

Аспектно-ориентированное программирование

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin

NOTICE: Published in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

Мы обнаружили множество программных проблем, для которых ни процедурной, ни объектно-ориентированной техник не достаточно для того, что бы ясно и чётко фиксировать некоторые важные характеристики программных систем. Это ведёт к реализации архитектурных решений, рассыпающихся по всей программе, приводящих к чрезвычайно запутанному, сложному в разработке и поддержке коду. Мы представляем анализ таких характеристик и называем свойства, адресованные для этих решений. Мы покажем что основная причина трудностей лежит в пересечении функциональности, которая делает невозможным (с использованием существующих парадигм) чётко инкапсулировать функциональность. В данной статье мы представим основу для новой техники программирования, называемой аспектно-ориентированным программированием, которая делает возможным чёткое выражение структуры программ включающих в себя такую "сквозную функциональность". Мы покажем, что с помощью применения "аспектов" реализация имеет может приобрести необходимую изоляцию, логичную композицию, при этом не исключая повторное использование кода аспектов. Решение базируется на системе, которую мы построили, используя технику аспектно-ориентированного программирования.

Мы обнаружили множество программных проблем, для которых ни процедурной, ни объектно-ориентированной техник не достаточно для того, что бы ясно и чётко фиксировать некоторые важные характеристики программных систем. Это ведёт к реализации архитектурных решений, рассыпающихся по всей программе, приводящих к чрезвычайно запутанному, сложному в разработке и поддержке коду. Мы

представляем анализ таких характеристик и называем свойства, адресованные для этих решений. Мы покажем что основная причина трудностей лежит в пересечении функциональности, которая делает невозможным (с использованием существующих парадигм) чётко инкапсулировать функциональность. В данной статье мы представим основу для новой техники программирования, называемой аспектно-ориентированным программированием, которая делает возможным чёткое выражение структуры программ включающих в себя такую "сквозную функциональность". Мы покажем, что с помощью применения "аспектов" реализация имеет может приобрести необходимую изоляцию, логичную композицию при этом не исключая повторное использование кода аспектов. Решение базируется на системе, которую мы построили, используя технику аспектно-ориентированного программирования.

1. Введение

Объектно-ориентированное программирование (ООП) было представлено как технология способствующая процессу проектирования ПО, т.к. объектная модель лучше вписывается в реальную предметную область и может быть представлена наглядно. Но мы обнаружили множество проблем в реализации систем, где ООП не достаточно для чёткого фиксирования важных архитектурных решений программы. Нам представляется, что существуют некоторые проблемы, для которых не подходят ни объектно-ориентированные (ООП) ни процедурно-ориентированные языки программирования (ПОП).

Этот документ представляет собой отчёт о нашей работе, по исследованию программной техники, выделяющей чёткое выражение приемов, которые оказались затруднительными для ООП и ПОП.

Мы проанализируем сложности возникающие при реализации некоторых архитектурных решений, покажем решение этих проблем с помощью аспектов и причину, почему сквозную функциональность так трудно зафиксировать с использованием существующих техник. Мы представим базис для новой программной техники, называемой аспектно-ориентированным программированием (АОП), которая позволяет чётко выразить код вовлечённый в аспект, включая соответствующую изоляцию, композицию, и повторное использование. Мы думаем, что текущее состояние исследований в сфере АОП сопоставимо с тем, что было с ООП 20 лет

1. Основные концепции были взяты из [1, 4, 13, 28]. Кроме того, даже несмотря на то, что АОП и является новой идеей, существуют другие системы, имеющие похожие свойства. Вклад нашего документа — это анализ проблем решаемых АОП, а также начальное множество терминов и концепций сопровождающих законченный системный дизайн АОП.

Документ представляет АОП на примере — большинство обобщений и определений выводятся на основе примера. Раздел 1 описывает важные допущения о связях между языками программирования и процессом проектирования ПО. Раздел 2 вводит понятие "аспекта" функциональности. Раздел 3 использует небольшой (medium-scale) пример для представления проблемы спутывания (**tangling**), решаемой АОП. В конце раздела приведено определение термина "аспект". Раздел 4 представляет несколько более мелких примеров аспектов. Раздел 5 и 6 представляют пример законченной АОП системы. Оставшиеся разделы представляют планы на будущее, связанные работы и заключение.

