**4. Оценка качества программного продукта на основе метрик Чидамбера и Кемерера**

**4.1. Теоретические сведения**

В 1994 г. С. Чидамбер и К. Кемерер (Shyam R. Chidamber и CrisЕ. Кетегег) предложили шесть проектных метрик, основанных на анализе методов класса, дерева наследования и других характеристик объектно-ориентированной среды.

*Взвешенные методы на класс* *WMC* (Weighted Method С1аss). Данная метрика позволяет измерять сложность классов с учетом сложности их методов. Метрика называется взвешенной, потому что весом метода считается количественная характеристика сложности метода.

Пусть в классе *С* определены *n* методов со сложностью *c*[1]. *c*[2], ..., *с*[*n*]. Для оценки сложности может быть выбрана любая метрика сложности (например, Холстеда или цикломатическая сложность в зависимости от интересующего критерия). Основным при этом является процесс нормализации этой метрики таким образом, чтобы номинальная сложность для метода принимала значение 1. В этом случае количество методов и их сложность могут служить своеобразным индикатором затрат, необходимых на реализацию и тестирование классов.

Кроме того, с ростом числа методов возрастает сложность дерева наследования, поскольку все подклассы должны наследовать методы их родителей. С ростом количества методов в классе применение этого класса становится все более специфичным, ограничивается возможность его многократного использования. По этим причинам требование к этой метрике таково: метрика *WMC* должна иметь разумно низкое значение.

В простейшем случае сложность метода можно определить количеством строк текста этого метода в анализируемой программе. Пусть *L*[*i*] характеризует количество строк текста у метода с номером *i*, а всего в классе содержится *n* методов. Тогда для этой метрики измерения методов сложность класса *К* вычисляется следующим образом:

|  | (1) |
| --- | --- |

Нетрудно заметить, что для измерения сложности класса метрика *WMC* зависит от способа измерения сложности методов этого класса.

Часто используется разновидность метрики *WMC*, когда все методы имеют одинаковый вес. В этом случае метрика носит наименование «Количество методов на класс» *NМ* (Number of Methods).

Метрика *NМ* применяется для измерения сложности классов на ранних этапах разработки системы, когда еще нет детальной информации о применяемых методах. В качестве практических рекомендаций специалисты советуют на этапе проектирования системы разделять классы со слишком большим количеством методов на несколько классов для снижения сложности используемых методов.

Оказывается, что подсчитывать количество методов в классе достаточно сложно. Возможны два противоположных варианта учета.

1. Подсчитываются только методы текущего класса. Унаследованные методы игнорируются, поскольку они считаются уже подсчитанными в тех классах, где определялись. В связи с этим лучшим показателем функциональных возможностей класса становится инкрементность класса, т. е. возможность его пошагового развития. Для понимания того, что делает класс, основным источником информации являются его собственные операции. Если класс не может отреагировать на сообщение самостоятельно (например, в нем отсутствует собственный метод), этот класс направляет сообщение своему родителю.

2. Подсчитываются как методы, определенные в текущем классе, так и все унаследованные методы. Этот подход подчеркивает важность пространства состояний в понимании класса (а не инкрементности класса).

На практике приемлем любой из описанных вариантов. Единственным условием является постоянство применения того или иного метода, чтобы не приходилось менять вариант учета при переходе к новому проекту. В этом случае будет обеспечен корректный сбор метрических данных.

*Глубина дерева наследования* *DIT* (Depth of Inheritance Tree) позволяет определить количество классов-предков, которые потенциально оказывают влияние на данный класс. Другими словами, эта метрика характеризует самый длинный путь по иерархии классов к данному классу от класса-предка. Этот показатель должен быть возможно большим, так как при большей глубине возрастает абстракция данных, снижается насыщенность класса методами. В то же время не следует забывать, что при достаточно большой глубине существенно возрастает сложность понимания и написания программы.

Количество потомков *NOC* (Number Of Child) позволяет определить количество непосредственных потомков данного класса.

С ростом значения *NOC* возрастает многократность использования, поскольку наследование представляет собой одну из форм повторного использования. В то же время при возрастании метрики *NOC* снижается уровень абстракции родительского класса. Это означает, что в действительности некоторые из детей уже могут не быть элементами родительского класса, вследствие чего возможно их некорректное использование. Кроме того, количество детей характеризует потенциальное влияние класса на проект. По мере роста *NOC* возрастает количество тестов, необходимых для проверки каждого ребенка.

