74 Глава 5: Дизайн в конструировании

5.1 Сложности дизайна

Фраза “дизайн программного обеспечения” означает концепцию, изобретение схемы для превращения спецификации компьютерного программного обеспечения в операционное программное обеспечение. Дизайн - это деятельность, которая связывает требования с кодированием и отладкой. Хороший дизайн верхнего уровня обеспечивает структуру, которая может безопасно содержать несколько конструкций нижнего уровня. Хороший дизайн полезен для небольших проектов и незаменим для больших проектов.

Дизайн также характеризуется многочисленными сложностями, которые описаны в этом разделе.

Дизайн - Это Злая Задача

Хорст Риттел и Мелвин Уэббер определили “злую” задачу как задачу, которую можно четко определить, только решив ее или решив ее часть (1973). Этот парадокс подразумевает, по сути, что вы должны “решить” задачу один раз, чтобы четко определить ее, а затем решить ее снова, чтобы создать работающее решение. Этот процесс был основополагающим в разработке программного обеспечения на протяжении десятилетий (Питерс и Трипп, 1976).

В моей части света ярким примером такой злой задачи был проект оригинального моста Такома-Нэрроуз. В то время, когда мост был построен, главной задачей при проектировании моста было то, чтобы он был достаточно прочным, чтобы выдерживать запланированную нагрузку. В случае с мостом Такома-Нэрроуз ветер создал неожиданную гармоническую рябь из стороны в сторону. В один ветреный день 1940 года волна неконтролируемо росла, пока мост не рухнул, как показано на рисунке 5-1.

Это хороший пример злой задачи, потому что до тех пор, пока мост не рухнул, его инженеры не знали, что аэродинамику нужно учитывать в такой степени. Только построив мост (решив проблему), они могли узнать о дополнительной задаче, что позволило им построить еще один мост, который все еще стоит.

5.1 Проблемы проектирования 75

Рисунок 5-1 Мост Такома—Нарроуз - пример злой задачи.

Одно из главных различий между программами, которые вы разрабатываете в школе, и теми, которые вы разрабатываете как профессионал, заключается в том, что задачи проектирования, решаемые школьными программами, редко, если вообще когда-либо, являются злыми. Задания по программированию в школе разработаны таким образом, чтобы вы двигались по прямой от начала до конца. Вы, вероятно, захотите облить грязью и обсыпать перьями учителя, который дал вам задание по программированию, затем изменил задание, как только вы закончили дизайн, а затем снова изменил его, как раз когда вы собирались сдать завершенную программу. Но сам этот процесс является повседневной реальностью в профессиональном программировании.

Дизайн - Это Небрежный Процесс (Даже Если он Дает Чистый Результат).

Готовый дизайн программного обеспечения должен выглядеть хорошо организованным и чистым, но процесс, используемый для разработки дизайна, далеко не так аккуратен, как конечный результат.

Дизайн небрежен, потому что вы делаете много неверных шагов и заходите во многие тупики — вы совершаете много ошибок. Действительно, делать ошибки — это смысл дизайна - дешевле делать ошибки и исправлять дизайн, чем допускать те же ошибки, распознавать их после кодирования и исправлять полноразмерный код. Дизайн небрежен, потому что хорошее решение часто лишь незначительно отличается от плохого.

Дизайн также небрежен, потому что трудно понять, когда ваш дизайн “достаточно хорош”. Сколько детализации достаточно? Какой объем дизайна должен быть выполнен с использованием формальных обозначений дизайна, и сколько еще нужно сделать на клавиатуре? Когда ты закончишь? Поскольку дизайн открыт, наиболее распространенный ответ на этот вопрос - “Когда у вас заканчивается время”.

Дизайн - Это Компромиссы и Приоритеты

В идеальном мире каждая система могла бы работать мгновенно, потреблять нулевое пространство для хранения, использовать нулевую пропускную способность сети, никогда не содержать ошибок и ничего не стоить для создания. В реальном мире ключевая часть работы дизайнера заключается в том, чтобы взвесить конкурирующие характеристики дизайна и найти баланс между этими характеристиками. Если быстрая скорость отклика важнее, чем минимизация времени разработки, дизайнер выберет один дизайн. Если минимизация времени разработки более важна, хороший дизайнер разработает другой дизайн.