2. Допущения

Процесс проектирования ПО и язык программирования взаимно поддерживают и дополняют друг друга. Процесс проектирования разбивает систему на меньшие части. Язык программирования предоставляет механизм, который позволяет программисту определить абстракции подсистем и потом составлять эти абстракции различными способами для создания общей системы. Процесс проектирования хорошо дополняет язык программирования если язык программирования представляет механизмы абстракции и композиции, чётко поддерживающие типы единиц, на которые процесс проектирования в свою очередь разбивает систему. Исходя из этой перспективы, многие из существующих языков программирования, включая ОО языки, процедурные языки и функциональные языки могут быть рассмотрены как имеющие общий корень в их ключевой абстракции, так как все они базируются на общих абстракционных и композиционных механизмах и процедурах. Для рассмотрения предмета обсуждения, мы будем ссылаться на них (ООП, ПОП, ФОП языки) как на языки обобщённых процедур (ОП). (Это совсем не значит, что мы собираемся игнорировать множество важных достоинств ООП языков, это только предположение, сделанное для того, что бы проще фокусироваться на общих свойствах ОП языков).

Метод проектирования, использованный для работы с ОП языками, ведёт к разделению системы на более мелкие подсистемы поведения или функции. Этот стиль был назван функциональной декомпозицией [\[25, 27\]](#)². Точное происхождение декомпозиции, безусловно, различается среди парадигм языков, но каждая структурная единица заключается в процедуру/функцию/объект, и в каждом случае можно сказать о том, что заключено, как о функциональном элементе общей системы. Это последнее утверждение может быть истолковано как нечто излишнее. Но это очень важно, так как сейчас мы уделим этому особое внимание, потому что по ходу этого документа мы будем выделять элементы систем не являющиеся функциональными.

3. Что такое Аспекты

Для лучшего понимания происхождения проблемы спутывания и решения предлагаемого АОП, этот раздел построен на основе разработанного нами примера [приложения \[18, 22\]](#). Существует три реализации данного приложения: простое для понимания, но не эффективное, эффективное, но трудное для понимания, и АОП приложение которое и простое для понимания и эффективное. Далее мы приведём упрощённые версии реализаций этого приложения.

Рассмотрим реализацию системы обработки чёрно-белых изображений, в которой одно или несколько изображений пропускается через серию фильтров для получения результата. В ходе разработки поставлены следующие условия:

- Система должна быть проста в разработке и поддержке. Нам необходимо быстро разработать исправную систему, и к тому же иметь возможности разработки расширений к ней.
- Система должна эффективно использовать память. Обрабатываемые изображения имеют большой размер, и для того, что бы система была эффективной она должна минимизировать как ссылки в памяти, так и общие требования по хранению промежуточных данных.

3.1. Основная функциональность

Достигнуть первой цели относительно легко. Хорошо знакомое процедурно-ориентированное программирование может быть использовано для того,

Аспектно-ориентированное программирование

что бы реализовать систему чётко, лаконично с хорошо упорядоченной моделью предметной области. В таком приложении фильтры могут быть определены как процедуры принимающие на вход несколько изображений и выдающие единственное результирующее изображение. Набор примитивных процедур будет реализовывать основные фильтры, а более высокоуровневые фильтры будут определены в терминах примитивных. Для примера примитивный фильтр **or**, который берёт два изображения и возвращает попиксельное логическое или по модулю два может быть реализован так³:

```
(defun or! (a b)
  (let ((result (new-image)))
    (loop for i from 1 to width do
      (loop for j from 1 to height do
        (set-pixel result i j
          (or (get-pixel a i j)
              (get-pixel b i j))))))
  result))
```

Операция, выполняемая над пикселями

Цикл по всем пикселям исходного изображения

Сохранение пикселей в изображении результат

Начиная с **or** или других примитивных фильтров, программист может определить фильтр, который выделяет только чёрные пиксели на горизонтальной грани, возвращая новое изображение содержащие только эти, граничные пиксели.

Функциональность	Реализация
Попиксельные логические операции	Используя "loop" описанный выше
Сдвиг, изображение вверх, вниз	Написаны используя "loop" примитив, немного отличается от написанного выше
Различие между двумя изображениями (remove)	(defun remove! (a b) (and! a (not! b)))
Пиксели на верхней грани региона (top-edge)	(defun top-edge! (a) (remove! a (down! a)))
Пиксели на нижней грани региона (bottom-edge)	(bottom-edge! a (up! a))
Горизонтальная грань региона (horizontal-edge)	(defun horizontal-edge! (a) (or! (top-edge! a) (bottom-edge! a)))

Таблица 1. uuhuhu

Замечание:

Только примитивные фильтры взаимодействуют непосредственно с перебором пикселей изображения. Фильтры более высокого уровня такие как, например "Пиксели на горизонтальной грани" выражены исключительно через примитивные фильтры. Мы получили легко читаемый, отлаживаемый и расширяемый код, который в общем случае удовлетворяет первому условию.

