Навыки элегантного программирования. Часть 1.

Михаил Зускин | 04.10.2010

В самом большом за всю мою карьеру проекте участвовало лишь 3,5 разработчика (менеджер проекта, два штатных разработчика, включая меня, и еще один парень, приходивший на помощь в случае авралов). Ключевым фактором столь высокой эффективности было наличие хорошей библиотеки базовых компонентов (framework). Благодаря ей, мы в основном думали о том, «что» создать, а не «как». Другими словами, мы фокусировались на бизнес-логике продукта, а написание «технического» кода занимало лишь малую часть нашего времени. Вторая причина: приложение создавалось в соответствии с правилами, представленными в данной статье (бо́льшая их часть была придумана не мной). С другой стороны, я принимал участие в гораздо более простых с точки зрения пользователя проектах, в которых было вовлечено намного больше сотрудников. Целые IT-департаменты бились с завалами безнадежно запутанного кода. Что вы скажете о хранимой процедуре в 2800 строк? Это больше исходного кода первой версии Unix! Я очень сильно надеюсь, что читатели этой статьи НЕ найдут ее полезной, поскольку они хорошо знают описанные принципы и действуют в соответствии с ними. Мои примеры представлены для C# и PL/SQL, но они также будут понятны разработчикам, пишущим на других языках программирования.

1. Структура кода

Отступы

Не увлекайтесь излишними отступами. Порой более 3 табуляций – это «красная лампочка», означающая неправильную организацию структуры подпрограмм. Я ленив и не люблю напрягаться, поэтому в моих методах вы не увидите более 3 табуляций (хорошо, хорошо, иногда увидите, но по крайней мере я всегда стремлюсь этого избегать). И, конечно же, в моем коде вы не найдете длинных лесенок из фигурных скобок и структур ‘END IF’!

Подпрограммы

Создавайте подпрограмму (метод, функцию, событие, процедуру и т.п.) для каждой логической задачи. Другими словами, разбивайте длинный скрипт на более короткие (см. Разделение ответственности). В Oracle: создавайте пакет (package) в ситуациях, когда плохой программист создал бы одну длинную автономную хранимую процедуру. В СУБД без пакетов (скажем, MS SQL Server и Sybase ASE) применяйте префиксы в названиях процедур (сразу после префикса, добавляемого в соответствии с соглашениями по присвоению имен). Тогда все процедуры, принадлежащие к одной логической задаче, будут сгруппированы при сортировке (к примеру, p_emp… для процедур в компоненте Employee Management). Нет ничего страшного в наличии большого количества методов/процедур, даже если их десятки. Разработчик всегда сможет открыть любой из них и сконцентрироваться только на нем, автоматически выбирая нужный уровень абстракции. В то же время, очень трудно понять, как работает ОДИН замысловатый скрипт длиной с рулон туалетной бумаги. Сложность не столько в прокрутке, сколько в смешении разных уровней абстракции в одном методе. Основная программа должна вызывать *хорошо именованные* подпрограммы (а они – следующие), и тогда разработчик сможет легко «путешествовать» вверх и вниз между различными уровнями абстракции по иерархической структуре любой глубины. Скрипты длиннее 2-3 экранов так же настораживают, если только скрипт действительно не выполняет большой кусок «черной работы», которую нельзя (или не стоит) разбивать на части. Это правило особенно полезно в циклах: если код, выполняемый в цикле, достаточно большой, поместите его в метод! Разработчик сможет открыть этот метод и сконцентрироваться на построчной обработке, а при изучении вызывающего скрипта, содержащего операторы цикла, он подумает о логике более высокого уровня. Обратите внимание: сложности с многоэкранными скриптами обычно соседствуют с безобразными избыточными отступами, обсуждавшимися ранее. Прошу прощения, если этот параграф был скучным. Трудно поверить, но представленные советы не очевидны для некоторых разработчиков!

Маленькие блоки кода

Старайтесь не раздувать блоки кода. Располагайте открывающие и закрывающие фигурные скобки (или конструкции ‘BEGIN’…’END’, ‘IF’…’END IF’) недалеко друг от друга. Очень здорово размещать их в пределах одного экрана. Иногда я исследую процедуры других ребят и прикладываю палец к экрану, помечая расположение ‘BEGIN’ и нажимая клавишу PageDown много раз (действительно, много!) в поисках соответствующего ‘END’. *Обычно* он находится, но много бумажных салфеток надо потратить чтобы вытереть экран! К счастью, некоторые инструменты (скажем, Visual Studio) автоматически выделяют начало и конец блока, но тем не менее…

Проверка параметров

Внутри метода отделяйте проверку параметров от бизнес-логики:

*** ПЛОХОЙ код: ***

```
if ([condition 1])
{
        if ([condition 2])
        {
                [code fragment]
        }
        else
        {
                return 100;
        }
}
else
{
        return 0;
}

return 1;
```