Метрики *DIT* и *NOC* являются количественными характеристиками формы и размера структуры классов. Хорошо структурированная объектно-ориентированная система чаще организуется в виде леса классов, нежели как очень высокое дерево.

*Связанность между классами объектов СВО* (Coupling Between Object classes) дает возможность определить количество классов, с которыми связан данный класс. Это имеет существенное значение, когда один класс использует методы или экземпляры другого класса. Для данной метрики справедливы все утверждения, введенные ранее при характеристике связанности модулей. Другое определение метрики имеет следующую трактовку: *СВО* равно количеству сцеплений класса, а сцепление организует вызов метода или свойства в другом классе.

Данная метрика характеризует статическую составляющую внешних связей классов. С ростом значения *СВО* уменьшается абстракция данных, и многократное использование класса уменьшается.

Вполне очевидно, что чем более независим класс, тем легче его повторно использовать в другом приложении. Высокое значение данной метрики усложняет модификацию и тестирование, которое неминуемо следует за модификацией программного продукта. Чем больше сцеплений, тем выше чувствительность всего проекта к изменениям в отдельных его частях. Минимизация сцеплений между объектами улучшает модульность и содействует инкапсуляции проекта.

*СВО* для каждого класса должно иметь разумно низкое значение. Это также согласуется с рекомендациями по уменьшению сцепления стандартного ПО.

Количество откликов на класс *RFC* (Responce For С1аss) позволяет определить количество методов, которое может быть выполнено в ответ на получение сообщения данным классом. В этой метрике учитываются не только выполняемые методы данного класса, но и методы других классов, т. е. с помощью этой метрики можно оценить и степень потенциального «общения» конкретного класса с другими классами:

| *RFС* = {*R*} + {*М*} | (2) |
| --- | --- |

где {R} — множество методов, вызываемых методом *r*; {*М*} - множество всех методов в классе.

Возможно другое толкование метрики: *RFС* представляет собой количество методов класса, увеличенное на число методов других классов, вызываемых из данного класса.

Метрика *RFС* является мерой потенциального взаимодействия данного класса с другими классами и позволяет судить о динамике поведения объекта в системе, т. е. данная метрика характеризует динамическую составляющую внешних связей классов. Если в ответ на сообщение может быть вызвано большое количество методов, то усложняются тестирование и отладка класса, так как от разработчика тестов в этом случае потребуется высокий уровень понимания класса. Кроме того, возрастает длина тестовой последовательности.

С ростом *RFС* увеличивается сложность класса. Самая плохая величина отклика может использоваться при определении времени тестирования.

*Отсутствие сцепления в методах LСОМ* (Lack Cohesion Of Меthods) позволяет оценить зависимость методов класса друг от друга. Для вычисления этой метрики подсчитывается количество пар методов, которые не используют общие атрибуты класса. Затем подсчитывается количество пар методов, которые используют общие переменные. Метрика *LСОМ* равна разности между первым числом и вторым. Если при этом получается отрицательное число, то значение метрики считается равным нулю.

Каждый метод внутри своего класса обращается к одному или нескольким свойствам (экземплярным переменным). Метрика *LСОМ* показывает, насколько методы не связаны друг с другом через свойства (переменные). Если все методы обращаются к одинаковым свойствам, то значение *LСОМ* = 0.

Связность методов внутри класса должна быть высокой, так как это содействует инкапсуляции. Если метрика *LСОМ* имеет высокое значение, то методы слабо связаны друг с другом через свойства. Это увеличивает сложность, что ведет к возрастанию появления ошибок при проектировании. Высокие значения *LСОМ* означают, что класс, очевидно, спроектирован не самым лучшим образом, и требуется разбить его на два или более отдельных класса. Вычисление *LСОМ* помогает определить недостатки в проектировании классов, поскольку эта метрика характеризует качество упаковки данных и методов в оболочку класса. Следовательно, желательно сохранять высокий уровень связности в классе, добиваясь низкого значения метрики *LСОМ.*

**4.2. Выбор программного модуля для оценки качества**

Для оценки качества разработанного ПП достаточно взять код главной формы (или код каких-либо других функционально значимых модулей). Рассматриваемый код может располагаться в приложении к дипломной работы.