3.2. Оптимизация использования памяти

Но эта простая реализация не удовлетворяет второму условию, а именно оптимизации использования памяти. Каждая процедура в ходе своей работы обрабатывает все входные изображения и выдаёт новое. Выходное изображение создаётся часто, причём зачастую существуя только временно, перед тем как оно снова будет обработано другим циклом. В результате мы получаем неэффективное выделение памяти, которое ведёт к недостаткам кэширования и низкой производительности. Если взять более глобальную перспективу программы, то можно заметить, что промежуточные результаты являются входными данными для остальных фильтров. Это замечание приведёт к новой версии программы, которая комбинирует циклы для реализации исходной функциональности при создании как можно меньшего количества промежуточных изображений. Исправленный код для фильтра **horizontal-edge** будет выглядеть так:

```
(defun horizontal-edge! (a)
  (let ((result (new-image))
        (a-up (up! a))
        (a-down (down! a)))
    (loop for i from 1 to width do
      (loop for j from 1 to height do
        (set-pixel result i j
          (or (and (get-pixel a i j)
                  (not (get-pixel a-up i j)))
              (and (get-pixel a i j)
                  (not (get-pixel a-down i j)))))))
      result))
```

Создаются только три изображения результата

Одна структура цикла разделяется между многими фильтрами

Операции из многих подфильтров

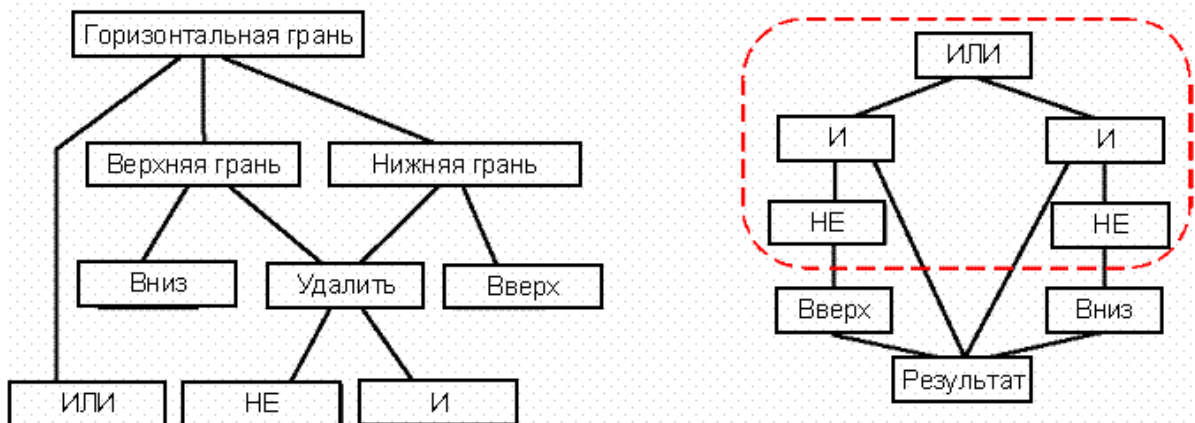
По-сравнению с оригинальным кодом, этот код "спутан". Он включает в себя все

Аспектно-ориентированное программирование

различные фильтры, которые фильтр **horizontal-edge** ранее определял в терминах примитивных фильтров, и комбинирует вместе многие, но не все, из их циклов. (Циклы для **up** и **down** не объединены, т.к. эти операции имеют различную структуру цикла)⁴. В общем, преобразованный код более эффективно использует память, но чёткая компонентная структура в нём отсутствует.

Конечно, это очень простой пример, и не трудно совладать с такой небольшой путаницей, но на практике сложности, связанные с путаницей, быстро разрастаются и становятся главной причиной проблем возникающих в ходе разработки и сопровождения больших программных систем.

Разрабатываемая нами реальная система является важной частью системы оптического распознавания символов. Её чёткая реализация, похожая на первый код представляет собой всего 768 строк кода, в то время как спутанная реализация, которая делает комбинационную оптимизацию, сохранение промежуточных результатов, динамическое выделение памяти и использует специализированные промежуточные структуры данных занимает уже 35213 строк кода. Запутанный код очень трудно сопровождать, т.к. малые изменения в функциональности требуют мысленно распутать реализацию, а потом снова запутать её.



На Рисунке 3 изображены две различные диаграммы для оптимизированной и неоптимизированной процедуры выделения горизонтальной грани (**horizontal-edge**). На левой части схемы представлена функциональная декомпозиция фильтра упорядоченная в соответствии с моделью предметной области. На правой части схемы

изображена диаграмма потоков данных, в которой квадраты это примитивные фильтры, а грани — это потоки данных между ними в ходе выполнения программы. Квадрат внизу — выходное изображение.