Дизайн Предполагает Ограничения

Смысл дизайна отчасти в том, чтобы создавать возможности, а отчасти в том, чтобы ограничивать возможности. Если бы у людей было бесконечное время, ресурсы и пространство для строительства физических сооружений, вы бы увидели невероятные огромные здания с комнатой для каждой туфли и сотнями комнат. Вот какое может выйти программное обеспечение без намеренно налагаемых ограничений. Ограничения ресурсов для строительства зданий вынуждают пойти на упрощение решения, что в конечном итоге улучшает его. Цель при разработке программного обеспечения та же.

Дизайн Недетерминирован

Если вы отправите трех человек разрабатывать дизайн одной и той же программы, они легко могут вернуться с тремя совершенно разными дизайнами, каждый из которых может быть вполне приемлемым. Может быть несколько способов снять шкуру с кошки, но обычно существуют дюжины способов разработать компьютерную программу.

Дизайн - Это Эвристический Процесс

Поскольку дизайн недетерминирован, методы проектирования, как правило, являются эвристическими — “эмпирическими правилами” или “вещами, которые иногда срабатывают”, — а не повторяющимися процессами, которые гарантированно дают предсказуемые результаты. Дизайн предполагает метод проб и ошибок. A инструмент или техника дизайна, которые хорошо работали на одной работе или на одном аспекте работы, могут не сработать так же хорошо на следующем проекте. Ни один инструмент не подходит для всего.

Дизайн Является Возникаемым

Чистый способ обобщить эти атрибуты дизайна - сказать, что дизайн является “возникающим”. Проекты не возникают полностью сформированными непосредственно в чьем-то мозгу. Они развиваются и совершенствуются с помощью обзоров дизайна, неофициальных обсуждений, опыта написания самого кода и опыта пересмотра кода.

Практически все системы претерпевают определенные изменения в дизайне во время их первоначальной разработки, а затем они обычно изменяются в большей степени по мере их расширения в более поздние версии. Степень, в которой изменения являются выгодными или приемлемыми, зависит от характера создаваемого программного обе спечения.

5.2 Ключевые Концепции Дизайна

Хороший дизайн зависит от понимания нескольких ключевых концепций. В этом разделе обсуждается роль сложности, желательных характеристик конструкций и уровней проектирования.

Основной Технический Императив Программного обеспечения: Управление сложностью

Чтобы понять важность управления сложностью, полезно обратиться к эпохальной статье Фреда Брукса “Нет серебряных пуль: сущность и случайности разработки программного обеспечения” (1987).

Случайные и существенные трудности

Брукс утверждает, что разработка программного обеспечения затрудняется из—за двух разных классов проблем - существенных и случайных. Говоря об этих двух терминах, Брукс опирается на философскую традицию, восходящую к Аристотелю. В философии существенные свойства - это свойства, которыми должна обладать вещь, чтобы называться этой вещью. Автомобиль должен иметь двигатель, колеса и двери, чтобы быть автомобилем. Если у него нет ни одного из этих существенных свойств, на самом деле это уже не автомобиль.

Случайные свойства - это свойства, которыми вещь просто обладает, свойства, которые на самом деле не влияют на то, является ли вещь тем, что она есть. Автомобиль может иметь V8, 4-цилиндровый двигатель с турбонаддувом или какой-либо другой двигатель и быть автомобилем независимо от этой детали. У автомобиля может быть две двери или четыре; у него могут быть тонкие колеса или магнитные колеса. Все эти детали являются случайными свойствами. Вы также можете думать о случайных свойствах как о инцидентных, дискреционных, необязательных и случайных.

Брукс отмечает, что основные случайные трудности в программном обеспечении были устранены давным-давно. Например, случайные трудности, связанные с неуклюжим синтаксисом языка, были в значительной степени устранены при переходе от языка ассемблера к языкам третьего поколения, и с тех пор их значение постепенно снижалось. Случайные трудности, связанные с неинтерактивными компьютерами, были устранены, когда операционные системы с разделением времени заменили системы пакетного режима. Интегрированные среды программирования также устранили неэффективность в работе по программированию, возникающую из-за плохо работающих вместе инструментов.

