УДК 004.43

О.Н. Половикова

Анализ преимуществ использования системы CLOS для реализации объектных моделей

O.N. Polovikova

An Analysis of the Benefits to Use System CLOS for Object Models' Implementation

Объектная система CLOS предлагает достаточно отличающуюся от других языков реализацию принципов объектно-ориентированного программирования, имеющую свои преимущества и недостатки. Цель настоящей работы – анализ возможностей системы CLOS, а также формирование базовых требований и условий к практическим задачам, для реализации которых следует использовать данную объектную систему.

Ключевые слова: CLOS, Common Lisp, множественная диспетчеризация, обобщенные функции, множественное наследование, мультиметоды, пакеты, скрывающие символы.

Object system CLOS offers realization of the OOP principles differing enough from the other languages and having its own advantages and disadvantages. The purpose of this study is to analyze the capabilities of the system CLOS and to formulate basic requirements and conditions to the practical problems for realization of which it is necessary to use a given object system.

Key words: CLOS, Common Lisp, multiple dispatching, generalized functions, multiple inheritances, multi-methods, packages, shadowing symbols.

В статье рассмотрена объектная система функционального языка Lisp, которая представляет интерес как для разработчиков программных приложений, так и для аналитиков объектно-ориентированной методологии. В опубликованном компанией TIOBE Software рейтинге популярности языков программирования за прошедший год (2010) язык Lisp с 23 места (апрель 2010 г.) передвинулся на 15 место (апрель 2011 г.) [1]. Промышленным стандартом языка Лисп является язык Common Lisp, который вытеснил большинство других диалектов. Один из немаловажных фактов, способствующий выбору в пользу этого языка для написания современных прикладных систем, – возможности объектной системы Common Lisp (CLOS). Следует заметить, что данная система предлагает достаточно отличающуюся от других языков реализацию принципов объектно-ориентированного программирования [2, 3]. Для организации поведения экземплярами класса применяется множественная диспетчеризация, в отличие от Си++ и Java, где для этих целей используется передача объектам сообщений.

Такой подход к реализации принципов полиморфизма и инкапсуляции классов имеет свои преимущества и недостатки. Явное отсутствие спецификаторов, позволяющих управлять видимостью атрибутов и методов класса при его построении, относят к изъянам при поверхностном анализе конструкций языка. При этом, как правило, не учитывается, что отсутствие привычной функциональности по инкапсуляции хранимых в классе данных можно частично перекрыть средствами сокрытия символов в текущем пакете. Цель этой работы – анализ возможностей системы CLOS, а также формирование базовых требований и условий к практическим задачам, для реализации которых следует использовать данную объектную систему.

Ниже представлены и основные возможности разработки прикладных систем на основе рассматриваемой объектной системы, полученные из анализа информационных источников [2, 3], а также опыта программирования на языке Common Lisp в среде Allegro Common Lisp 8.2 Free Express Edition. Формирование преимуществ с акцентом на специфику их практического применения и противопоставление с другими объектно-ориентированными языками позволили определить требования к прикладным задачам для приоритетного применения CLOS.

Комбинирование функциональности объектной системы и пакетной механики Common Lisp предоставляет следующие возможности разработки прикладных систем:

1) динамическое комбинирование действующего метода реализации;
2) реализация множественного наследования;
3) использование мультиметодов;
4) инкапсуляция объектов и функционала класса за счет использования скрывающих символов.

Динамическое комбинирование действующего метода реализации. Каждому вызову функции для объекта определенного класса может соответствовать несколько методов реализаций. Причем

в процессе работы с программой можно добавлять, удалять или модифицировать их реализации. Построение так называемого эффективного (действующего, рабочего) метода, который и будет отвечать за выполнение указанной функции, выполняется также динамически и зависит от выбранного или созданного комбинатора. Следует заметить, что данное преимущество обеспечивается не только механизмами множественной диспетчеризации методов, но и функциональностью языка Lisp.