3.3. Пересечения (Cross-Cutting)

Возвращаясь к предыдущему примеру, рисунок 3 даёт различные основы для понимания спутывания. На левой части рисунка иерархическая структура функциональности фильтра, на правой — диаграмма потоков данных оригинальной, неоптимизированной версии фильтра "горизонтальная грань". На этой диаграмме квадраты и линии показывают примитивные фильтры и потоки данных между ними. Пунктирный овал показывает границы единого цикла для оптимизированной версии фильтра. Заметьте, что этот овал не включает весь фильтр, кроме того, это разграничение никак не связано с какой-нибудь иерархией на левой части рисунка. Реализованные два условия — функциональность и слияние циклов (то есть эффективность) — оба выражаются из одних и тех же примитивных фильтров, но составляются различно. Функциональность организуются иерархически в традиционном смысле. Смешивание циклов организуется с помощью смешивания циклов тех примитивных фильтров, которые имеют одинаковую структуру цикла как непосредственные соседи в графе диаграммы потоков данных. Каждое из таких композиционных правил по отдельности легко понять, когда смотришь на его собственную диаграмму. Но вместе, эти два отношения настолько сильно пересекают друг друга, что очень сложно определить одно отношение, имея перед глазами граф другого.

Феномен пересечения напрямую ответственен за спутывание кода. Единственный механизм композиции, который предоставлен нам языком — вызов процедуры — очень хорошо подходит для построения неоптимизированных функциональных подсистем. Но он не может нам помочь объединить функциональные подсистемы и смешивание циклов одновременно, потому что они следуют различным композиционным правилам и должны быть реорганизованы.

Это разделение вынуждает нас комбинировать свойства полностью в ручную — то, что случается в перепутанном коде выше. В общем случае, всякий раз, когда необходимо реализовать оба требования, и при этом не потерять координации свойств,

Аспектно-ориентированное программирование

мы говорим, что данные свойства пересекают друг друга. Так как ОП языки предоставляют только один механизм объединения свойств, программист должен делать декомпозицию вручную, что ведёт к сложному и запутанному коду.

Теперь мы можем определить два важных термина более точно. По отношению к системе и к её реализации используя ОП языки свойство (функциональность) должно быть реализовано как:

Компонент, если свойство может быть чётко заключено в обобщённой процедуре (то есть объект, метод, процедура, API). Под чётким мы понимаем хорошо локализованный, легко доступный и способный к сопоставлению если необходимо. Компоненты тяготеют быть подсистемами функциональной декомпозиции системы, такие как, например, фильтр, банковский счёт и компонент пользовательского интерфейса.

Аспект, если свойство не может быть чётко инкапсулировано в обобщённой процедуре. Аспекты стремятся быть подсистемами функциональной декомпозиции системы, быть свойствами, которые влияют на производительность или семантику компонентов в системном виде. Примеры аспектов включают доступ к памяти, паттерны и синхронизация параллельно действующих объектов (в Разделе 4 приведено больше примеров).

Используя эти термины можно сформулировать цели АОП: Инструментальная поддержка программиста в чётком разделении компонентов и аспектов друг от друга⁵ с помощью предоставления механизма, позволяющего абстрагировать и составлять их (компоненты и аспекты) для разработки общей системы. В противопоставление к ОП программированию, позволяющему разделять только компоненты, предоставляя механизм который делает возможным абстрагироваться и составлять их для производства общей системы⁶.

4. Другие примеры пересечения компонент

Перед тем как продолжать анализ АОП, и как эта техника решает проблемы аспектов вплетённых в код, этот раздел кратко покажет ещё несколько примеров аспектов и компонентов. Для каждого примера в таблице мы перечислим приложение, ОП язык который будет выполнять работу фиксации компонентной структуры приложения,

наиболее вероятную компонентную структуру приложения, если программист будет использовать этот тип языка программирования и аспекты, которые будут пересекать компонентную структуру.

Приложения	ОП язык	Компоненты	Аспекты
Обработка изображений	Процедурный	Фильтры	Динамическое выделение памяти
Цифровая библиотека	ООП	Репозитории, принтеры, сервисы	Минимизация сетевого трафика Синхронизационные ограничения Обработка отказов
Матричные алгоритмы	Процедурный	Операторы линейной алгебры	Матричное представление Перестановка Ошибки операций с плавающей запятой

Некоторые аспекты настолько обычны, что они могут быть легко представлены безотносительно некоторой предметной области. Один из лучших примеров — это обработка ошибок. Мы знакомы с феноменом, что добавление хорошей обработки ошибок в простую систему сводится к необходимости внесения небольших исправлений по всему коду системы. Это случается, потому что ошибки возникают в различных контекстах (динамически изменяющиеся контексты), или контекст влияет на обработку ошибки. Множество вопросов связанных с производительностью — аспекты, потому что оптимизация производительности использует информацию о контексте выполнения, который охватывает все компоненты.

5. Первый пример АОП

В этом разделе мы вернемся к примеру системы обработки изображений, и используем её для описания АОП реализации этого приложения. Представление базируется на системе, которую мы разработали, и упрощает некоторые элементы. Полная система обсуждается в [22]. Цель этого раздела — быстро получить законченную структуру АОП реализации, но не описывать эту структуру детально. Полное описание

Аспектно-ориентированное программирование

структуры будет дано в разделе 6.