*** ХОРОШИЙ код: ***

```
if (![condition 1]) return 0;
if (![condition 2]) return 100;

[code fragment] // нет отступа, поэтому не страшно, если у фрагмента есть свои отступы

return 1;
```

Код после проверок

Если *большой* фрагмент кода выполняется после нескольких проверок и располагается после нескольких ‘if’, разместите этот фрагмент и проверки в отдельном методе и завершайте его выполнение сразу после неудачных проверок. Это не только поможет избежать избыточных отступов (что важно само по себе), но также сообщит, что весь алгоритм будет выполнен только при прохождении всех проверок. Например, если первая строка в методе – *«if (Page.IsPostBack) return;»*, и метод длиннее экрана, то читатель сразу поймет, что весь код будет выполнен только в случае первой загрузки страницы (не обновления). Если же в коде присутствует блок *«if (!Page.IsPostBack) { [фрагмент на много экранов] }»*, то придется прокручивать вниз, чтобы узнать о наличии или отсутствии кода после блока ‘if’, который выполняется во всех случаях, включая обновления страниц. Вот как можно преобразовать чудовищный код в элегантный:

*** ПЛОХОЙ код: ***

```
if (this.Validate1())
{
        if (this.Validate2())
        {
                if (this.Validate3())
                {
                        [фрагмент кода с собственными уровнями отступов]
                }
        }
}
```

*** ХОРОШИЙ код (внутри созданного метода): ***

```
if (!this.Validate1()) return;
if (!this.Validate2()) return;
if (!this.Validate3()) return;

[фрагмент кода с собственными уровнями отступов]
```

Вы можете спросить: как насчет правила «единственной точки выхода»? Я не хочу его здесь обсуждать, но такая идея создает больше проблем, чем решает. Я согласен с Дейкстра, убежденным противником единственной точки выхода. Как бы то ни было, здесь говорится о предварительных проверках. Если они успешно пройдены, и началось выполнение «интересного» кода («мясо» метода), то с этого момента можно при желании следовать правилу «единственной точки выхода».

Если фрагмент кода, расположенный после многих проверок, не очень длинный, и вы не хотите создавать для него специальный метод, следует применить «флаг-метод» (только без большого отступа!):

*** ХОРОШИЙ код (без отдельного метода): ***

```
boolean ok;

ok = this.Validate1();
if (ok) ok = this.Validate2();
if (ok) ok = this.Validate3();
if (ok)
{
        [фрагмент кода с собственными уровнями отступов]
}
```

Циклы с ‘continue’

Операторы ‘continue’ в циклах помогают значительно упростить код. Если в данном месте кода не нужно выполнять никаких действий, немедленно переходите к следующей итерации (банальная идея: зачем выполнять ненужные шаги до конца цикла?):

*** ПЛОХОЙ код: ***

```
while ([condition])
{
        if (this.Validate1())
        {
                if (this.Validate2())
                {
                        if (this.Validate3())
                        {
                                [фрагмент кода с собственными уровнями отступов]
                        }
                }
        }
}
```

*** ХОРОШИЙ код: ***

```
while ([condition])
{
        if (!this.Validate1()) continue;
        if (!this.Validate2()) continue;
        if (!this.Validate3()) continue;

        [фрагмент кода с собственными уровнями отступов]
}
```

Никаких дубликатов условий цикла

В цикле, который может быть ни разу не выполнен, не пишите одни и те же строки кода дважды (до цикла и внутри него). Почему так, если в большинстве книг условие встречается дважды? Ответ прост: зачем набирать что-либо дважды, если это можно написать один раз и достичь той же цели?

*** ПЛОХОЙ код: ***

```
FETCH emp_cursor INTO v_row;
WHILE emp_cursor%FOUND LOOP
        [DO something]
        FETCH emp_cursor INTO v_row; -- deja vu! :-)
END LOOP;
```

Внешний цикл с ‘break’ (или ‘exit when’ в PL/SQL) будет очень элегантным решением в подобных случаях:

*** ХОРОШИЙ код: ***

```
LOOP
        FETCH emp_cursor INTO v_row;
        EXIT WHEN emp_cursor%NOTFOUND;
```

```
        [DO something]
END LOOP;
```

Технический код и бизнес-код

«… и отделил Бог свет от тьмы»
Первая книга Моисея

Не смешивайте обработку ошибок (полностью технический код) и бизнес-логику. Разработчикам, читающим скрипт, стоит фокусироваться только на бизнес-аспектах. Они не должны исследовать, где располагается код обработки ошибок (эта часть не представляет интереса и пропускается автоматически) и «интересная» часть, «мясо» программы. В следующем примере метод GetOrdersCount() возвращает 0 (заказы не найдено) или положительное число найденных заказов. В этих случаях должна применяться разная бизнес-логика, но метод также возвращает -1, что означает неудачу при получении данных (ошибку). Последнее – довольно скучный технический материал, который нужно отделить от бизнес-кода. При рассмотрении следующих примеров не забудьте, что фрагменты кода, представленные здесь парой слов в квадратных скобках, на практике могут быть огромными кусками кода!