Рассмотрим текст программы для оценки ее качества с помощью метрик С. Чидамбера и К. Кемерера, которые позволяют оценить меру сложности объектно-ориентированной программы на основе анализа сложности методов классов программы и их связности, так как все классы работают во взаимодействии.

Текст программы на языке C# для реализации возможного алгоритма решения поставленной задачи представлен на рис. 4.1.

|  |  |
| --- | --- |
| Номера  строк | Строки программы |
| 1 | using System; |
| 2 | using System.Collections.Generic; |
| 3 | using System.Text; |
| 4  5 | namespace EX 1  { |
| 6  7 | class Электро  { .  private static double тариф = 1.84; |
| 8 |
| 9 | private static int потреблениеСреднее = 300; |
| 10 | private string фамилия; |
| 11 | private int потребление; |
| 12 | public Электро(string фамилия) |
| 13 | { |
| 14 | this.фамилия = фамилия; |
| 15 | потребление = потреблениеСреднее; |
| 16 | } |
| 17 | public Электро(string фамилия, int текущее, int предыдущее) |
| 18 | { |
| 19 | this.фамилня = фамилия; |
| 20  21 | потребление = текущее - предыдущее;  } |
| 22 | public double Сумма() |
| 23 | { |
| 24 | return потребление \* тариф; |
| 25 | } |
| 26 | public string Инфо() |
| 27 | { |
| 28 | return string.Format(“{0,-20} {1,-20} {2,10:f2} {3,10:d6}”, |
| 29 | "Электроэнергия",фамилия,Сумма(),потреблснне); |
| 30 | } |
| 31 | } |
| 32 | class Запрос |
| 33 | { |
| 34 | public static void Заполнить(Электро[] платеж) |
| 35 | { |
| 36 | string фам; |
| 37 | int счПред=0, счТек=0; |
| 38 | Console.Clear(); |
| 39 | for (int i = 0; i < плaтeж.Length; i++) |
| 40 | { |
| 41 | Console.Write("Платеж "+ i + Фамилия -> "); |
| 42 | фам = Console. ReadLine(); |
| 43 | Console.Write(“Платеж " + i + Текущее значение счетчика -> "); |
| 44 | счТек = int.Parse(Console.ReadLin()); |
| 45 | if (счТек > 0) |
| 46 | { |
| 47 | Console.Write("Платеж" + i + Предыдущее значение счетчика ->"); |
| 48 | счПред = int.Parse(Console.ReadLine()); |
| 49 | } |
| 50 | if(cчTек <= 0) |
| 51 | платеж[i] = new Электро(фам); |
| 52 | else |
| 53 | платеж[i] = new Электро(фам, счТек, счПред); |
| 54 | } |
| 55 | } |
| 56 | public static void Вывести(Электро[] платеж) |
| 57 | { |
| 58 | for (int i = 0; i < плaтеж.Lеngth; i++) |
| 59 | Соnso1е>WriteLine(платеж[i].Инфо()); |
| 60 | } |
| 61 | } |
| 62 | class Платежи |
| 63 | { |
| 64 | static void Main(string[] args) |
| 65 | { |
| 66 | ConsoleKeylnfo rep; |
| 67 | Электро[] плэ; |
| 68 | int кпэ; |
| 69 | do |
| 70 | { |
| 71 | Console.Clear(); |
| 72 | Console.Write("Количество платежей за электроэнергию:"); |
| 73 | кпэ = int.Parse(Console.ReadLine()); |
| 74 | плэ = new Электро[кпэ]; |
| 75 | Запрос.Заполнить(плэ); |
| 76 | Запрос. Вывести(плэ); |
| 77 | Console.WriteLine(“Для выхода нажмите ESC"); |
| 78 | rep = Console. ReadKey( true); |
| 79 | }while(rep.Key!= ConsoleKey.Escape); |
| 80 | } |
| 81 | } |
| 82 | } |

Рис. 4.1. Пример реализации программы «Платеж за электроэнергию»

**4.3 Взвешенные методы на класс**

Определим метрику взвешенных методов на класс WMC, которая определяет сложность класса на основе анализа сложности определенных в нем методов. Рассмотрим самый простой случай определения метрики сложности, когда в качестве критерия сложности принимается количество строк в исходном коде метода. Как отмечалось ранее, программа содержит в своем составе три класса. Проанализируем каждый из классов на предмет их сложности.

