Ханспетер Мессенбок

Плюсы и минусы объектно-ориентированного программирования

Hanspeter Moessenboeck (1993) Costs and Benefits of OOP // In Object-Oriented Programming in Oberon-2 // Springer-Verlag.

Р. Богатырев, перевод с англ.

Аннотация

Эта глава из книги Х.Мессенбока Object-Oriented Programming in Oberon-2 представляет собой лаконичное изложение достоинств и недостатков объектно-ориентированного программирования (ООП). Затрагиваются такие вопросы, как расширяемость систем, накладные расходы на поддержку ООП и производство законченных продуктов из программных полуфабрикатов.

В этой книге мы пытались показать, в каких ситуациях классы полезны, а когда их использовать не следует. Давайте теперь подведем некоторые итоги и зададимся следующими вопросами.

- Почему нам следует программировать объектно-ориентированным способом, а не процедурным?
- Каковы плюсы и минусы объектно-ориентированного программирования?
- Перевешивают ли достоинства имеющиеся недостатки?

Если программист четко себе представляет силу и предел возможностей объектно-ориентированного программирования (ООП) и если он использует классы осознанно, то достоинства превзойдут недостатки. Однако негативные моменты могут стремительно нарастать, если классы применять бездумно, особенно в тех ситуациях, когда они не только не уменьшают проблемы, а наоборот, лишь добавляют сложности.

Достоинства ООП

От любого подхода в программировании мы ждем, что он поможет нам в решении наших проблем. Но одной из самых значительных проблем здесь является сложность. Чем больше и сложнее программа, тем важнее разбить ее на небольшие, четко очерченные части. Чтобы побороть сложность, мы должны абстрагироваться от мелких деталей. В этом смысле классы представляют собой весьма удобный инструмент.

- Классы позволяют проводить конструирование из полезных компонентов, обладающих простыми инструментами, что дает возможность абстрагироваться от деталей реализации.
- Данные и операции вместе образуют определенную сущность и они не "размазываются" по всей программе, как это нередко бывает в случае процедурного программирования.
- Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.
- Инкапсуляция информации защищает наиболее критичные данные от несанкционированного доступа.

ООП дает возможность создавать расширяемые системы (extensible systems). Это одно из самых значительных достоинств ООП и именно оно отличает данный подход от традиционных методов программирования. Расширяемость (extensibility) означает, что существующую систему можно заставить работать с новыми компонентами, причем без внесения в нее каких-либо изменений. Компоненты могут быть добавлены на этапе выполнения.

Расширение типа (type extension) и вытекающий из него полиморфизм переменных оказываются полезными преимущественно в следующих ситуациях.

- Обработка разнородных структур данных. Программы могут работать, не утруждая себя изучением вида объектов. Новые виды могут быть добавлены в любой момент.
- Изменение поведения во время выполнения. На этапе выполнения один объект может быть заменен другим. Это способно привести к изменению алгоритма, в котором используется данный объект.
- Реализация родовых компонентов. Алгоритмы можно обобщать до такой степени, что они уже смогут работать более чем с одним видом объектов.
- Доведение полуфабрикатов. Компоненты нет надобности подстраивать под определенное приложение. Их можно сохранять в библиотеке в виде полуфабрикатов (semifinished products) и расширять по мере необходимости до уровня различных законченных продуктов.
- Расширение каркаса. Независимые от приложения части предметной области могут быть реализованы в виде каркаса и в дальнейшем расширены за счет добавления частей, специфичных для конкретного приложения.

Многократного использования программного обеспечения на практике добиться не удается из-за того, что существующие компоненты уже не отвечают новым требованиям. ООП помогает этого достичь без нарушения работы уже имеющихся клиентов, что позволяет нам извлечь максимум полезного от многократного использования компонентов.

- Мы сокращаем время на разработку, которое с выгодой может быть отдано другим проектам.
- Компоненты многократного использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.
- Когда некий компонент используется сразу несколькими клиентами, то улучшения, вносимые в его код, одновременно оказывают свое положительное влияние и на множество работающих с ним программ.
- Если программа опирается на стандартные компоненты, то ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает ее использование.

Недостатки ООП

Объектно-ориентированное программирование требует знания четырех вещей.