*** ПЛОХОЙ код: ***

```
ordersCount = this.GetOrdersCount();
if (ordersCount == -1) // ошибка :-O
{
        [фрагмент кода для банальной обработки ошибки]
        return -1;
}
else if (ordersCount == 0) // заказы не найдены... :-(
{
        [фрагмент кода для бизнес-сценария "Заказы не найдены"]
}
else /* заказы найдены!!! */
{
        [фрагмент кода для бизнес-сценария "Заказ(ы) найден(ы)"]
}
```

*** ОЧЕНЬ ПЛОХОЙ (!!!) код: ***

```
ordersCount = this.GetOrdersCount();
if (ordersCount == -1) // ошибка :-O
{
        [фрагмент кода для банальной обработки ошибки]
        return -1;
}
else
{
        if (ordersCount == 0) // заказы не найдены... :-(
                {
                [фрагмент кода для бизнес-сценария "Заказов не найдено"]
                }
        else /* заказ(ы) найден(ы)!!! */
                {
                [фрагмент кода для бизнес-сценария "Заказ(ы) найден(ы)"]
                }
}
```

*** ХОРОШИЙ код: ***

```
ordersCount = this.GetOrdersCount();
// Следующий блок 'if' - стандартный технический блок, поэтому читатель скрипта пропустит его автоматически:
if (ordersCount == -1) // error :-O
{
        [фрагмент кода для банальной обработки ошибки]
        return -1;
}

// Здесь начинается "интересная" бизнес-логика:
if (ordersCount == 0) // заказы не найдены... :-(
{
        [фрагмент кода для бизнес-сценария "Заказов не найдено"]
}
else /* заказы найдены!!! */
{
        [фрагмент кода для бизнес-сценария "Заказ(ы) найден(ы)"]
}
```

К счастью, в наши дни, когда вместо возвращения статуса «успех/неудача» мы обычно применяем исключения, этот совет можно рассматривать как устаревший при обсуждении обработки ошибок.

Избегайте чересчур вложенных выражений

Хорошие разработчики разбивают один длинный метод на несколько коротких подметодов. Тот же принцип работает для сложного, излишне вложенного выражения – его следует разбить на автономные, простые строки. Единственная проблема: такой код не поразит воображение молодых программистов, поэтому они перестанут считать вас одним из лучших в мире разработчиков…

*** ПЛОХОЙ код: ***

```
countryName = this.GetCountryNameByCountryId(this.GetCountryIdByCityId(this.GetCityId(rowNum)));
```

*** ХОРОШИЙ код: ***

```
cityId = this.GetCityId(rowNum);
countryId = this.GetCountryIdByCityId(cityId);
countryName = this.GetCountryNameByCountryId(countryId);
```

Помимо улучшения читабельности кода, такой подход облегчает отладку. В нашем примере: если countryName не было заполнено как ожидалось, вы сразу увидите в отладчике, какой шаг создает проблему (без STEP IN). Кроме того, если вы хотите выполнить STEP IN GetCountryNameByCountryId в целях отладки, то вы будете вынуждены сначала зайти в (STEP IN) и выйти из (STEP OUT) каждого вложенного метода, начиная с GetCityId.

Оператор переключения вместо многочисленных OR

Оператор переключения (switch) помогает избегать многочисленных ‘||’ и создавать более красивый код в случаях, когда значение переменной сравнивается со множеством значений.

*** ПЛОХОЙ код: ***

```
if (month == Months.Jan || month == Months.Feb || month == Months.Apr || month == Months.Aug || month == Months.Sep)
{
        [фрагмент кода]
}
```

*** ХОРОШИЙ код: ***

```
switch (month)
{
        case Months.Jan:
        case Months.Feb:
        case Months.Apr:
        case Months.Aug:
        case Months.Sep:
                [фрагмент кода]
                break;
}
```

Короткие условия в операторах ‘if’

Не пишите слишком сложных выражений в операторах ‘if’. Вместо этого, храните их результат в булевой переменной с наглядным именем и применяйте эту переменную в описываемых операторах:

*** ПЛОХОЙ код: ***

```
if ((calculationMethod == CalculationMethods.Additive && additiveCalculationPassed) ||
        (calculationMethod == CalculationMethods.Ratio && ratioCalculationPassed))...
```

*** ХОРОШИЙ код: ***

```
boolean structuralChangeOccurred;

structuralChangeOccurred = (calculationMethod == CalculationMethods.Additive && additiveCalculationPassed) ||
                        (calculationMethod == CalculationMethods.Ratio && ratioCalculationPassed);

if (structuralChangeOccurred)...
```

