Метапрограммирование

Мультипарадигменный подход в инженерии программного обеспечения

Киев, 2015 — 2022

Аннотация

Все программы - это данные. Одни данные интерпретируются, как значения, а другие - как типы этих значений, третьи - как инструкции по обработке первых двух. Любые парадигмы и техники программирования - это лишь способ формировать метаданные, дающие правила и последовательность потока обработки других данных. Мультипарадигменное программирование берет лучшее из всех парадигм и строит из них синтаксические конструкции, позволяющие более понятно и удобно описать предметную область. Мы связываем высокоуровневые DSL (доменные языки) с низкоуровневыми машинными инструкциями через множество слоев абстракций. Тут важно не фанатично следовать одной парадигме, а наиболее эффективно отображать задачу для исполнения на машинном уровне. Наиболее эффективно - это с меньшим количеством слоев и зависимостей, наиболее удобно для понимания человеком, для поддержки и модификации, обеспечения надежности и тестируемости кода, расширяемости, переиспользования, ясности и гибкости конструкций метаданных на каждом уровне. Мы полагаем, что такой подход позволит получать как быстрые первые результаты в разработке каждой задачи, так и не терять темпов при большом потоке изменений на этапах, когда проект уже достиг высокой зрелости и сложности. Мы постараемся рассмотреть приемы и принципы программирования из разных парадигм через призму метапрограммирования и не столько изменить этим программную инженерию, как ее осмысление новыми поколениями инженеров.

Оглавление

(машины состояний)

11.4. Специализированные языки для предметных областей (DSL)

11.5. Программирование на потоках данных

11.6. Метапрограммирование

11.7. Динамическая интерпретация метамодели

12. Базы данных и постоянное хранение

12.1. История баз данных и навигационные базы данных

12.2. Ключ-значение и другие абстрактные структуры данных

12.3. Реляционная модель данных и ER-диаграммы

12.4. Бессхемные, объектно- и документо-ориентированные базы данных

12.5. Иерархическая модель данных и графовые базы данных

12.6. Колоночные базы данных и in-memory базы данных

12.7. Распределенные базы данных

13. Распределенные системы

13.1. Межпроцессное взаимодействие

13.2. Бесконфликтные реплицированные типы данных (CRDT)

13.3. Согласованность, доступность и распределенность

13.4. Стратегии разрешения конфликтов

13.5. Протоколы консенсуса

13.6. CQRS, EventSourcing

1. Введение

Постоянное переосмысление своей деятельности, даже самой простой, должно сопровождать инженера всю его жизнь. Привычка записывать свои мысли словами и оттачивать формулировки очень помогает в этом. Текст этот появился как мои отрывочные заметки, написанные в разные годы, которые я накапливал и критически вычитывал десятки раз. Часто, я не соглашался с самим собой, перечитывая отрывок после того, как он полежит некоторое время. Поэтому, я доводил текст до того, пока сам не соглашался с написанным после продолжительных периодов выдержки материала. Своей задачей я принял писать как можно более кратко и неоднократно переписывал большие фрагменты, находя, что их можно выразить короче. Структура текста и оглавление начали появляться после первого года преподавания, но на десятом году я решил выложить все материалы не только в виде открытых видеолекций, как делал уже около пяти лет, но и в виде текста. Это позволило участвовать в формировании книги всем желающим из сообщества Метархия, быстро находить опечатки и неточности благодаря читателям, а также многим просто удобнее воспринимать в виде книги. Актуальную версию всегда можно найти на https:// github.com/ HowProgrammingWorks/ Book, она будет дополняться постоянно. Прошу присылать запросы на исправления и дополнения в [issues] (https:// github.com/ HowProgrammingWorks/ Book/ issues) на английском языке, новые идеи в [discussions] (https:// github.com/ HowProgrammingWorks/ Book/ discussions) на любом языке, а свои дополнения и исправления оформлять в виде pull-request в репозиторий книги.