(1) Необходимо понимать базовые концепции, такие как классы, наследование и динамическое связывание. Для программистов, уже знакомых с понятием модуля и с абстрактными типами данных, это потребует минимальных усилий. Для тех же, кто никогда не использовал инкапсуляцию данных, это может означать изменение мировоззрения и способно отнять значительное количество времени на изучение.

(2) Многократное использование требует от программиста познакомиться с большими библиотеками классов. А это может оказаться сложнее, чем даже изучение нового языка программирования. Библиотека классов фактически представляет собой виртуальный язык, который может включать в себя сотни типов и тысячи операций. В языке Smalltalk, к примеру, до того, как перейти к практическому программированию, нужно изучить значительную часть его библиотеки классов. А это тоже требует времени.

(3) Проектирование классов — задача куда более сложная, чем их использование. Проектирование класса, как и проектирование языка, требует большого опыта. Это итеративный процесс, где приходится учиться на своих же ошибках.

(4) Очень трудно изучать классы, не имея возможности их "пощупать". Только с приобретением мало-мальского опыта можно уверенно себя почувствовать при работе с использованием ООП. Как мы видели, усилия на освоение базовых концепций невелики, но вот в случае библиотек классов и их использования они могут быть очень существенными.

Поскольку детали реализации классов обычно неизвестны, то программисту, если он хочет разобраться в том или ином классе, нужно опираться на документацию и на используемые имена.

И время, которое было сэкономлено на том, что удалось обойтись без написания собственного класса, должно быть отчасти потрачено (особенно вначале освоения) на то, чтобы разобраться в существующем классе.

Документирование классов — задача более трудная, чем это было в случае процедур и модулей. Поскольку любой метод может быть переопределен, в документации должно говориться не только о том, что делает данный метод, но также и о том, в каком контексте он вызывается. Ведь переопределенные методы обычно вызываются не клиентом, а самим программным каркасом. Таким образом, программист должен знать, какие условия выполняются, когда вызывается данный метод. Для абстрактных методов, которые пусты, в документации должно даже говориться о том, для каких целей предполагается использовать переопределяемый метод.

В сложных иерархиях классов поля и методы обычно наследуются с разных уровней. И не всегда легко определить, какие поля и методы фактически относятся к данному классу. Для получения такой информации нужны специальные инструменты вроде навигаторов классов. Если конкретный класс расширяется, то каждый метод обычно сокращают перед передачей сообщения базовому классу. Реализация операции, таким образом, рассредоточивается по нескольким классам, и чтобы понять, как она работает, нам приходится внимательно просматривать весь код.

Методы, как правило, короче процедур, поскольку они осуществляют только одну операцию над данными. Зато количество методов намного выше. Короткие методы обладают тем преимуществом, что в них легче разбираться, неудобство же их связано с тем, что код для обработки сообщения иногда "размазан" по многим маленьким методам.

Абстракция данных ограничивает гибкость клиентов. Клиенты могут лишь выполнять те операции, которые предоставляет им тот или иной класс. Они уже лишены неограниченного доступа к данным.

Причины здесь аналогичны тем, что вызвали к жизни использование высокоуровневых языков программирования, а именно, чтобы избежать непонятных программных структур.

Абстракцией данных не следует злоупотреблять. Чем больше данных скрыто в недрах класса, тем сложнее его расширять. Отправной точкой здесь должно быть не то, что клиентам не разрешается знать о тех или иных данных, а то, что клиентам для работы с классом этих данных знать не требуется.

Часто можно слышать, что ООП является неэффективным. Как же дело обстоит в действительности? Мы должны четко проводить грань между неэффективностью на этапе выполнения, неэффективностью в смысле распределения памяти и неэффективностью, связанной с излишней универсализацией.

(1) Неэффективность на этапе выполнения. В языках типа Smalltalk сообщения интерпретируются во время выполнения программы путем осуществления поиска их в одной или нескольких таблицах и за счет выбора подходящего метода. Конечно, это медленный процесс. И даже при использовании наилучших методов оптимизации Smalltalk-программы в десять раз медленнее оптимизированных Си-программ [1].