Как видно из примера, плохой код говорит нам только КОГДА условие истинно, а хороший код говорит не только КОГДА, но и ПОЧЕМУ!!! Это настолько важно, что я с легкостью иду наперекор правилу, обязывающему нас уменьшать размер скриптов. Я не пытаюсь сокращать мои скрипты *любой ценой* и время от времени предпочитаю ДОБАВЛЯТЬ СТРОКИ, если они делают код более понятным и менее напоминающим кучу мусора. И, конечно, не забывайте, что в моем примере выражение довольно мало (этого достаточно чтобы донести идею). В реальной жизни внутри ‘if’ встречаются булевы выражения на полэкрана. Одно из них было настолько запутанным (классическая куча мусора!), что вместо невоспринимаемого выражения-монстра я не только применил бы булеву переменную как описано

выше, но и заполнял бы эту переменную пошагово с помощью более простых и менее запутывающих булевых выражений.

Тот же совет применим к методам: не вызывайте их внутри ‘if’, а храните возвращаемое ими значение в переменной. Однажды я нашел такую строку в коде:

```
if (this.DisplayCommentsWindow())...
```

Что возвращает этот метод? Есть идеи? Я открыл код метода и увидел, что он возвращает истину, если пользователь кликнул на кнопку OK в окне комментариев, открытом в методе, и ложь, если он кликнул CANCEL. Я написал бы этот метод так:

```
boolean bUserClickedOkInCommentsWindow; // В стиле Льва Толстого!

bUserClickedOkInCommentsWindow = this.DisplayCommentsWindow();
if (bUserClickedOkInCommentsWindow)... // это язык программирования или нормальный человеческий язык? :-)
```

Однако, хорошо именованный метод, возвращающий булево значение, можно напрямую применять в операторах ‘if’, если список параметров не слишком длинный. Можно заметить, что метод DisplayCommentsWindow() нельзя назвать bUserClickedOkInCommentsWindow(), потому что он служит для отображения окна комментариев. Но, возможно, лучшее решение в этом случае – применить аргумент OUTPUT, который *заставит* разработчиков создать переменную (!!!), передать ее методу и красиво задействовать в конструкции ‘if’:

```
this.DisplayCommentsWindow(out bUserClickedOkInCommentsWindow);
```

Присвоение значений переменным boolean

Если это поможет сократить объём кода, присваивайте булевским переменнам не ‘true’ и ‘false’, а результаты логических выражений:

*** ПЛОХОЙ код: ***

```
if (rowsRetrieved > 1)
	{
	multiRowsMode = true;
	}
else
	{
	multiRowsMode = false;
	}
```

*** ХОРОШИЙ код: ***

```
multiRowsMode = (rowsRetrieved > 1);
```

Обрабатывайте невозможные варианты (да, да, это возможно!)

Если ваш код обрабатывает несколько вариантов, скажем, различные статусы или режимы (обычно с применением конструкции ‘switch’), не забывайте, что в будущем список возможных вариантов может расти. И тогда вы (или другие разработчики) можете упустить обработку новых вариантов. Как этого избежать? Мы можем заставить код пожаловаться: «Разработчик, открой меня и ПОДУМАЙ, как обработать новый вариант!». Как заставить? С помощью двух простых вещей:

1. В конструкцию ‘switch’ добавляем новый ‘case’ для КАЖДОГО варианта, который не обрабатывается этим ‘switch’ (обычно они совсем не упоминаются людьми, не задумывающимся о будущем). Эти секции case не будут выполнять никаких действий, поэтому напишите комментарий вроде «ничего не делать», «нерелевантно» или «неприменимо» в их выполняемых частях. Это позволит разработчикам понять, что вы оставили их пустыми сознательно, а не по ошибке.

2. Добавьте секцию ‘default’, которая будет выдавать исключение.

Рассмотрим два следующих фрагмента. Здесь используются 4 статуса заказчика: Active, Pending, Inactive и Deleted, но в настоящее время в соответствии с бизнес-логикой обрабатываются только заказчики со статусами Active и Pending:

*** ПЛОХОЙ код (статусы Inactive и Deleted даже не упоминаются…): ***

```
switch (customerStatus)
{
	case CustomerStatuses.Active:
		[фрагмент кода, обрабатывающий активных заказчиков]
		break;
	case CustomerStatuses.Pending:
		[фрагмент кода, обрабатывающий потенциальных заказчиков]
```

```
                break;
}
```

*** ХОРОШИЙ код (статусы Inactive и Deleted упоминаются явным образом в специально созданных ветках): ***

```
switch (customerStatus)
{
        case CustomerStatuses.Active:
                [фрагмент кода, обрабатывающий активных заказчиков]
                break;
        case CustomerStatuses.Pending:
                [фрагмент кода, обрабатывающий потенциальных заказчиков]
                break;
        case CustomerStatuses.Inactive:
        case CustomerStatuses.Deleted:
                // do nothing
                break;
        default:
                throw new Exception ("Не определено поведение для заказчика со статусом " + customerStatus);
}
```