Программирование - это искусство и инженерия решения задач при помощи вычислительной техники. Инженерия, потому, что оно призвано извлекать пользу из знаний, а искусство, потому, что знаниями программирование на современном этапе развития, к сожалению, не ограничивается и вынуждено прибегать к интуиции и слабо осмысленному личному опыту. Задача программиста не в нахождении математически верного решения, а в отыскании обобщенного механизма решения, способного нас приводить к нахождению приемлемого решения за ограниченное время в как можно большем классе задач. Другими словами, в нахождении абстрактного класса решений. Не все парадигмы программирования предполагают решение пошаговое, но

физическая реализация вычислительной техники и природа человеческого мышления предполагают пошаговость. Сложность в том, что эти действия далеко не всегда сводятся к машинным операциям и вовлекают внешнее взаимодействие с устройствами ввода/ вывода и датчиками, а через них, с внешним миром и человеком. Это обстоятельство создает большую неопределенность, которая не позволяет математически строго доказать правильность способа решения всех задач и, тем более, строго вывести такое решение из аксиом, как это характерно для точных наук. Однако, отдельные алгоритмы могут и должны быть выведены аналитически, если они сводимы к чистым функциям. То есть, к функциям, которые в любой момент для определенного набора входных данных однозначно дают один и тот же результат. Чистая функция не имеет истории (памяти или состояния) и не обращается ко внешним устройствам (которые могут такое состояние иметь), может обращаться только к другим чистым функциям. От математики программирование унаследовало возможность находить точные решения аналитически, да и сама вычислительная машина функционирует строго в рамках формального математического аппарата. Но процесс написания программного кода не всегда может быть сведен к формальным процедурам, мы вынуждены принимать решения в условиях большой неопределенности и конструировать программы инженерно. Программист ограничен и временем разработки программы, поэтому, мы сокращаем неопределенность благодаря введению конструктивных ограничений, не являющихся при этом строго выводимыми из задачи и основанных на интуиции и опыте конкретного специалиста. Проще говоря, за неимением оптимального алгоритма, программист может решать задачу любым способом, который дает приемлемые результаты за разумное время, и который может быть реализован за такое время, пока задача еще остается актуальной. В таких условиях мы должны принимать во внимание не только меру приближения решения к оптимальному, но и знания программиста, владение инструментарием и другие ресурсы, имеющиеся в наличии. Ведь даже доступ к знаниям уже готовым программным решениям ограничен авторским правом, правами владения исходным кодом и документацией, соответствующими лицензионными ограничениями, не только на программные продукты, но и на книги, видео, статьи, обучающие материалы, и т.д. Все это существенно усложняет и замедляет развитие отрасли, но со временем доступность знаний необратимо растет, они просачиваются в свободное хождение в сети через популяризаторов, энтузиастов и

движение свободного программного обеспечения.

1.1. Подход к изучению программирования

1.2. Примеры на языках JavaScript, Python и C

1.3. Моделирование: абстракции и повторное использование

В основе любого программирования, лежит моделирование, то есть, создание модели решения задачи или модели объектов и процессов в памяти машины. Языки программирования предоставляют синтаксисы для конструирования ограничений при создании моделей. Любая конструкция и структура, призванная расширить функциональность и введенная в модель, приводит к дополнительным ограничениям. Повышение же уровня абстракции, наоборот, может снимать часть ограничений и уменьшать сложность модели и кода программы, выражающего эту модель. Мы все время балансируем между расширением функций и сверткой их в более обобщенную модель. Этот процесс может и должен быть многократно итеративным.

Удивительно, но человек способен успешно решать задачи, сложность которых превышает возможности его памяти и мышления, при помощи построения моделей и абстракций. Точность этих моделей определяет их пользу для принятия решений и выработки управляющих воздействий. Модель всегда не точна и отображает только малую часть реальности, одну или несколько ее сторон или аспектов. Однако, в ограниченных условиях использования, модель может быть неотличимой от реального объекта предметной области. Есть физические, математические, имитационные и другие модели, но нас будут интересовать, в первую очередь, информационные и алгоритмические модели.