Структура АОП приложения аналогична структуре ОП приложения. Принимая во внимание, что ОП реализация приложения состоит из (1) языка программирования, (2) компилятора (или интерпретатора) для этого языка, (3) программы написанной на этом языке; АОП реализация состоит из (1а) компонентного языка, с помощью которого создаются компоненты, (1б) одного или нескольких аспектных языков, с помощью которых реализуются аспекты, (2) компоновщика (weaver) аспектов для комбинации языков, (3а) компонентной программы, которая реализует компоненты, используя компонентный язык и (3б) одной или нескольких аспектных программ, которые реализуют аспекты, используя для этого язык аспектов. Как ОП язык, так АОП языки и компоновщики могут быть спроектированы так, что компоновка (вплетение аспектов) производится во время выполнения (Runtime weaving) или во время компиляции (Compile time weaving).

5.1. Компонентный язык и программа

В нашем примере мы будем использовать один компонентный язык и один язык аспектов. Компонентный язык похож на процедурный язык, который мы использовали ранее, с небольшими изменениями. Первое изменение — фильтры не являются больше процедурами. Второе изменение — примитивные циклы написаны таким образом, что их циклическая структура выражена наиболее явно. Используя компонентный язык, фильтр **or** может быть написан следующим образом:

```
(define-filter or! (a a)
  (pixelwise (a b) (aa bb) (or aa bb)))
```

Конструкция **pixelwise** это итератор, который в данном случае пробегает через изображения A и B фиксированным шагом, приравнивая **aa** и **bb** к значениям пикселей соответствующих изображений, возвращая суммарное изображение. Четыре похожие конструкции предоставляют различные варианты объединения, распределения, смещения и комбинирования значений пикселей которые необходимы системе. Ввод этих высокоуровневых циклических конструкций — критическое изменение, которое позволит языку аспектов определять и анализировать объединение циклов более просто.

5.2. Аспектный язык и программа

Дизайн аспектного языка, используемый в этом приложении основан на наблюдении, что диаграмма потоков данных в Рисунке 1 делает более простым понимание необходимых смешиваний циклов. Язык аспектов это простой процедурный язык, который предоставляет простые операции над точками в диаграмме потоков данных. Аспектная программа может непосредственно находить циклы, которые должны быть объединены и выполнять необходимые объединения. Следующий фрагмент кода это часть ядра этой аспектной программы — она выполняет слияние, описанное в разделе 5. Она проверяет две точки соединенные информационным потоком, если обе имеют по пиксельную структуру, то соединяет их в один цикл, имеющий попиксельную структуру, выполняя при этом соответствующее слияние входов, переменных цикла и тел обоих циклов.

```
(cond ((and (eq (loop-shape node) 'pointwise)
            (eq (loop-shape input) 'pointwise))
      (fuse loop input 'pointwise
            :inputs (splice ...)
            :loop-vars (splice ...)
            :body (subst ...))))
```

Описание композиционных правил и слияния структуры четырёх типов циклов в реальной системе требует около дюжины похожих условий: о том, где и когда объединение должно иметь место. Эта проблема является одной из причин, по которой эта система не может быть построена с помощью оптимизирующего компилятора для необходимого объединения циклов, т.к. для упрощения применяется анализ алгоритма программы недоступный компилятору. (Хотя многие компиляторы смогли бы оптимизировать этот конкретный пример). Другая сложность в этом примере — это аспект обработки ошибок, включающий в себя разделение промежуточных результатов и обеспечение объёма динамической памяти на фиксированном пределе.