Таким образом, если спустя годы в бизнес-логику будет добавлен новый статус (например, ‘Deceased’, описанный константой CustomerStatuses.Deceased), фрагмент кода заставит разработчиков внести изменения в логику: если переменная customerStatus будет содержать значение CustomerStatuses.Deceased (которое, разумеется, не обрабатывается в ‘switch’), выполнение программы пойдет по руслу секции ‘default’, и будет выдано исключение. И случится это скорее всего на ранней стадии разработки или выполнения модульного теста! Но даже если исключение будет выдано в работающей системе, это будет лучше, чем долгая жизнь со скрытым багом. Возможно, новый вариант должен быть обработан особым образом, и вам необходимо решить (или обсудить с сотрудниками из бизнес-отдела), что с ним делать. Если новый статус не требует особой обработки, то просто добавьте его в компанию необрабатываемых кодов (как ещё один case «ничего не делать»). Я помню цепочку таких ошибок при сборке приложения для футбольной ассоциации (шесть или семь – это было забавно!) после добавления нового значения. Это позволило мне исправить возможные баги даже до модульного теста! Цепочка сообщений напоминала серью пенальти – и я выиграл, не пропустив ни одного мяча!

Давайте слегка изменим условия задачи. 4 статуса заказчика (Active, Pending, Inactive и Deleted) остаются, но теперь ВСЕ они обрабатываются. Статусы Inactive и Deleted обрабатываются одинаково, поэтому разработчик поместил код для обоих в секцию ‘default’:

*** ПЛОХОЙ код (статусы Inactive и Deleted обрабатываются в секции ‘default’, поэтому их названия даже не упоминаются…): ***

```
switch (customerStatus)
{
        case CustomerStatuses.Active:
                [фрагмент кода, обрабатывающий активных заказчиков]
                break;
        case CustomerStatuses.Pending:
                [фрагмент кода, обрабатывающий потенциальных заказчиков]
                break;
        default:
                [фрагмент кода, обрабатывающий неактивных и удаленных заказчиков]
}
```

*** ХОРОШИЙ код (статусы Inactive и Deleted упоминаются явным образом: ***

```
switch (customerStatus)
{
        case CustomerStatuses.Active:
                [фрагмент кода, обрабатывающий активных заказчиков]
                break;
        case CustomerStatuses.Pending:
                [фрагмент кода, обрабатывающий потенциальных заказчиков]
                break;
        case CustomerStatuses.Inactive:
        case CustomerStatuses.Deleted:
                [фрагмент кода, обрабатывающий неактивных и удаленных заказчиков]
                break;
        default:
                throw new Exception ("Не определено поведение для заказчика со статусом " + customerStatus);
}
```

Таким образом, подход «ХОРОШИЙ код» позволяет задействовать ‘default’ для исключений. Метод, в котором перечисляются ВСЕ возможные варианты вместо их неявной обработки в секции ‘default’, имеет еще одно не менее важное преимущество: с помощью Глобального Текстового Поиска можно найти ВСЕ места в приложении, где обрабатывается искомый вариант. Если вариант обрабатывается в секции ‘default’, то он не будет найден! Вы удивлены? Нет? Тогда почему во всех проектах, в которых мне довелось работать, применялся только «ПЛОХОЙ код»? Я

действительно не знаю… Вот пример: однажды статус ‘Inactive’ устаревает и перестает использоваться. Его нужно заменить двумя новыми статусами и обрабатывать различными способами в зависимости от ситуации. Вы должны найти все фрагменты в коде приложения, где присутствует статус, выражаемый CustomerStatuses.Inactive. Если ваш выбор – стиль «ХОРОШИЙ код», то вы легко их найдете. Но если вы любите размещать бизнес-логику в секции ‘default’, то вам придется нелегко – потребуется время на исследования с большой вероятностью того, что вы не найдете ВСЕ, что ищите.

У описанного метода есть еще одно дополнительное преимущество (очень важное, по моему мнению). Взглянув на конструкцию ‘switch’, вы видите всю картину, а не ее часть. Вам не нужно гадать, какие варианта попадут в ‘default’. Как говорится, зачастую полуправда хуже лжи…

Мы обсудили только операторы ‘switch’, но концепция также работает и для ‘if’. Обычно вы заменяете ‘if’ на ‘switch’, чтобы избежать ужасной кучи операторов ‘else if’:

*** ПЛОХОЙ код: ***

```
if (_currentEntity == Entities.Car)
{
    this.Retrieve(_carId);
}
else
{
    this.Retrieve(_bikeId);
}
```