Абстракция, это способ обобщения, сводящий множество различных, но схожих между собой случаев, к одной модели. Нас интересуют абстракции данных и абстрактные алгоритмы. Самые простые примеры абстракции в алгоритмах, это циклы (итерационное обобщение) и функции (процедуры и подпрограммы). При помощи цикла мы можем описать множество

итераций одним блоком команд, предполагая его повторяемость несколько раз, с разными значениями переменных. Функции так же повторяются много раз с разными аргументами. Примеры абстракции данных, это массивы, ассоциативные массивы, списоки, множества и т.д. В приложениях абстракции нужно объединять в уровни - слои абстракций. Низкоуровневые абстракции встроены в язык программирования (переменные, функции, массивы, события). Абстракции более высокого уровня содержатся в программных платформах, рантаймах, стандартных библиотеках, и внешних библиотеках или их можно построить самостоятельно из простых абстракций. Абстракции так называются, потому, что решают абстрактные обобщенные задачи общего назначения, не связанные с предметной областью.

Построение слоев абстракций это чуть ли не самая важная задача программирования от удачного решения которой зависят такие характеристики программного решения, как гибкость настройки, простота модификации, способность к интеграции с другими системами и период жизни решения. Все слои, которые не привязаны к предметной области и конкретным прикладным задачам, мы будем называть системными. Над системными слоями программист надстраивает прикладные слои, абстракция которых наоборот снижается, универсальность уменьшается и конкретизируется применение, привязываясь к конкретным задачам.

Абстракции разных уровней могут находиться как в одном адресном пространстве (одном процессе или одном приложении), так и в разных. Отделить их один от другого и осуществить взаимодействие между ними можно при помощи программных интерфейсов, модульности, компонентного подхода и просто усилием воли, избегая прямых вызовов из середины одного программного компонента в середину другого, если язык программирования или используемая платформа не заботятся о предотвращении такой возможности. Так следует поступать даже внутри одного процесса, где можно было бы обращаться к любым функциям, компонентам и модулям из любых других, даже если они логически относятся к разным слоям. Причина этого в необходимости понизить связанность слоев и программных компонентов, обеспечив их взаимозаменяемость, повторное использование и делая возможной их раздельную разработку. Одновременно нужно повышать связность внутри слоев, компонентов и модулей, что обеспечивает рост

производительности кода, простоту его чтения, понимания и модификации. Если же нам удастся избегать связанности между разными уровнями абстракций, и при помощи декомпозиции добиться того, чтобы один модуль всегда мог быть полностью охвачен вниманием одного инженера, то процесс разработки становится масштабируемым, управляемым и более предсказуемым. Подобная идея положена в основу архитектуры микросервисов, но более общий принцип применим для любых систем, и не важно, будут ли это независимо запущенные микросервисы или модули, запущенные в одном процессе.

Нужно отметить, что чем лучше система распределена, тем лучше она централизована. Потому, как решения задач в таких системах находятся на адекватных уровнях, где уже достаточно информации для принятия решений, обработки и получения результата, отсутствует жесткая связанность моделей разного уровня абстракции. При таком подходе не происходит излишних эскалаций задачи на верхние уровни, избегаются “перегревы” узлов принятия решений, минимизирована передача данных и повышено оперативное быстродействие.

1.4. Алгоритм, программа, синтаксис, язык

1.5. Декомпозиция и разделение ответственности

1.6. Обзор специальности инженер-программист

1.7. Обзор парадигм программирования

Математик рассматривает программу, как функцию, которую можно декомпозировать (разделить) на более простые функции так, чтобы программа-функция была их суперпозицией. То есть, грубо говоря, программа есть сложной формулой, преобразователем данных, когда на вход подаются условия задачи, а на выходе мы получаем решение. Не всякий программист знаком с этой точкой зрения, хоть она и не идеальная, но полезная для переосмысления своей деятельности. Более распространена противоположная точка зрения, которую легко получить из практики программирования.

Заключается она в написании программ исходя из представления пользователя, из рисунков экранов пользовательского интерфейса, и из инструментария, языка, платформы и библиотек. В результате, мы получаем не программу-функцию, а большую систему состояний, в которой происходит комбинаторный взрыв переходов и поведение которой непредсказуемо даже для автора, не то что для пользователя. Но нельзя сразу отметать этот, казалось бы, ужасный подход. В нем есть конструктивное зерно, и заключается оно в том, что не все программы возможно в краткие сроки реализовать в функциональной парадигме, как преобразователями данных. Тем более, что человеческая деятельность свя состоит из шагов и изменения состояний окружающих нас предметов по принципу пошаговых манипуляций ими, а представить ее в виде функций было бы достаточно неестественным для нашего мышления.