Брукс утверждает, что прогресс в решении остающихся существенных проблем программного обеспечения неизбежно будет медленнее. Причина в том, что по своей сути разработка программного обеспечения состоит из проработки всех деталей очень сложного, взаимосвязанного набора концепций. Существенные трудности возникают из-за необходимости взаимодействия со сложным, беспорядочным реальным миром; точного и полного определения зависимостей и случаев исключения; разработки решений, которые не могут быть только приблизительно правильными, но которые должны быть абсолютно правильными; и так далее. Даже если бы мы могли изобрести язык программирования, который использовал бы ту же терминологию, что и реальная задача, которую мы пытаемся решить, программирование все равно было бы трудным из-за сложности точного определения того, как работает реальный мир. По мере того, как программное обеспечение решает все более масштабные задачи реального мира, взаимодействия между сущностями реального мира становятся все более сложными, и это, в свою очередь, увеличивает существенную сложность программных решений.

Корнем всех этих существенных трудностей является сложность — как случайная, так и существенная.

Важность управления сложностью

Когда отчёты программных проектов сообщают о причинах сбоя проекта, они редко определяют технические причины в качестве основных причин сбоя проекта. Проекты чаще всего терпят неудачу из-за неудовлетворительных требований, плохого планирования или плохого управления. Но когда проекты терпят неудачу по причинам, которые в основном носят технический характер, причиной часто является неконтролируемая сложность. Программному обеспечению позволено становиться настолько сложным, что никто на самом деле не знает, что оно делает. Когда проект достигает точки, когда никто полностью не понимает, какое влияние изменения кода в одной области окажут на другие области, прогресс останавливается.

Управление сложностью - самая важная техническая тема в разработке программного обеспечения. На мой взгляд, это настолько важно, что основным техническим императивом программного обеспечения должно быть управление сложностью.

Сложность не является новой особенностью разработки программного обеспечения. Пионер вычислительной техники Эдсгер Дейкстра указал, что вычислительная техника - единственная профессия, в которой один разум обязан охватывать расстояние от бита до нескольких сотен мегабайт, соотношение 1 к 109, или девять порядков величины (Дейкстра 1989). Это гигантское соотношение ошеломляет. Дейкстра выразил это так: “По сравнению с таким количеством семантических уровней средняя математическая теория почти плоская. Пробуждая потребность в глубоких концептуальных иерархиях, автоматический компьютер ставит перед нами радикально новый интеллектуальный вызов, не имеющий прецедента в нашей истории”. Конечно, программное обеспечение стало еще более сложным с 1989 года, и соотношение Дейкстры 1 к 109 сегодня может быть больше похоже на 1 к 1015.

Дейкстра указал, что ни у кого череп на самом деле не настолько велик, чтобы вместить современную компьютерную программу (Дейкстра 1972), а это значит, что мы, разработчики программного обеспечения, не должны пытаться втиснуть в наш череп сразу целые программы; мы должны попытаться организовать наши программы таким образом, чтобы мы можете спокойно сосредоточиться на одной его части за раз. Цель состоит в том, чтобы свести к минимуму объем программы, о которой вам приходится думать в любой момент времени. Вы можете подумать об этом как о ментальном жонглировании — чем больше ментальных шаров программа требует, чтобы вы держали в воздухе одновременно, тем больше вероятность, что вы уроните один из шаров, что приведет к ошибке дизайна или кодирования.

На уровне архитектуры программного обеспечения сложность проблемы снижается за счет разделения системы на подсистемы. Людям легче понять несколько простых фрагментов информации, чем один сложный фрагмент. Цель всех методов разработки программного обеспечения состоит в том, чтобы разбить сложную задачу на простые части. Чем более независимыми являются подсистемы, тем безопаснее вы можете сосредоточиться на одной части сложности за раз. Тщательно определенные объекты разделяют заботы, чтобы вы могли сосредоточиться на чем-то одном за раз. Пакеты обеспечивают то же преимущество на более высоком уровне агрегирования.