*** ХОРОШИЙ код: ***

```
switch (_currentEntity)
{
        case Entities.Car:
                this.Retrieve(_carId);
                break;
        case Entities.Bike:
                this.Retrieve(_bikeId);
                break;
        default:
                throw new Exception ("Не определен вариант для сущности " + _currentEntity);
}
```

Не применяйте булевы флаги (включая CHAR-переменные со значениями вида ‘Y’ и ‘N’ или ’1′ и ’0′), если теоретически могут существовать более двух вариантов. Допустим, у вас есть один класс с автомобилями и велосипедами (потому что эти сущности обрабатываются схожим образом). Иногда вам понадобится знать, в каком из двух режимов создан объект (или экземпляр) – в режиме ‘автомобиль’ или ‘велосипед’. Тогда можно задуматься о булевом флаге _isCarMode, инициализируемом как истина или ложь в зависимости от режима). Флаг можно опросить так:

*** ПЛОХОЙ код: ***

```
if (_isCarMode)
{
    this.Retrieve(_carId);
}
else // Bike Mode
{
    this.Retrieve(_bikeId);
}
```

Наилучшее решение в такой ситуации: создать переменную с типом enum несмотря на то, что возможны лишь два варианта, с чем вроде бы прекрасно справляется булев тип. – см. предыдущий «ХОРОШИЙ код». Если в один прекрасный день вы захотите использовать обсуждаемый класс еще и для лодок, то просто добавьте Boat к ранее созданному перечисляемому типу в качестве нового возможного значения, инициализируйте этим значением _currentEntity, запустите приложение и следуйте своим собственным инструкциям, написанным месяцы или годы тому назад!

Не дублируйте логику

Методы с одинаковой или очень похожей функциональностью следует объединять в один общий метод. Если такие методы появляются в классах, унаследованных от одного и того же родителя, создайте общий метод в прародителе и вызывайте его из потомков. Если классы не унаследованы от одного родителя, создайте общий метод в третьем классе (даже если вам понадобится создать класс только ради одного метода!). Если вы ловите себя на размышлении «что лучше: продублировать код с помощью ‘копирования-вставки’ за 10 минут или разместить его в третьем месте (2 часа, включая тестирование)», немедленно остановитесь! НИКОГДА НЕ ДУМАЙТЕ ОБ ЭТОМ, даже в последние дни вашего участия в проекте. ПРОСТО РАЗМЕСТИТЕ КОД В ТРЕТЬЕМ МЕСТЕ и вызывайте его при необходимости.

Если вы все еще находитесь в сомнениях об использовании своего времени (на самом деле, это время компании, а не ваше), то спросите своего менеджера. А еще лучше хорошо сделать свою работу, а потом объяснить менеджеру, почему это заняло больше времени. Если менеджер понимает, насколько это более качественный подход к программированию, он оценит ваши усилия.

Как правило, методы, размещаемые в третьем месте в целях предотвратить дублирование, *очень похожи*, но *не обязательно абсолютно идентичны*. Тем не менее, объединяйте скрипты, насколько возможно, а вызывающие классы пусть поставляют данные, специфические для каждого них. Например, у нас есть два метода в разных классах (фрагменты 1 и 2 во втором классе точно такие же как соответствующие фрагменты 1 и 2 в первом классе, но DataObjects различаются):

*** ПЛОХОЙ код: ***

```
SomeMethod() первого класса:

[фрагмент 1]
_entityDescription = "Car";
[фрагмент 2]

SomeMethod() второго класса:

[фрагмент 1] // точно такой же как [фрагмент 1] в первом классе :-)
_entityDescription = "Bike"; // oooops, отличается от того же куска в первом классе... :-(
[фрагмент 2] // точно такой же как [фрагмент 2] в первом классе :-)
```

*** ХОРОШИЙ код: ***

```
SomeMethod(), размещенный в родительском классе:

[фрагмент 1]
_entityDescription = this.GetEntityDescription();
[фрагмент 2]
```

Таким образом, мы используем метод GetEntityDescription, чтобы справиться с различием в двух методах. GetEntityDescription создается в родительском классе как абстрактный метод и впоследствии должен быть определен в потомках, чтобы предоставить специфические описания сущностей. В первом потомке метод должен быть закодирован как *‘return «Car»;’*, а во втором – как *‘return «Bike»;’*. Если метод размещен в третьем классе (не входящем в иерархию наследования), то, как вариант, СПЕЦИФИЧНЫЕ (различные) данные могут быть переданы через параметр(ы) нового объединенного метода, поэтому фрагмент *‘_entityDescription = this.GetEntityDescription();’* в этом случае будет выглядеть так: *‘_entityDescription = aEntityDescription;’*, где ‘aEntityDescription’ есть параметр метода.