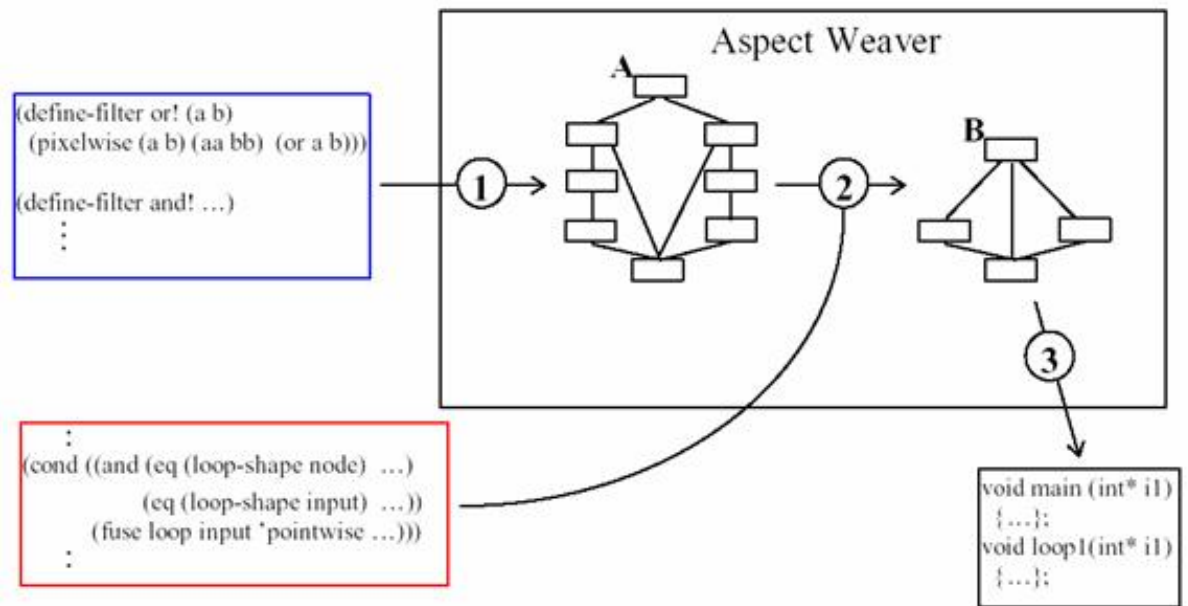
5.3. Вплетение

Аспектно-ориентированное программирование

Аспектный компоновщик принимает компоненты и аспектные программы на вход, и выводит программу на языке C на выход. Этот процесс разделяется на три различные фазы, как иллюстрировано на Рисунке 2.

```
(pointwise (#<edge1> #<edge2>) (i1 i2) (or i1 i2))
```

В ходе первой фазы, компоновщик использует технику развёртывания для генерации диаграммы потока данных из компонентной программы. На этой диаграмме, узлы представляют примитивные фильтры, а грани представляют изображение идущее от одного примитивного фильтра к другому. Каждый узел состоит из одного цикла. Для примера узел обозначенный **A** состоит из следующего цикла, где #<: > — ссылка на грань входящую в узел.



В ходе второй фазы выполняется аспектная программа, для редактирования диаграммы с помощью свёртывание узлов вместе и соответствующего преобразования тел схлопываемых циклов. Результат — граф, в котором некоторые из циклических структур имеют больше одной примитивной операции, то есть некоторые циклы объединены. Для примера узел обозначенный **B** который соответствует объединению пяти циклов из исходного графа, имеет следующую конструкцию в качестве тела цикла:

```
(pointwise (#<edge1> #<edge2> #<edge3>) (i1 i2 i3)
  (or (and (not i1) i2) (and (not i3) i2))))
```

Конечная фаза 3 — простой кодогенератор проходит по созданному графу, генерируя одну C функцию для каждого узла, и генерирует главную функцию, которая вызывает циклические функции в соответствующем порядке, передавая им соответствующие результаты из предыдущих циклов. Кодогенерация проста, т.к. каждый узел содержит только одиночную циклическую конструкцию, с телом, составленным полностью из примитивных операций над пикселями.

Критическое свойство этой системы в том, что компоновщик (weaver) не обладает "интеллектуальностью" компилятора, который достаточно трудно построить. Используя АОП, мы упорядочили стратегию принятия решений — вся "интеллектуальность" предоставляется программистом, используя языки аспектов. Работа компоновщика это скорее интеграция, чем интеллектуальная работа⁷.

5.4. Результаты

Реальная система, несомненно, намного более сложна. Для условия эффективности, создано две дополнительных аспектных программы, одна из которых управляет разделением общих вычислений, и одна из которых заботится о том, что бы минимально возможное количество изображений было в памяти в одно и тоже время. Все три аспектных программы написаны на одном и том же аспектом языке. В этом примере АОП реализация достигла исходных задач проектирования — код приложения лёгок для понимания, разработки и сопровождения, будучи так же очень эффективным.

Программисту легко понять влияние аспектных программ на весь выходной код. Изменения или в фильтре или в аспекте просто распространяются на всю систему с помощью вплетения. Что сложно для программиста, так это генерация выходного кода. Мощь АОП в том, что компоновщик заботится об этих подробностях, вместо программиста который вынужден делать это вручную.

Наша АОП реализация содержит 1039 строк кода, включая компонентную программу и все три аспектных программы. Аспектный компоновщик, включая повторно-используемый генератор кода, содержит 3520 строк кода (ядро

Аспектно-ориентированное программирование

компоновщика всего 1959 строк). Производительность АОП реализации сопоставима с 35213 строчной версией (эффективность по времени немного хуже, эффективность по памяти лучше)[8](#).