Сокращение рутины помогает снизить вашу умственную нагрузку. Написание программ с точки зрения проблемной области, а не с точки зрения низкоуровневых деталей реализации, и работа на самом высоком уровне абстракции снижают нагрузку на ваш мозг.

Суть в том, что программисты, которые компенсируют присущие человеку ограничения, пишут код, который легче понять им самим и другим и в котором меньше ошибок.

Как атаковать сложность

Чрезмерно дорогостоящие и неэффективные проекты возникают из трех источников:

■ Комплексное решение простой задачи

■ Простое, неправильное решение сложной задачи

■ Неадекватное, сложное решение сложной задачи

Как отметил Дейкстра, современное программное обеспечение по своей сути сложное, и как бы вы ни старались, в конечном итоге вы столкнетесь с некоторым уровнем сложности, присущим задаче реального мира. Это предполагает двусторонний подход к управлению сложностью:

■ Сведите к минимуму количество существенных сложностей, с которыми приходится сталкиваться мозгу любого человека в любой момент времени.

■ Не допускайте ненужного увеличения случайной сложности.

Как только вы поймете, что все другие технические цели в программном обеспечении вторичны по отношению к сложности, создаваемой человеком, многие конструктивные соображения становятся понятными.

Желательные характеристики дизайна

Высококачественный дизайн имеет несколько общих характеристик. Если бы вы могли достичь всех этих целей, ваш дизайн был бы действительно очень хорош. Некоторые цели противоречат другим целям, но в этом и заключается сложность дизайна — создать хороший набор компромиссов между конкурирующими целями. Некоторые характеристики качества проектирования также являются характеристиками хорошей программы: надежность, производительность и так далее. Другие - это внутренние характеристики дизайна.

Вот список характеристик внутреннего дизайна:

Минимальная сложность Основной целью проектирования должно быть сведение к минимуму сложности по всем только что описанным причинам. Избегайте создания “умных” дизайнов. Хитроумные конструкции обычно трудно понять. Вместо этого создавайте “простые” и “понятные” проекты. Если ваш дизайн не позволяет вам безопасно игнорировать большинство других частей программы, когда вы погружены в одну конкретную часть, дизайн не выполняет свою работу.

Простота обслуживания Простота обслуживания означает дизайн для программиста технического обслуживания. Постоянно представляйте себе вопросы, которые задал бы программист по техническому обслуживанию о коде, который вы пишете. Думайте о программисте технического обслуживания как о своей аудитории, а затем спроектируйте систему так, чтобы она не требовала объяснений.

Слабая связь Слабая связь означает проектирование таким образом, чтобы свести к минимуму соединения между различными частями программы. Используйте принципы хорошей абстракции в интерфейсах классов, инкапсуляции и скрытии информации, чтобы создавать классы с как можно меньшим количеством взаимосвязей. Минимальная подключенность сводит к минимуму работу во время интеграции, тестирования и обслуживания.

Расширяемость Расширяемость означает, что вы можете улучшить систему, не нанося ущерба базовой структуре. Вы можете изменить часть системы, не затрагивая другие части. Наиболее вероятные изменения наносят системе наименьшую травму.

Возможность повторного использования Возможность повторного использования означает проектирование системы таким образом, чтобы вы могли повторно использовать ее части в других системах.

Высокий уровень вовлеченности Высокий уровень вовлеченности означает наличие большого количества классов, использующих данный класс. Высокая вентиляция подразумевает, что система была разработана таким образом, чтобы эффективно использовать классы полезности на более низких уровнях системы.

Уровни дизайна 20k

Проектирование требуется на нескольких различных уровнях детализации в программной системе. Некоторые методы проектирования применимы на всех уровнях, а некоторые - только на одном или двух. На рисунке 5-2 показаны уровни.

Уровень 1: Программная система

Первый уровень - это вся система. Некоторые программисты сразу переходят с системного уровня к разработке классов, но обычно полезно продумывать комбинации классов более высокого уровня, такие как подсистемы или пакеты.