В отличие от популярных языков Java и Си++ создание обобщенной функции и реализация для нее методов не зависят от места и времени объявления классов. В процессе работы Lisp-программы, реализующей одну из объектно-ориентированных моделей, можно изменять или добавлять подходящие методы, специализированные для конкретного класса, тем самым меняя функционал действующего метода.

Методы обобщенной функции можно использовать и в качестве аргументов, что позволяет их оборачивать функциями более высокого порядка. На этой возможности базируется реализация различных комбинаторов методов. Простые комбинаторы методов создают действующий (эффективный) метод, содержащий код всех основных методов, расположенных по порядку и обернутых вызовом функции, макросом или специальным оператором. Такой способ отсутствует в объектно-ориентированных языках, в которых для организации поведения экземплярами класса используется передача объектам сообщений.

Независимость объявления методов обобщенных функций от места и времени объявления классов позволяет размещать в разных программных блоках реализации методов и классов. В процессе работы интерпретатора можно модифицировать поведение классов (изменяя функциональную составляющую методов), при этом не требуется корректировать классы или объекты.

Реализация множественного наследования. Несмотря на то, что объектно-ориентированное программирование развивается уже более 30 лет, до сих пор необходимость в множественном наследовании остается предметом споров и дискуссий различного уровня [4–9]. Множественное наследование прямо поддерживается языком C++ и системой CLOS, а также до некоторой степени в Smalltalk. Существует множество популярных языков программирования, которые поддерживают объектно-ориентированную парадигму, но не поддерживают множественное наследование: C#, Java, Nemerle, PHP и др. В языках, которые позиционируются как наследники C++ (Java, C# и др.), от множественного наследования было решено отказаться в пользу интерфейсов.

В статьях периодических изданий и в учебной литературе приведены практические примеры использования данного вида наследования, что подтверждает реальную потребность в нем. В своем исследовании один из теоретиков объектно-ориентированного программирования Гради Буч подчеркивает необходимость реализации такого вида наследования: «По нашему опыту, множественное наследование – как парашют: как правило, он не нужен, но, когда вдруг он понадобится, будет жаль, если его не окажется под рукой» [6].

Выделим две основные проблемы, которые возникают при реализации схем множественного наследования:

1) конфликт имен;

2) повторное наследование.

Рассмотрим ситуацию, которая приводит к конфликту имен. В родительских классах разные компоненты названы одним именем, при одновременном наследовании потомок приобретает одноименные методы или атрибуты по разным веткам иерархии, что приводит к противоречию.

Повторное наследование (repeated inheritance) возникает, когда класс является потомком другого класса более чем на одном пути наследования. При этом возникает потенциальная неоднозначность, которую и следует разрешить. Об этой проблеме Мейер пишет следующее: «Одно тонкое затруднение при использовании множественного наследования встречается, когда один класс является наследником другого по нескольким линиям. Если в языке разрешено множественное наследование, рано или поздно кто-нибудь напишет класс D, который наследует от B и C, которые, в свою очередь, наследуют от A. Эта ситуация называется повторным наследованием, и с ней нужно корректно обращаться» [10].

Несмотря на возможные проблемы и неопределенности, связанные с использованием множественного наследования, можно выделить ряд ситуаций, когда множественное наследование можно удачно применять. В первую очередь, для объединения независимых или почти независимых иерархий классов, когда новый дочерний класс объединяет свойства и поведение нескольких родительских классов [6].

Система CLOS поддерживает множественное наследование, кроме того, две обозначенные выше проблемные ситуации успешно разрешаются конструкцией языка Common Lisp и механизмами наследования. Правила и механизмы реализации одиночного и множественного наследования принципиально не отличаются.