Так же как и с множеством других проектов, связанных с проектированием ПО, очень трудно измерить преимущества от использования АОП без большого количества экспериментов, с привлечением нескольких программистов использующих АОП и традиционные техники для разработки и поддержки различных приложений [\[6, 21, 36\]](#). Такие исследования находятся за пределами нашей работы, хотя мы надеемся сделать одно из таких исследований в будущем. Тем временем мы разработали один исходный показатель, который может быть применён к АОП технике. Этот показатель строится на сравнении ОП реализации приложения с АОП реализацией того же самого приложения. Он показывает степень, с которой аспекты более лаконично выражены в АОП реализации по сравнению с ОП реализацией. Ниже отображена общая формула и формула со значениями справедливыми для нашего приложения:

$$\begin{array}{l} \text{Уменьшение} \\ \text{раздувания} \\ \text{в связи со} \\ \text{спутыванием} \end{array} = \frac{\text{Размер спутанного кода} - \text{размер компонентного кода}}{\text{Суммарный код аспектов}} = \frac{35213 - 756}{352} = 98$$

В этой метрике, любое число больше чем 1 показывает положительную отдачу от применения АОП. Наше приложение сильно выиграло от использования АОП, в других приложениях которые мы разработали, отдача была от 2 до 98-ми [\[2, 14, 22\]](#). Можно сказать, что размер компоновщика так же должен быть включён в сумму делителя. Точка зрения обсуждаема, т.к. компоновщик может быть повторно использован любым количеством похожих систем обработки изображения. Но даже при включении объёма кода компоновщика значение показателя всё ещё остается достаточно большим.

Полезность каждой из отдельных метрик ограничена. Мы полагаем, что приведённая метрика является одной из полезных в данном случае, т.к. с точки зрения производительности приложения АОП реализация сравнима с обычным приложением. Раздел 7 представит некоторые из требований, которые мы выделили для качественных оценок полезности АОП.

6. Библиография

1. Aksit M., Wakita K., et al., Abstracting object interactions using composition filters, in proc. ECOOP'93 Workshop on Object-Based Distributed Programming, pp. 152-184, 1993.
2. Bobrow D. G., DeMichiel L. G., et al., Common Lisp Object System Specification, in SIGPLAN Notices, vol. 23, 1988.
3. Chiba S., A Metaobject Protocol for C++, in proc. Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 95), Austin, 1995.
4. Consel C., Program Adaptation based on Program Transformation, in proc. ACM Workshop on Strategic Directions in Computing Research, 1996.
5. Harrison W. and Ossher H., Subject-oriented programming (a critique of pure objects), in proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications, pp. 411--428, Washington D.C., 1993.
6. Henry S. and Kafura D., Software Structure Metrics Based on Information Flow, in IEEE Transactions on Software Engineering, vol. SE-7: 509--518, 1981.
7. Ichisugi Y., Matsuoka S., et al., Rbc1: A reflective object-oriented concurrent language without a run-time kernel, in proc. International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture, pp. 24--35, 1992.
8. Irwin J., Loingtier J.-M., et al., Aspect-Oriented Programming of Sparse Matrix Code, Xerox PARC, Palo Alto, CA. Technical report SPL97-007 P9710045, February, 1997
9. Kiczales G., Foil for the Workshop on Open Implementation, Xerox PARC, Web pages, www.parc.xerox.com/spl/eca/oi/workshop-94/foil/main.html
10. Kiczales G., Why are Black Boxes so Hard to Reuse?, Invited Talk, OOPSLA'94, Video tape, Web pages, <http://www.parc.xerox.com/spl/eca/oi/gregor-invite.html>
11. Kiczales G., des Rivieres J., et al., The Art of the Metaobject Protocol. Book published by MIT Press, 1991.
12. Kiczales G., Lamping J., et al., Open Implementation Design Guidelines, in proc. International Conference on Software Engineering, (Forthcoming), 1997.
13. Lieberherr K. J., Silva-Lepe I., et al., Adaptive Object-Oriented Programming Using Graph-Based Customization, in Communications of the ACM, vol. 37(5): 94-101, 1994.
14. Lopes C. V. and Kiczales G., D: A Language Framework for Distributed Programming, Xerox PARC, Palo Alto, CA. Technical report SPL97-010 P9710047, February, 1997.
15. Lopes C. V. and Lieberherr K., Abstracting Process-to-Function Relations in Concurrent