Уровень 2: Разделение на подсистемы или Пакеты

Основным продуктом проектирования на этом уровне является идентификация всех основных подсистем. Подсистемы могут быть большими: база данных, пользовательский интерфейс, бизнес-правила, интерпретатор команд,

механизм отчетов и так далее. Основная деятельность по проектированию на этом уровне заключается в принятии решения о том, как разделить программу на основные подсистемы, и определении того, как каждой подсистеме разрешено использовать каждую другую подсистему. Разделение на этом уровне обычно требуется для любого проекта, который занимает больше времени, чем несколько недель. Внутри каждой подсистемы могут использоваться различные методы проектирования — выбор подхода, который наилучшим образом соответствует каждой части системы. На рисунке 5- 2 дизайн на этом уровне отмечен знаком 2.

Особое значение на этом уровне имеют правила о том, как различные подсистемы могут взаимодействовать. Если все подсистемы могут взаимодействовать со всеми другими подсистемами, вы теряете преимущество их разделения вообще. Сделайте каждую подсистему значимой, ограничив коммуникации.

Предположим, например, что вы определяете систему с шестью подсистемами, как показано на рис. 5-3. Когда нет никаких правил, в игру вступает второй закон термодинамики, и энтропия системы будет увеличиваться. Одним из способов увеличения энтропии является то, что без каких-либо ограничений на коммуникации между подсистемами связь будет происходить неограниченным образом, как показано на рис. 5-4.

Рисунок 5-3 Пример системы с шестью подсистемами.

Рисунок 5-4 Пример того, что происходит при отсутствии ограничений на межсистемные коммуникации.

Как вы можете видеть, каждая подсистема в конечном итоге напрямую взаимодействует со всеми другими подсистемами, что вызывает некоторые важные вопросы:

■ Сколько различных частей системы разработчику нужно хотя бы немного понять, чтобы что-то изменить в графической подсистеме?

■ Что происходит, когда вы пытаетесь использовать бизнес-правила в другой системе?

■ Что происходит, когда вы хотите внедрить в систему новый пользовательский интерфейс, возможно, пользовательский интерфейс командной строки для целей тестирования?

■ Что происходит, когда вы хотите разместить хранилище данных на удаленном компьютере?

Вы можете думать о линиях между подсистемами как о шлангах, по которым течет вода. Если вы хотите протянуть руку и вытащить подсистему, к этой подсистеме будет прикреплено несколько шлангов. Чем больше шлангов вам придется отсоединять и подсоединять снова, тем больше вы промокнете. Вы хотите спроектировать свою систему так, чтобы, если вы вытащите подсистему для использования в другом месте, у вас не было много шлангов для повторного подключения, и эти шланги будут легко подключаться.

При предусмотрительности все эти проблемы могут быть решены небольшой дополнительной работой. Разрешить связь между подсистемами только на основе “необходимости знать” - и это должно быть веской причиной. Если вы сомневаетесь, проще ограничить связь на ранней стадии и ослабить ее позже, чем ослабить ее на ранней стадии, а затем попытаться сузить ее после того, как вы закодировали несколько сотен межсистемных вызовов. На рисунке 5-5 показано, как несколько рекомендаций по коммуникации могут изменить систему, изображенную на рисунке 5-4.

Рис. 5-5 С помощью нескольких правил взаимодействия вы можете значительно упростить взаимодействие подсистем.

Чтобы связи было легко понять и поддерживать, ошибитесь в сторону простых межсистемных отношений. Самая простая взаимосвязь заключается в том, чтобы одна подсистема вызывала подпрограммы в другой. Более сложная взаимосвязь заключается в том, чтобы одна подсистема содержала классы из другой. Наиболее сложная взаимосвязь заключается в том, что классы в одной подсистеме наследуются от классов в другой.

Хорошее общее правило состоит в том, что диаграмма системного уровня, подобная рис. 5-5, должна быть ациклическим графом. Другими словами, программа не должна содержать никаких циклических связей, в которых класс A использует класс B, класс B использует класс C, а класс C использует класс A.

В больших программах и семействах программ дизайн на уровне подсистемы имеет значение. Если вы считаете, что ваша программа достаточно мала, чтобы пропустить проектирование на уровне подсистемы, по крайней мере, примите сознательное решение пропустить этот уровень проектирования.

Общие подсистемы Некоторые виды подсистем появляются снова и снова в разных системах. Вот некоторые из обычных подозреваемых.