Для каждого класса строится собственный список приоритетности классов, в котором линейно упорядочены все родительские классы, причем любой родительский класс в этот список включается один раз. Таким образом, в объектной системе Common Lisp отсутствует проблема повторного наследования. Порядок, в котором родительские классы перечислены в определении класса-наследника, задает и порядок приоритетов наследования, при этом определения, сделанные в классе, наибо-

лее приоритетны, а любой прямой родитель имеет выше приоритет, чем класс предок родителя.

Список приоритетных классов позволяет в любой ситуации строить действующий (эффективный) метод для любого вызова функции. Так же, как и при одиночном наследовании, действующий метод может заключать в себе реализацию наиболее приоритетного (специфичного) метода или из нескольких приоритетных методов, обернутых вызовом определенной функции. Это предоставляет программисту дополнительный функционал возможностей по реализации специфики предметной области.

В Common Lisp конкретный объект может иметь только один слот с определенным именем, поэтому в CLOS отсутствует проблема неоднозначности интерпретации одноименных символов при наследовании от нескольких родителей. В ситуации, когда класс-наследник включает название слота с тем же именем, что и у слота в родительском классе, либо когда родительские классы имеют одинаковые имена слотов, потомок приобретает только один слот. Этот слот будет наделен своими спецификаторами, которые получены путем слияния всех спецификаторов с одним и тем же именем из нового класса и всех его родительских классов.

Таким образом, в механизмах объектной системы Common Lisp не возникает проблем, связанных с возможным конфликтом имен или с повторным наследованием, которые имеются в C++. Действующий метод для класса-потомка может комбинироваться из одноименных родительских методов различными способами по выбору программиста, а не только простым перенаправлением.

Использование мультиметодов. Одна из проблемных ситуаций связана со сложностью определения принадлежности к конкретному классу какого-либо действия, которое совершается при взаимодействии объектов разных классов. Исследователь Peter Seibel в одном из своих произведений описывает пример с барабаном: «Звук, который издает барабан, когда вы стучите по нему палочкой, является функцией барабана, или функцией палочки? Конечно, он принадлежит обоим предметам» [2]. Подобные спорные моменты возникают в процессе разработки информационных систем на языках с передачей объектам сообщений.

Разрешить такие ситуации можно несколькими способами. Например, можно определить принадлежность такого действия (реализация в виде отдельного метода) одному из классов, которые участвуют во взаимодействии. При этом необходимо данному методу предоставить полный доступ к свойствам объекта другого класса. Можно также создать внешний метод, дружественный каждому из классов. Но при любом известном способе решения описанной выше проблемы на языках с передачей экземплярам сообщений теряется структурированность объектно-ориентированной модели, тем более, что схожая ситуация наследуется производными классами.

В системе CLOS для действий, которые не могут принадлежать только одному классу, также используются обобщающие функции, но их реализация выполняется через мультиметоды. Мультиметоды используются в ситуациях, когда поведение относится к нескольким классам, они связывают независимые классы для описания их совместного поведения, при этом не требуется дополнительно прописывать никакого взаимодействия между ними.

Инкапсуляция объектов и функционала класса за счет использования скрывающих символов. Прямым назначением пакетов является возможность создания для своей программы собственного пространства имен, чтобы не возникало конфликтов между одноименными символами в различных программах, загруженных в одну Lisp-среду. Использование пакетов не позволяет контролировать доступ к объектам, их слотам или методам обобщенных функций, они явно не предоставляют механизмов для ограничения видимости символов, будь это имя переменной или имя функции. Но при этом есть возможность в рамках одного пакета регулировать доступность символов из других пакетов, а также составлять публичные API библиотеки, для их использования в других программных продуктах.

Пакетная механика системы CLOS разрешает, с одной стороны, использовать неспециализированные имена символов в пакетах, в которых эти символы не присутствуют. Тем самым упрощается механизм обращения к именам символов из разных пакетов. С другой стороны, **скрывающие** символы позволяют сохранить отображение имен в символы однозначным, делая другие символы с такими же именами недоступными. Так как пакетная система может определить только один доступный символ в данном пакете для каждого имени, то возможность скрывать **присутствующий** в пакете символ или наследуемый символ из одного или нескольких пакетов позволяет разрешить конфликт имен.