В гибридных языках типа Oberon-2, Object Pascal и C++ посылка сообщения приводит лишь к вызову через указатель процедурной переменной. На некоторых машинах сообщения выполняются лишь на 10% медленнее, чем обычные процедурные вызовы. И поскольку сообщения встречаются в программе гораздо реже других операций, их воздействие на время выполнения влияния практически не оказывает. Однако, существует другой фактор, который затрагивает время выполнения: это абстракция данных. Абстракция запрещает непосредственный доступ к полям класса и требует, чтобы каждая операция над данными выполнялась через методы. Такая схема приводит к необходимости выполнения процедурного вызова при каждом доступе к данным. Однако, когда абстракция используется только там, где она необходима (т.е. не из одной лишь прихоти), то замедление вполне приемлемое.

(2) Неэффективность в смысле распределения памяти. Динамическое связывание и проверка типа на этапе выполнения требуют по ходу работы информации о типе объекта. Такая информация хранится в дескрипторе типа, и он выделяется один на класс. Каждый объект имеет невидимый указатель на дескриптор типа для своего класса. Таким образом, в объектно-

ориентированных программах требуемая дополнительная память выражается в одном указателе для объекта и в одном дескрипторе типа для класса.

(3) Излишняя универсальность. Неэффективность может также означать, что программа имеет ненужные возможности. В библиотечном классе часто содержится больше методов, чем это реально необходимо. А поскольку лишние методы не могут быть удалены, то они становятся мертвым грузом. Это не воздействует на время выполнения, но влияет на возрастание размера кода.

Одно из возможных решений — строить базовый класс с минимальным числом методов, а затем уже реализовывать различные расширения этого класса, которые позволят нарастить функциональность. Другой подход — дать возможность компоновщику удалять лишние методы. Такие интеллектуальные компоновщики уже доступны для различных языков и операционных систем. Oberon избрал третий путь избавления от излишней универсальности. Программные части могут добавляться на этапе выполнения. Таким образом, нет надобности загружать всю программу целиком, а можно обойтись лишь теми ее частями, которые в данный момент необходимы. Как показала практика, это экономит гораздо больше кода, чем можно добиться при удалении лишних методов.

Таким образом, нельзя утверждать, что ООП вообще неэффективно. Если классы используются лишь там, где это действительно необходимо, то потеря эффективности и на этапе выполнения и в смысле памяти сводится практически на нет.

Будущее ООП

Выживет ли объектно-ориентированное программирование, или оно лишь модное поветрие, которое скоро исчезнет?

Классы нашли свое место в большинстве современных языков программирования. Одно лишь это говорит о том, что им суждено остаться. Классы в самом ближайшем будущем войдут в стандартный набор концепций для каждого программиста, точно так же, как многие сегодня применяют динамические структуры данных и рекурсию, которые двадцать лет назад также были в диковинку. В то же время классы — это просто еще одна новая конструкция наряду с остальными. Нам нужно узнать, для каких ситуаций они подходят, и только здесь мы и будем их использовать.

Правильно выбрать инструмент для конкретной задачи — обязательное требование для каждого исполнителя и в еще большей степени для каждого инженера.

ООП ввергает многих в состояние эйфории. Пестрящая тут и там реклама сулит нам невероятные вещи, и даже некоторые исследователи, похоже, склонны рассматривать ООП как панацею, способную решить все проблемы разработки программного обеспечения. Со временем эта эйфория постепенно уляжется. И после периода разочарования люди, быть может, перестанут уже говорить об ООП, точно также как сегодня вряд ли от кого можно услышать о структурном программировании. Но классы будут использовать как нечто само собой разумеющееся, и мы сможем, наконец, понять, что они собой представляют: просто компоненты, которые помогают строить модульное и расширяемое программное обеспечение.

Литература

[1] Chambers C. (1992) The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages // Stanford University, Ph.D. thesis.

Об авторе. Ханспетер Мессенбок (Hanspeter Moessenboeck) — известный специалист в области компиляторов, языков и систем программирования, объектно-ориентированного программирования, программной инженерии; разработчик экспериментального языка Object Oberon, вместе с Никлаусом Виртом (Niklaus Wirth) является создателем языка Oberon-2. Он участвовал в ряде проектов, проводимых в Швейцарском Федеральном технологическом институте (Swiss Federal Institute of Technology), был профессором в департаменте компьютерных наук (Computer Science Department) в ETH (Цюрих, Швейцария). Он автор известной книги "Object-Oriented Programming in Oberon-2" (Springer-Verlag, 1993).