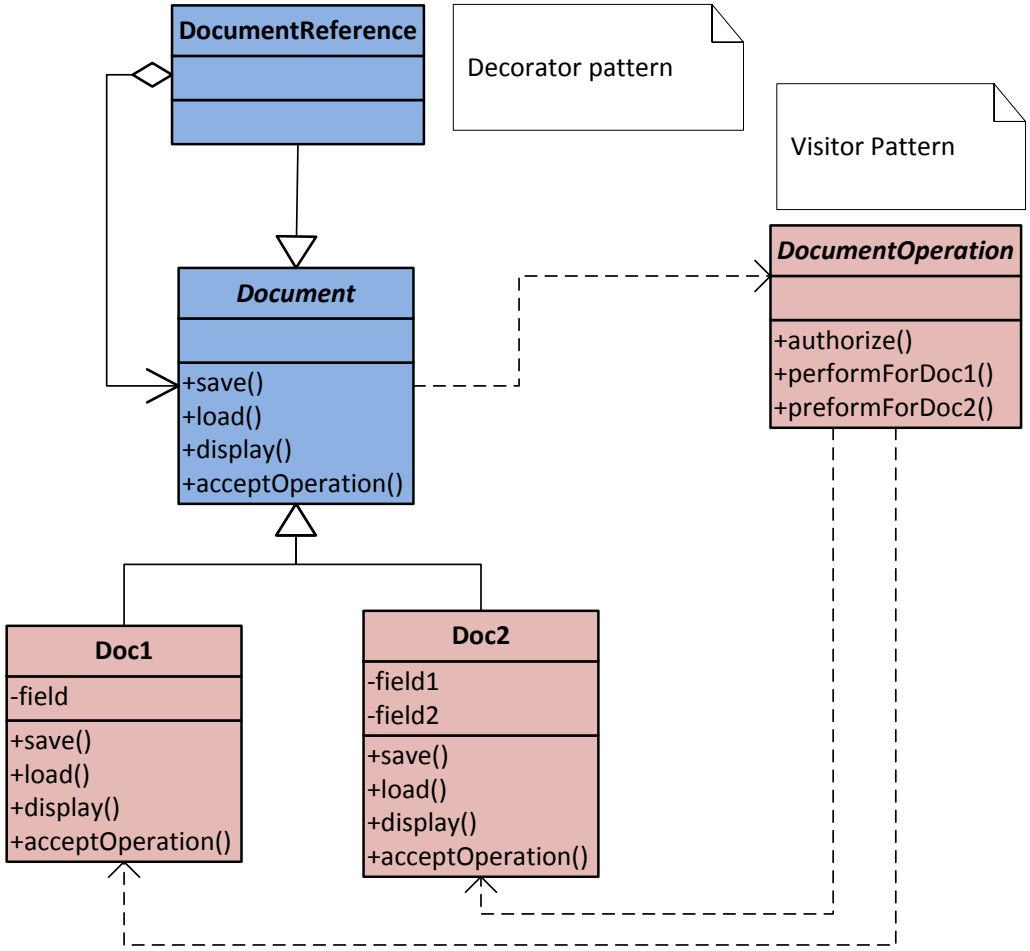
- Подклассы должны имплементировать системные операции (save, load)
- Ответственность за границы транзакций, загрузку объектов из базы возлагается на подклассы
- Непонятно, каким образом транзакция будет отслеживать изменение ссылок
- Нет расширяемости набора операций

Вывод:

Решение не удовлетворяет требованиям (слишком много ответственности на прикладной бизнес-логике), нарушает SoC

ООП-решение с применением шаблонов проектирования

- Доступ к документам через транзакционно-управляемые ссылки
- Операции задаются вне классов документов
- Вызов операции автоматически авторизуется и «оборачивается» транзакцией



Анализ решения

Преимущества:

- Бизнес-логика ничего не знает про транзакции и авторизацию
- Разделены операции и структура документа

Недостатки:

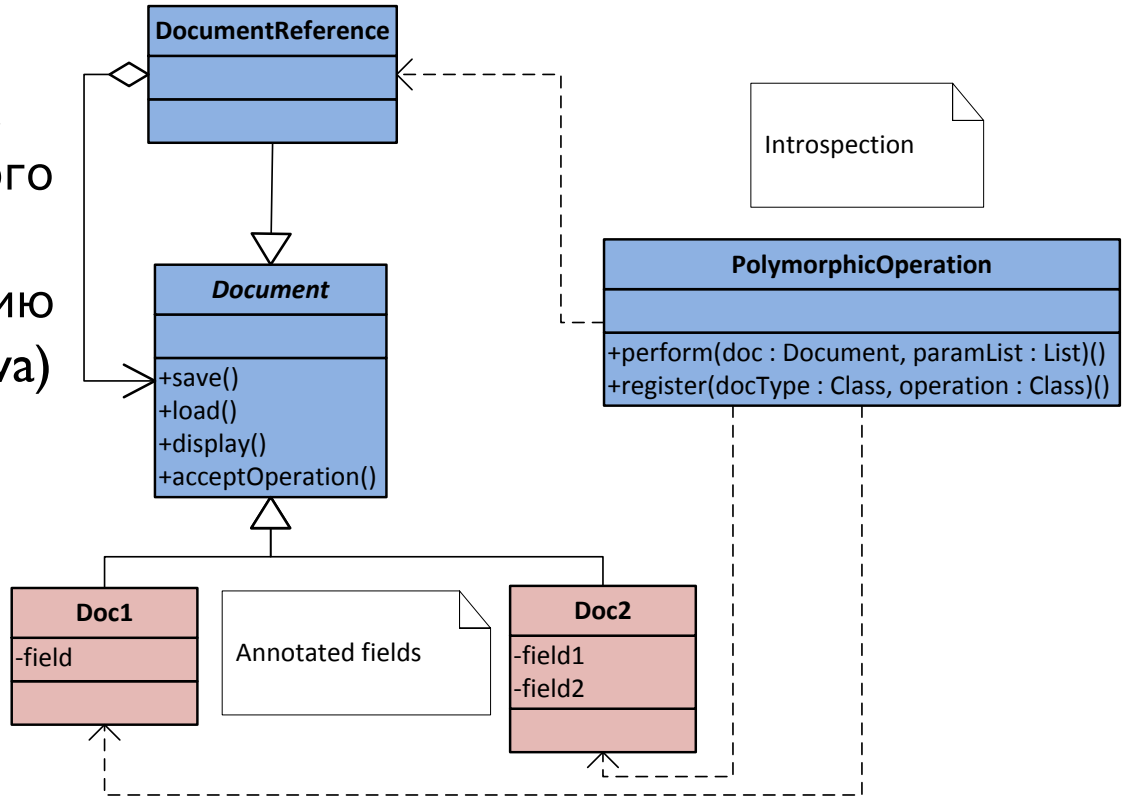
- Подклассы по-прежнему должны имплементировать системные операции (save, load)
- Применение Visitor усложняет модификацию иерархии документов: базовый класс операции должен знать о всей иерархии (либо отказываемся от полиморфизма)
- Усложнение пользовательского кода:
  было: `doc.op1()`
  стало: `doc.acceptOperation(new Op1())`

Вывод:

Решение лучше предыдущего, но много проблем осталось

Решение за рамками ООП

- Реализация save, load, display (умолчательный вариант) генерируются автоматически на основе аннотаций полей. В Java: переопределение ClassLoader + инструментирование байт-кода, либо АОП-механизм
- PolymorphicOperation. register – для заданного класса документа регистрирует операцию (подкласс Callable в Java)



PolymorphicOperation.perform

Сценарий:

1. Определяем точный класс документа
2. Ищем по классу документа зарегистрированный класс операции
3. Если не получилось, то повторяем для суперкласса
4. Ищем у класса операции (подкласс Callable для Java) конструктор с соответствующим числом параметров
5. Инстанцируем операцию (как объект)
6. Вызываем

Замечание: при наличии в языке лямбда-выражений регистрируем их вместо класса, пропускаем пп. 4-5

Анализ решения

Преимущества:

- Бизнес-логика ничего не знает про транзакции и авторизацию
- Разделены операции и структура документа, на этот раз полноценно
- Классы документов больше не должны имплементировать системные операции

Недостатки:

- Потенциально сложная реализация (зависит от языка и доп. средств)
- Необходимость аннотации полей
- Отсутствие декларативного синтаксиса для описания операций

Вывод:

Технически, требования выполнены. Есть ли другие эффективные методы решения?

План реализации (платформа)

- Язык – Clojure
- Механизм управления транзакциями – на первом этапе встроенный, возможно, с расширениями
- Объектная модель – строим свою

Первая итерация: Proof of Concept

Реализуем:

- Представление типов документов
- Скалярные атрибуты
- Одиночное наследование
- ACI (без «D» и, частично, без «C»)
- Поведение (должно определяться вне типа документа)
- Полиморфизм по одному параметру

Не реализуем:

- «D» в ACID
- Валидация («C»)
- Авторизация
- Ссылки на другие документы

Дополнительно:

На первой итерации не заботимся о производительности, оптимизировать будем потом

Оговорки

В рассматриваемом решении также отсутствует:

- Проверка ошибок
- Покрытие тестами

Внимание!!! Это сделано исключительно для компактности изложения.

В реальной системе такое допустимо только если мы делаем прототип, код которого заведомо не пойдет в релиз.