Отметим, что использование сокрытия символов в одном из пакетов не позволяет полностью закрывать доступ к каким-то свойствам или функциям из других пакетов, так как всегда существует возможность обратиться к этим символам через специализированное имя. Использование имени с двойным двоеточием (название пакета :: имя символа) позволяет обратиться к любому символу из любого доступного пакета. Но использование таких специализированных имен считается плохим тоном, так как происходит нарушение решений автора пакета: какие имена являются внешними и определяют публичный интерфейс, а какие внутренними, и их нельзя использовать в других пакетах.

При использовании спецификаторов видимости в таких объектно-ориентированных языках, как C++ и Java, для инкапсуляции данных также возникают спорные моменты, связанные с закрытым наследова-

нием открытых или защищенных атрибутов или методов. Рассмотрим, например, ситуацию: в базовом классе (class1) объявлен какой-либо открытый атрибут класса (attribute) или метод, закрытым наследованием сформирован производный класс (class2), с целью сокрытия (инкапсуляции) всех наследуемых свойств и поведения, создан объект производного класса (object2). К свойствам и поведению базового класса для объекта производного класса, используя точечный оператор, нельзя обратиться из внешних функций (object2.attribute). Но такой запрет можно обойти, используя оператор разрешающий пространство имен (object2.class1:: attribute).

Анализ основных возможностей, которые предоставляет разработчику множественная диспетчеризация методов, в сравнении с передачей объектам сообщений для организации поведения экземплярами классов, а также особенности функционального аппарата системы CLOS позволили сформировать требования к созданию программных продуктов, для выполнения которых следует использовать данную объектную систему:

1. Алгоритм исполнения метода, реализующего поведение объектов класса, определяется только в момент работы с программой.

2. Методы производного класса строятся на каком-либо способе комбинирования методов классов родителей (суммирование, построение списка результатов и т.д.).

3. Методы производного класса используют специфическую выборку реализаций родительских методов, которая не отвечает их последовательному линейному выполнению.

4. Производные классы и их функционал создаются в результате множественного наследования.

5. Поведение, которое необходимо реализовать в рамках объектной модели, относится к нескольким независимым классам.

В заключение следует отметить, что представленные в статье преимущества объектной системы Common Lisp не перекрывают весь объем возможностей множественной диспетчеризации, используемой для организации поведения объектов. Тем не менее проведенный анализ CLOS в сравнении с другими объектно-ориентированными системами поможет разработчикам делать выбор в пользу определенного языка программирования в зависимости от специфики предметной области и требований прикладной задачи.

Библиографический список

1. TIOBE Programming Community Index for April 2011 April Headline: Lua is approaching the top 10 of the TIOBE index. URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. 2011. (дата обращения 20.05.2011).

2. Kent M. Pitman. Condition handling in the lisp language family. – 2001.

3. Peter Seibel. Practical Common Lisp. – 2006.

4. Ален И. Голуб. Правила программирования на Си и Си++: пер. с анг. / под ред. В. Зацепина. – М., 2001.

5. Страуструп Б. Дизайн и эволюция С++: пер. с англ. – М., 2000.

6. Гради Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на С++: пер. с анг. / под ред. И. Романовского, Ф. Андреева. – 2-е изд. – Калифорния, 1998.

7. Основы объектно-ориентированного программирования. Множественное наследование // eXcode e-zine :: On-line журнал для программистов. – 2006. URL : http://www.excode.ru/art6131p2.html (дата обращения 20.05.2011).

8. Труба И. О проблемах множественного наследования // Открытые системы. – 2001. – №2.

9. Легалов А. О стрельбе по множественному наследованию // Открытые системы. – 2001. – №05-06.

10. Meyer B. Object-oriented Software Construction. – New York, 1988.