- Object-Oriented Applications, in proc. European Conference on Object-Oriented Programming, pp. 81-99, Bologna, Italy, 1994.
16. Lopes C. V. and Lieberherr K., AP/S++: Case-Study of a MOP for Purposes of Software Evolution, in proc. Reflection'96, pp. 167-184, S. Francisco, CA, 1996.
 17. Maeda C., Lee A., et al., Open Implementation Analysis and Design, in proc. Symposium on Software Reuse (To Appear, May 1997), 1997.
 18. Mahoney J. V., Functional Visual Routines, Xerox Palo Alto Research Center, Palo Alto SPL95-069, July 30, 1995, 1995.
 19. Massalin H. and Pu C., Threads and Input/Output in the Synthesis Kernel, in Proceedings of the 12th ACM Symposium on Operating Systems Principles :pp 191-201, 1989.
 20. Matsuoka S., Watanabe T., et al., Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming, in European Conference on Object Oriented Programming:pp 231-250, 1991.
 21. McClure C., A Model for Program Complexity Analysis, in proc. 3rd International Conference on Software Engineering, Los Alamitos, CA, 1978.
 22. Mendhekar A., Kiczales G., et al., RG: A Case-Study for Aspect-Oriented Programming, Xerox PARC, Palo Alto, CA. Technical report SPL97-009 P9710044, February, 1997.
 23. Okamura H., Ishikawa Y., et al., AI-1/d: A distributed programming system with multi-model reflection framework, in proc. International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture, pp. 36--47, 1992.
 24. Okamura H., Ishikawa Y., et al., Metalevel Decomposition in AI-1/D, in proc. International Symposium on Object Technologies for Advanced Software, pp. 110-127, 1993.
 25. Parnas D. L., Designing Software for Extension and Contraction, in proc. 3rd International Conference on Software Engineering, pp. 264-277, 1978.
 26. Parnas D. L., On a 'Buzzword': Hierarchical Structure, in proc. IFIP Congress 74, pp. 336-339, 1974.
 27. Parnas D. L., On the Criteria to be Used in decomposing Systems into Modules, in Communications of the ACM, vol. 15(2), 1972.
 28. Pu C., Autrey T., et al., Optimistic Incremental Specialization: Streamlining a Commercial Operating System, in proc. 15th ACM Symposium on Operating Systems Principles (SOSP'95), 1995.
 29. Rational, Rational Web pages, Rational Software Corporation, Web pages, <http://www.rational.com>

30. Rumbaugh J., Blaha M., et al., Object-Oriented Modeling and Design. Book published by Prentice Hall, 1991.
31. Seiter L. M., Palsberg J., et al., Evolution of Object Behavior Using Context Relations, in proc. Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 46--57, San Francisco, 1996.
32. Smith B. C., Reflection and Semantics in a Procedural Language LCS Technical Report, M.I.T., Cambridge, MA, 1982.
33. Steele G. L., Common LISP: The Language, 2nd Edition. Book published by Digital Press, 1990.
34. Wand M. and Friedman D. P., The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower, in Proceedings of the ACM Conference on LISP and Functional Programming :pp 298-307.ACM, 1986.
35. Watanabe T. and Yonezawa A., Reflection in an object-oriented concurrent language, in proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 88), pp. 306--315, San Diego, CA, 1988.
36. Yau S. and Collofello J., Some Stability Measures for Software Maintenance, in tse, vol. SE-6: 545--552, 1980.
37. Yokote Y., The Apertos Reflective Operating System: The Concept and its Implementation, in proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications, 1992.
38. Yonezawa A. and Watanabe T., An Introduction to Object-Based Reflective Concurrent Computation, in Proceedings of the ACM SIGPLAN

7. Примечания

1. Статья была опубликована в 1997 году
2. В некоторых сообществах этот термин так же ассоциируется с использованием в функционально-ориентированных языках программирования, но мы не используем его в этом смысле.
3. Мы выбрали синтаксис общего ЛИСПа (Common Lisp) для этого примера, но тоже самое может быть с лёгкостью написано на любом другом похожем на Агол языке.
4. Наша АОП реализация полного приложения также объединяет и эти циклы. Мы намеренно не показываем этот код т.к. он значительно сложнее для понимания.
5. Компоненты друг от друга, аспекты друг от друга, и компоненты от аспектов

6. Наш анализ аспектов, как свойств системы пересекающих компоненты помогает объяснить постоянную популярность таких механизмов как динамическая область видимости, "catch" & "throw", в ОП языках. Эти механизмы предоставляют различные возможности компоновки, помогающие программистам реализовать некоторые аспекты в их системах.
7. Просьба программисту явно указывать реализации аспектов может показаться шагом назад, но основываясь на нашем опыте можно сказать что это не так [\[9, 10, 12, 17\]](#). В то время как программист явно указывает место вплетения аспекта, например аспекта использования памяти, соответственное использование АОП значит, что он выражает стратегию реализации на соответствующем уровне абстракции, через соответствующий язык аспектов, с соответствующей локализацией. Программист не указывает особенности реализации, и программист не работает со спутанным кодом.
8. Наш кодогенератор в настоящий момент не использует упакованные структуры данных, как результат, 4-х кратное потеря производительности между оптимизированной вручную и АОП реализацией. Но тем не менее АОП реализация, по крайней мере, в 100 раз быстрее, чем неоптимизированная.

Copyright © 1997 Springer-Verlag. Перевод на русский © Ренат Зубаров, 2003