Пример целевого кода

```
;Определение типов документов
(def-doc-type :BaseDoc
  (:fields (:cnt1)))
(def-doc-type :DerivedDoc
  (:fields (:cnt2))
  (:super :BaseDoc))

;Определение «абстрактного метода»
(def-command inc-counters)

;Определение реализаций методов
(def-method inc-counters :BaseDoc [obj amount]
  (setf! obj :cnt1 (+ amount (getf obj :cnt1))))
(def-method inc-counters :DerivedDoc [obj amount]
  (super amount)
  (setf! obj :cnt2 (+ amount (getf obj :cnt2))))

;Инстанцирование документов, вызов «методов»
(let [doc (create-doc :DerivedDoc :cnt1 1 :cnt2 2 :cnt3 3)]
  (println (getf doc :cnt1) (getf doc :cnt2))
  (inc-counters doc 2)
  (println (getf doc :cnt1) (getf doc :cnt2)))
```

API: метаобъекты

```
;Определение типа документов
;sections: (:fields (:f1) (:f2)) (:super :Base)
(defmacro def-doc-type [name & sections]

;Элементы интроспекции
(defn super-type [type]

(defn has-field? [type field]

;Поведение
(defn def-command [name]

(defn def-query [name]

(defn def-method [command-name type argv & body]
```

Поведение: сообщения против обобщенных функций

Посылка сообщения

```
obj.message (params)
```

Обобщенная функция (generic function)

```
func(obj, params)
```

Найдите 10 не-синтаксических отличий (или хотя бы одно)

Обобщенные функции и полиморфизм

При посылке сообщения происходит диспетчеризация по типу объекта (т.е. по одному параметру)

При вызове обобщенной функции происходит диспетчеризация по типам выделенных управляющих параметров.

Варианты:

- 0 управляющих параметров – обычная функция
- 1 управляющих параметр – полный аналог посылке сообщений: диспетчеризация по одному параметру, реализации обобщенной функции называются методами
- N управляющих параметров – диспетчеризация по нескольким параметрам, реализации обобщенной функции называются мультиметодами

Command-query separation (CQS)

Принцип построение интерфейса класса (Б. Мейер), декларирующий, что поведение должно быть разделено на группы функций/методов:

- Команды – модифицируют состояние объекта, результат не возвращают.
- Запросы – не модифицируют объект, возвращают результат.

Дополнительные параметры не модифицируются никогда (изоляция побочных эффектов)

CQS в данной задаче

ACID:

Command – транзакция

Query – не транзакция (если используем MVCC)

Авторизация:

Command – требуются права на запись документа

Query – требуются права на чтение

API: объекты

```
;инстанцирование
;fields: :f1 1 :f2 2 …
(defn create-doc [type & fields]

;доступ к полям (атрибутам)
(defn getf [obj field]

(defn setf! [obj field val]

;интроспекция
(defn doc-type [doc]

;вызов команды/запроса
;(полноценная функция)
(command-or-query doc & args)
```

Реализация: иерархия классов

```
;здесь будет храниться иерархия
(def doc-hierarchy (ref {}))

;аналог Object, не может быть определен штатно
(dosync
  (alter doc-hierarchy
         (fn [h] (assoc h :Document
                        {::type :Document,
                         ::super nil
                         ::fields {}
                         ::doc-refs {}}))))
```

Определение типа

```
(defmacro def-doc-type [name & sections]
  `(let [s-map# (преобразуем sections в ассоциативный массив)
         super# (or (first (s-map# :super)) :Document)
         fields# (преобразуем список полей в а. массив)]
     (dosync
       (alter
         doc-hierarchy
         (fn [h#]
           (assoc
             h# ~name
             {::type ~name
              ::super super#
              ::fields fields#}))))))
```

Создание экземпляра

```
(defn create-doc [type & fields]
  (let [type-desc (@doc-hierarchy type)
        f-map (apply hash-map fields)
        state (ref {})]
    (dosync
      (doseq [kv f-map]
        (when (has-field? type (first kv))
          (alter state
                 #(assoc % (first kv) (second kv))))))
    {::type type
     ;;все состояние хранится под одним ref
     ;;можем использовать стандартный механизм валидации
     ::fields state}))
```

Команда

```
(defmacro def-command [name]
  ;;функция с состоянием
  `(let [vtable# (ref {})]
     (defn ~name [obj# & args#]
       (if (document? obj#)
         ;;стандартный вызов
         (apply perform-effective-command
                (concat (list @vtable# (doc-type obj#) obj#)
                        args#))
         ;;специальный вызов: регистрация метода
         (dosync
           (alter vtable#
                  #(assoc % (first obj#) (second obj#)))))))
```

Механизм вызова

```
;;«заглушка» для super
(def ^:dynamic super nil)

;;собственно, механизм вызова
(defn perform-effective-command [vtable eff-type obj & args]
  ;;dispatch ищет наиболее специфичный метод
  ;;возвращает пару [тип метод]
  (let [[d-type eff-fn] (dispatch vtable eff-type)
        d-super-type (super-type d-type)]
    ;;определение динамического лексического контекста
    (binding [super (partial
                      perform-effective-command
                      vtable d-super-type obj)]
      (dosync
        (apply eff-fn (cons obj args))))))

;;определение метода
(defmacro def-method [command-name type argv & body]
  `(~command-name [~type (fn ~argv ~@body)]))
```

Динамический лексический контекст

Динамический лексический контекст в отличие от статического (let) влияет не на АСД, а на стек вызовов.