Бизнес-правила Бизнес-правила - это законы, правила, политики и процедуры, которые вы кодируете в компьютерную систему. Если вы пишете систему расчета заработной платы, вы можете закодировать правила Налогового управления США о количестве допустимых удержаний и предполагаемой налоговой ставке. Дополнительные правила для системы начисления заработной платы могут вытекать из профсоюзного договора, в котором указываются ставки сверхурочной работы, оплата отпусков и отпускных и так далее. Если вы пишете программу для определения ставок автострахования, правила могут исходить из правительственных постановлений о требуемом страховании ответственности, таблиц актуарных ставок или ограничений на страхование

Пользовательский интерфейс Создайте подсистему для изоляции компонентов пользовательского интерфейса, чтобы пользовательский интерфейс мог развиваться без ущерба для остальной части программы. В большинстве случаев подсистема пользовательского интерфейса использует несколько подчиненных подсистем или классов для интерфейса GUI, интерфейса командной строки, операций с меню, управления окнами, справочной системы и так далее.

Доступ к базе данных Вы можете скрыть детали реализации доступа к базе данных, чтобы большая часть программы не беспокоилась о запутанных деталях манипулирования низкоуровневыми структурами и могла обрабатывать данные с точки зрения того, как они используются на уровне бизнес-задач. Подсистемы, скрывающие детали реализации, обеспечивают ценный уровень абстракции, который снижает сложность программы. Они централизуют операции с базой данных в одном месте и снижают вероятность ошибок при работе с данными. Они позволяют легко изменять структуру проектирования базы данных без изменения большей части программы.

Системные зависимости Упаковывают зависимости операционной системы в подсистему по той же причине, по которой вы упаковываете зависимости от оборудования. Например, если вы разрабатываете программу для Microsoft Windows, зачем ограничивать себя средой Windows? Изолируйте вызовы Windows в подсистеме интерфейса Windows. Если позже вы захотите перенести свою программу на Mac OS или Linux, все, что вам нужно будет изменить, - это подсистему интерфейса. Подсистема интерфейса может быть слишком обширной, чтобы вы могли реализовать ее самостоятельно, но такие подсистемы легко доступны в любой из нескольких коммерческих библиотек кода.

Уровень 3: Разделение на классы

Дизайн на этом уровне включает в себя идентификацию всех классов в системе. Например, подсистема интерфейса базы данных может быть дополнительно разделена на классы доступа к данным и классы инфраструктуры сохранения, а также метаданные базы данных. На рисунке 5-2, Уровень 3, показано, как одна из подсистем уровня 2 может быть разделена на классы, и это подразумевает, что три другие подсистемы, показанные на уровне 2, также разбиты на классы.

Подробные сведения о способах взаимодействия каждого класса с остальной частью системы также указываются по мере указания классов. В частности, определен интерфейс класса. В целом, основная работа по проектированию на этом уровне заключается в том, чтобы убедиться, что все подсистемы были разложены на достаточно детальный уровень, чтобы вы могли реализовать их части как отдельные классы.

Разделение подсистем на классы обычно требуется в любом проекте, который занимает больше времени, чем несколько дней. Если проект большой, разделение явно отличается от разделения программы уровня 2. Если проект очень мал, вы можете перейти непосредственно от представления всей системы уровня 1 к представлению классов уровня 3.

Классы против Объектов Ключевым понятием в объектно-ориентированном проектировании является разграничение объектов и классов. Объект - это любая конкретная сущность, существующая в вашей программе во время выполнения. Класс - это статическая вещь, на которую вы смотрите в списке программ. Объект - это динамическая вещь с определенными значениями и атрибутами, которые вы видите при запуске программы. Например, вы могли бы объявить класс Person с атрибутами имени, возраста, пола и так далее. Во время выполнения у вас будут объекты nancy, hank, diane, tony и так далее, то есть конкретные экземпляры класса. Если вы знакомы с терминами базы данных, это то же самое, что различие между “схемой” и “экземпляром”. Вы могли бы думать о классе как о резаке для печенья, а об объекте - как о файле cookie. В этой книге термины используются неофициально и, как правило, относятся к классам и объектам более или менее взаимозаменяемо.