Класс Электро в своем составе имеет четыре метода:

* *public Электро (string фамилия)*;
* *public Электро (string фамилия, int текущее, int предыдущее);*
* *public double Сумма ();*
* *public string Инфо ().*

Одноименные методы *Электро* отличаются друг от друга количеством входных параметров. Метод *public Электро* (*string* *фамилия*) имеет исходный код в 5 строк (см. рис 4.1, строки 12-16). Метод *public Электро* *(string фамилия, int текущее, int предыдущее)* имеет в своем составе 5 строк (строки 17-21). Метод *public double Сумма ()* состоит из 4 строк (строки 22-25). Исходный код метода *public string Инфо* () включает 5 строк (строки 26-30). Тогда сложность класса *Электро* можно определить следующим образом:

*WMC (Электро)* = *L*[0] + *L*[1]+ *L*[2] + *L*[3] =5+5+4+5 = 19.

Класс *Запрос* имеет в своем составе два метода:

* *public static void Заполнить* (*Электро*[ ] *платеж*)*;*
* *public static void Вывести* (*Электро*[ ] *платеж*)*.*

Исходный код метода *public static void Вывести* (*Электро*[ ] *платеж*) состоит из 21 строки (строки 34-55). Исходный код метода *public static void Вывести* (*Электро*[ ] *платеж*) влючает 5 строк. Сложность класса *Запрос* составляет:

*WMC (Запрос)* = *L*[0] + *L*[1] =21+5 = 26.

Класс *Платежи* содержит всего лишь один метод *static void Main* (*string*[ ] *args*), который в своем составе имеет 17 строк. Сложность класса *Платежи* составляет:

*WMC (Платежи)* = *L*[0] = 17.

Исходя из полученных результатов можно определенно заметить, что наиболее сложным является класс *Запрос*, так как в этом классе сосредоточены наиболее сложные методы.

Определим значение метрики *NM* (количество методов на класс). При разработке исходного кода поставленной задачи принцип наследования не применялся, поэтому в коде программы отсутствуют базовые классы и их наследники. Для определения значения метрики *NM* будем подсчитывать только методы текущих классов:

* *NM (Электро)* = 4;
* *ММ (Запрос)* = 2;
* *ММ (Платежи)* = 1.

Исходя из показателей метрики *NМ*, самым сложным является класс *Электро*, так как этот класс содержит наибольшее количество методов.

**4.4. Связанность между классами объектов**

Определим связность между классами объектов *СВО*, которая численно определяется по количеству классов, с которыми связан анализируемый класс. Связь класса с другими классами определяется количеством обращений к вызову методов в других классах. В классе *Электро* осуществляется один вызов метода string.Format из другого класса *string* (см. рис. 4.1, строка 28), следовательно, *СВО (Электро)* = 1.

В классе *Запрос* обращения к методам внешних классов осуществляется 12 раз:

* 8 раз - к различным методам класса *Console* (строки 38, 41, 42, 43, 44, 47, 48 и 59);
* 2 раза - к методу *Parse* класса *int* (строки 44 и 48);
* 2 раза - к конструктору класса *Электро* (строки 51 и 53).

Таким образом, *СВО (Запрос)* = 12.

В классе *Платежи* обращение к методам внешних классов осуществляется 10 раз:

* 5 раз - к методам класса Console (строки 71, 72, 73, 77 и 78);
* 1 раз - к методу Parse класса int (строка 73);
* 1 раз - к конструктору класса Электро (строка 74);
* 2 раза - к методам класса Запрос (строки 75 и 76);
* 1 раз - к свойству класса ConsoleKey (строка 79).

Таким образом, СВО (Платежи) = 10.

Из полученных результатов значений метрик СВО следует, что наименее чувствительным к модификации является класс Электро, так как уровень связности его объектов с другими объектами является наименьшим. Уровни связности классов Запрос и Платежи относительно велики, что говорит об усложнении процессов модификации и тестирования всей программы в целом.