Используется, например, для определения thread-local переменных.

```
(def ^:dynamic variable 1)

(defn my-f []
  (println variable))

(let [variable 5]
  (my-f))

(binding [variable 5]
  (my-f))

>>
1
5
```

Пример целевого кода: повтор

```
(def-doc-type :BaseDoc
  (:fields (:cnt1)))
(def-doc-type :DerivedDoc
  (:fields (:cnt2))
  (:super :BaseDoc))

(def-command inc-counters)

(def-method inc-counters :BaseDoc [obj amount]
  (setf! obj :cnt1 (+ amount (getf obj :cnt1))))
(def-method inc-counters :DerivedDoc [obj amount]
  (super amount)
  (setf! obj :cnt2 (+ amount (getf obj :cnt2))))

(let [doc (create-doc :DerivedDoc :cnt1 1 :cnt2 2 :cnt3 3)]
  (println (getf doc :cnt1) (getf doc :cnt2))
  (inc-counters doc 2)
  (println (getf doc :cnt1) (getf doc :cnt2)))
```

Реализация остальных механизмов

Валидация: расширяем определение **def-doc-type** и **create-doc**, валидатор помещаем в **ref** с состоянием.

Ссылки на документы: по аналогии с **fields**. Возможно, следует ввести параметры **:min-occurs**, **:max-occurs**.

Объектная персистентность (в первом приближении): вводим идентификаторы документов, сохранение документ в виде файла на диск, ссылки на объекты заменяем на идентификаторы.

Реализация остальных механизмов: продолжение

«D» в ACID: модифицируем STM (либо реализуем свой) таким образом, чтобы во время commit состояние сохранялось.

Объектная персистентность (во втором приближении): модифицируем ref, чтобы документ загружался при первом обращении. Более аккуратно пишем на диск (чтобы не было состояния, когда файл начали переписывать , но не успели закончить), либо используем СУБД (XML-oriented или document-oriented)

Реализация остальных механизмов: продолжение

Объектная персистентность (в третьем приближении): используем структурированную базу (например, реляционную), схему строим на основе деклараций полей. При необходимости расширяем декларации, например вводим различие между ассоциацией и композицией для ссылок на документы, типизацию.

Авторизация: расширяем механизм вызова обращением к некоторой внешней системе безопасности, которая принимает решение на основе объекта (его типа в простейшем случае) и текущего пользователя. Текущий пользователь может задаваться внешним контейнером через механизм динамических контекстов.

Реализация остальных механизмов: продолжение

Визуализация:

- Разрабатываем некоторый (декларативный) формат, позволяющий отобразить поля и вложенные документы на элементы форм, редактируемые или нет в зависимости от прав доступа.
- Определяем запрос **visualize**, который должен генерировать эту форму, строим реализацию по умолчанию, которая работает на основе интроспекции (концепция **naked objects**).
- Допускается переопределение этой реализации: кастомизация представления, привязка к командам и запросам.

Naked Objects

Интерфейс пользователя должен полностью автоматически генерироваться из определения бизнес-логики приложения

Дальнейшее развитие

- Создаем механизм для описания бизнес-процессов (жизненного цикла документов)
- Определяем пользовательский язык с той же семантикой, но более привычным для пользователя синтаксисом. Разрабатываем транслятор из этого языка в разработанное представление.
- …

Классификация решения

- Это ООП?
- Это АОП?
- Это новый язык?
- Это метамодель?

Анализ решения

- Созданная метамодель является практически полноценной объектной моделью с некоторой специализаций (ACID, авторизация)
- Определение поведения отличается от «общепринятого»
- Класс не является единицей модульности (что часто классифицируется как АОП)
- В рамках имеющегося синтаксиса определен, фактически, новый специализированный язык (domain-specific language, DSL) для представления документов
- Использовался метод проектирования domain-driven design (DDD): отталкиваемся в первую очередь от предметной области, а не требований конкретной задачи
- Все то же самое можно сделать в Ruby, Java и т.д. (за исключением, возможно, приспособления синтаксиса языка реализации), сложность при этом может существенно варьироваться