Наконец, есть еще один способ достичь той же цели. Мы можем определить переменную ‘_entityDescription’ экземпляра при инициализации экземпляра (например, в его конструкторе), но такой способ не всегда подходит. Наиболее важно вот что: СПЕЦИФИЧНЫЕ ДАННЫЕ ИЗ РАЗЛИЧНЫХ ЧАСТЕЙ ПРИЛОЖЕНИЯ НУЖНО ПЕРЕДАВАТЬ ОБЩЕМУ АЛГОРИТМУ, РЕАЛИЗОВАННОМУ ТОЛЬКО ОДИН РАЗ. В одном из проектов, в которых я участвовал по контракту, проблема с дублированием «почти такого же» кода была настолько серьезной, что я предложил провести рефакторинг всей подсистемы приложения, и мое предложение было принято. На эту задачу была потрачена неделя, и это была одна из наиболее интересных задач в моей карьере! В результате, мое резюме украсилось следующим прекрасным фрагментом:

Предложил и провел рефакторинг кода в существующей корпоративной системе отчетов, переместив дублирующуюся функциональность из классов-потомков в классы-родители. Облегчил поддержку и поиск неполадок, а добавление новых отчетов сделал более быстрым и простым.

Конечно, лучше потратить немного времени перед разработкой и подумать об организации классов вместо бездумного кодирования «в лоб», приводящему к интенсивной нагрузке на клавиши Ctrl+C и Ctrl+V.

Константы для [бессмысленных] кодов приложения

Избегайте жесткого кодирования чисел, если их смысл неочевиден. Используйте перечисления (enumerations) или даже простые константы! Хорошо, я могу понять, что означает *«if (invoiceYear > 2006)…»* или *«if (month == 11)…»*, но что вы захотите сказать программисту, который написал *«if (windowMode = 3)…»* или *«if (invoiceStatus = 8)…»*? Если по-прежнему не хотите применять их как enum, то по крайней мере оставьте комментарий: *«if (invoiceStatus = 8 /* closed */)…»*. Прошу прощения за столь банальный совет, но после того, что я видел на протяжении своей карьеры, я легко могу открыть компанию по частному сыску…

Мнемонические строки как коды приложения

Трудностей, описанных в предыдущем разделе, можно избежать, если коды приложения (скажем, статусы, режимы и т.п.) представляют собой текстовые, а не бессмысленные числовые значения (1, 2, 3…). Самоочевидные коды (отличная идея!) значительно облегчают отладку, чтение кода и исследование информации в таблицах баз данных! Когда ваш мозг

загружен поиском бага, значение переменной в отладчике или значение в таблице базы данных лучше увидеть как ‘CLOSED’ (вместо мистического числа 3), не правда ли? Мнемонические строки не должны быть длинными – VARCHAR(10) будет достаточно. При необходимости мы всегда сможем укоротить слова без ущерба для их понимания.

*** ПЛОХОЙ подход (вам нужно обращаться к таблице с каталогом кода для понимания числовых кодов): ***

```
Таблица ORDER:                                                  Таблица ORDER_STATUS (каталог кодов):

order_id        customer_id     order_status_code               code            description
123001          98765           3                               1               'Open'
123002          43210           1                               2               'Cancelled'
123003          12345           2                               3               'Closed'
```

*** ХОРОШИЙ подход (коды самоочевидны; таблица с каталогом кодов нужна лишь для целостности ссылочных данных и отображения кодов в графическом интерфейсе): ***

```
Таблица ORDER:                                                  Таблица ORDER_STATUS (каталог кодов):

order_id        customer_id     order_status_code               code                    description
123001          98765           'CLOSED'                        'OPEN'                  'Open'
123002          43210           'OPEN'                  'CANCELLED'             'Cancelled'
123003          12345           'CANCELLED'             'CLOSED'                'Closed'
```

Как бы то ни было, в нашем коде мы должны применять константы или перечисления, а не значения напрямую:

*** ПЛОХОЙ код: ***

```
if (newStatus == 'CLOSED')
```

*** ХОРОШИЙ код: ***

```
if (newStatus == OrderStatuses.Closed)
```

В первом подходе код читается достаточно хорошо, но подвержен багам: мы можем ошибиться со статусом при наборе и не получим ошибку при компиляции.

/* Комментарии */

Комментируйте свой код, если он не полностью очевиден. Это нужно не только для других разработчиков, которые будут пытаться понять ваш код, но и для вас самих: сейчас вы помните, что делает код, но сможете ли вспомнить через неделю, месяц, год? Но не оставляйте комментарии там, где они на самом деле не нужны. Невероятно, но я видел комментарии вида ‘Объявление переменных:’ непосредственно перед секцией объявления переменных и ‘Вызов метода XXX:’ прямо перед вызовом метода XXX! В то же время, действительно туманные фрагменты были абсолютно без комментариев!