**4.5 Количество откликов на класс**

Определим количество откликов на класс RFC, которое определяется числом методов, которые могут быть вызваны в рассматриваемом классе. Значение метрики RFC определяется по следующему соотношению:

RFC = R + М,

где R - это методы, вызываемые методами рассматриваемого класса, М - общее количество методов класса.

Как уже рассматривалось ранее, класс Электро содержит четыре метода (А/ = 4), в одном из них вызываются методы string. Format () и Сумма () (R = 2). Таким образом, метрика RFC равна:

RFC (Электро) = R + М = 2 + 4 = 6.

Класс Запрос содержит два метода, которые осуществляют вызов 12 внешних методов, следовательно, R= 12, М = 2.

RFC (Запрос) = R + М= 12 + 2 = 14.

Класс Платежи содержит один метод, который осуществляет вызов 10 внешних методов, поэтому R= 10, М = 1.

RFC (Платежи) = R + М = 10 + 1 = 11.

Сложность классов Запрос и Платежи почти в два раза выше сложности класса Электро, поэтому для тестирования вызываемых методов в классах Запрос и Платежи потребуется большее количество тестов, следовательно, и отладка этих классов займет большее количество времени.

**4.6 Сцепления в методах**

Определим значение метрики отсутствия сцепления в методах LCOM.

Числовое значение этой характеристики определяется разностью между количеством пар методов, которые не используют одни и те же переменные в классе, и количеством пар методов, которые используют одни и те же переменные класса. Начнем с класса Электро. Все четыре метода этого класса попарно обращаются к одинаковым переменным класса:

* методы public Электро (,string фамилия) и public Электро (string фамилия, int текущее, int предыдущее) обращаются к переменной потребление;
* методы public Электро (string фамилия) и public double Сумма() обращаются к переменной потребление;
* методы public Электро (string фамилия, int текущее, int предыдущее) и public double Сумма() обращаются к переменной потребление’,
* методы public Электро (string фамилия) и public string Инфо() обращаются к переменным потребление и фамилия’,
* методы public Электро (string фамилия, int текущее, int предыдущее) и public string Ипфо() обращаются к переменным потребление и фамилия',
* методы public double Сумма() и public string Инфо() обращаются к переменной потребление.

Таким образом, в классе Электро нет ни одной пары методов, не использующих общих атрибутов класса, и насчитывается 6 пар методов, которые обращаются к одинаковым переменным.

LCOM = 0-6 = -6.

Если LCOM < 0, то значение этой метрики принимают равным 0. Следовательно, для Электро значение LCOM - 0.

В классе Запрос содержится два метода. Эти методы не используют общие атрибуты класса по причине их отсутствия. Таким образом:

LCOM (Запрос) =1-0=1.

В классе Платежи имеется только один метод, следовательно, LCOM= 0.

Анализируя значения метрик LCOM всех классов программы, можно отметить, что уровень связности методов в классах достаточно высок, что позволяет в полной мере обеспечить инкапсуляцию классов.

**Выводы**

1. Рассмотрены основные метрики С. Чидамбера и К. Кемерера (WMC, NM, RFC, CBO, LCOM), основанные на анализе методов класса, дерева наследования и других характеристик объектно-ориентированной среды.

2. Проанализирован основной программный модуль разработанного ПП по метрикам С. Чидамбера и К. Кемерера.

3. Согласно метрике WMC наиболее сложным является класс *Запрос*, так как в этом классе сосредоточены наиболее сложные методы. Исходя из показателей метрики *NМ*, самым сложным является класс *Электро*, так как этот класс содержит наибольшее количество методов.

4. Из метрик *СВО* следует, что наименее чувствительным к модификации является класс *Электро*, так как уровень связности его объектов с другими объектами является наименьшим. Уровни связности классов *Запрос* и *Платежи* относительно велики, что говорит об усложнении процессов модификации и тестирования всей программы в целом.

5. По метрике FRC сложность классов *Запрос* и *Платежи* почти в два раза выше сложности класса *Электро*, поэтому для тестирования вызываемых методов в классах *Запрос* и *Платежи* потребуется большее количество тестов, следовательно, и отладка этих классов займет большее количество времени.

6. Анализируя значения метрик *LCOM* всех классов программы, можно отметить, что уровень связности методов в классах достаточно высок, что позволяет в полной мере обеспечить инкапсуляцию классов.