Комментарии помогут вам не только понять существующие методы, но и написать новые. Сразу же после создания нового метода напишите комментарий перед каждым из будущих фрагментов кода, выполняющих различные логические задачи (как будто вы комментируете существующий код) и ПОСЛЕ ЭТОГО приступайте к написанию исполняемого кода. Ниже представлен пример исходного скелета (образованного комментариями!) для метода, который вычисляет новый баланс для заказчика (также обратите внимание, что в англоязычных комментариях мы можем избегать артикли «the» и «a», тем самым сокращая их длину, что всегда хорошо):

```
private decimal calcNewBalance (integer customerID, decimal amountToAdd)
        {
                        // Validate parameters:

                        // Retrieve existing balance of customer:

                        // Retrieve their allowed overdraft:

                        // Calculate new balance:

                        // If it exceeds allowed overdraft then return original balance:

                        // If this code reached then new amount is ok - return it:

        }
```

На столь ранней стадии создания метода легче сфокусироваться на том, что должно быть в нем реализовано (я бы назвал это «бизнес-логикой высокого уровня»). И только после подготовки скелета из комментариев мы можем приступить к написанию выполняемого кода (каждый фрагмент идет сразу после соответствующего комментария). При таком подходе остается значительно меньше шансов на необходимость переписывания кода.

Закон Деметры

Представьте: вы делаете покупку в магазине, и кассир просит вас предъявить кредитную карту. Вы достаете бумажник из кармана, вынимает карту из бумажника и вручаете ее кассиру. Есть ли в этом что-то странное или интересное? Не думаю. Но представьте слегка видоизмененную ситуацию: вместо того чтобы попросить вашу карту, кассир молча достает бумажник из вашего кармана и вынимает карту из бумажника! Что бы вы сказали такому нахальному парню?

Стоп, стоп, стоп… Уверены ли вы в том, что ваши объекты не ведут себя так же как этот странный кассир? Давайте обе ситуации, произошедшие в супермаркете, опишем на языке программирования:

СИТУАЦИЯ 1: *Кассир просит покупателя предъявить кредитную карту:*

```
creditCard = Buyer.GetCreditCard();
```

Метод GetCreditCard() покупателя представлен следующим кодом:

```
return this.GetJacket().GetUpperLeftPocket().GetWallet().GetMastercard();
```

СИТУЦИЯ 2: *Кассир нахально достает карту из бумажника покупателя, взятого из его кармана:*

```
creditCard = Buyer.GetJacket().GetUpperLeftPocket().GetWallet().GetMastercard();
```

В обоих случаях результат одинаков – кредитная карта извлечена. Но если говорить о возможных опасностях, существует громадное различие. В первом случае покупатель ответственен за свою кредитную карту. Он знает, в каком кармане лежит кошелек и какой кредитной картой он хочет воспользоваться (если их несколько). Если в следующий раз покупатель решит положить бумажник в другой карман или предъявить другую кредитную карту, то это его личное дело. В терминах программирования: если метод хранения кредитной карты в классе «покупатель» изменился, то будет достаточно изменить только код метода GetCreditCard(). Все вызовы этого метода из любого места приложения будут продолжать работать! Вызываемые объекты не должны ничего знать о пиджаках, карманах и других деталях. Им достаточно знать о своих проблемах и не следует отвечать за личные дела других (хотите ли вы, чтобы кассир, неизвестная личность, несла ответственность за ваши кредитные карты?).

Поэтому запомните следующее правило: если у вас есть указатель на объект, и вы хотите получать от него данные, применяйте методы, объявленные в этом объекте, а не переменные и методы вложенных в него объектов. В вашем возрасте несерьезно играть с Матрешкой! Применяйте это правило к указателям в любом контексте видимости приложения: локальном (включая аргумент метода), экземпляре и классе (статическом).

Спустя годы с начала следования этому правилу я обнаружил, что это закон Деметры. Ooops… Это не мое изобретение… Какая досада! Я так хотел дать этому закону свое имя… :(

Оригинальная публикация: Elegant Coding Habits. Продолжение следует.

Рубрика: Методики
Метки: CSharp, SQL
Комментарии (11)
Отправить в Twitter, Facebook, ВКонтакте, Google+

Комментарии (11)

[...] Продолжение. Начало статьи здесь. [...]

Навыки элегантного программирования. Часть 2. « OpenQuality.ru | Качество программного обеспечения | Опыт экспертов | 26.09.2010 | 8:05 пп. Ответить #

[...] Зускин » Навыки элегантного программирования: Часть 1 и Часть [...]

OpenQuality.ru | Качество программного обеспечения | 01.11.2010 | 7:15 дп. Ответить #

Это переводная статья, как я понял? Деметра — это имя древнегреческой богини.
http://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BC%D0%B5%D1%82%D1%80%D0%B0

Посему, на русском языке Law of Demeter должно звучать как «Закон Деметры» — согласование рода и лица в русской письменности еще никто не отменял.

bialix | 16.11.2010 | 10:20 пп. Ответить #