3.4. Метрика и качество кода

Взять бы много кирпичей,
штук примерно пять,
Вот бы вышел славный дом,
только где их взять?
Песенка кума Тыквы из м/ф «Чиполлино»

Ввиду того, что в программировании отсутствуют рамочные законы, позволяющие оценить или рассчитать фундаментальные свойства программного продукта на основе его кода, приходится делать это косвенным образом. В 1.1 было введено понятие *качества кода* – формальное соответствие кода набору правил. Сами по себе они ничего не гарантируют, но их соблюдение способствует повышению качества процесса разработки в целом.

Цели получения метрических характеристик качества кода:

- поддержание стандартов кода при коллективном владении кодом;
- обнаружение потенциально опасного кода;
- обнаружение узких мест в структуре кода;
- мониторинг разрабатываемого проекта в системе контроля качества при наличии статистики, развернутой по времени;

Текущее состояние в этой области выглядит следующим образом:

- в основном преобладает прагматический подход: используются средства поддержания стандартов кодирования - единообразная стилистика кода и простые количественные метрики;
- отдельно развивается научно-исследовательский подход: разработка метрик, отражающих свойства *правильного* кода;
- сами по себе метрики не гарантируют эффективной реализации функционала;
- собранная метрическая статистика нуждается в обработке и интерпретации.

Таким образом, метрические характеристики кода в настоящее время достаточно разнообразны, но сами по себе они не являются показателями реального качества программного продукта и *нуждаются в интерпретации*.

В форме показателей качества могут выступать различные формальные и стилистические свойства:

- свойства стиля:
  - документированность;
  - читаемость;
  - устойчивость к потенциальным ошибкам.
- свойства структуры кода:
  - модульность;
  - сложность;
  - инкапсуляция - сокрытие свойств;
  - управляемость - автономность, независимость, связность.

Метрика – количественная оценка (мера) некоторого свойства кода, создаваемая путем введения параметра, который может его характеризовать

Говоря о метриках, нужно помнить, что они отражают *формальное качество кода,* которое может способствовать улучшению его реальных качеств, однако это не происходит автоматически. Между реальным и формальным качеством существует определенная корреляция, но это вовсе не означает, что формально некачественный код будет обязательно плохим, а формально качественный – хорошим. Реальное качество можно рассматривать в разных аспектах:

- **развитие**: понимаемость - внятность, эстетика, универсальность, модульность, управляемость, простота адаптации и повторного использования;
- **надежность:** потенциальное наличие ошибок, устойчивость к ошибкам, трассируемость, восстанавливаемость;
- **эффективность:** производительность, трудоемкость, использование памяти.

Очевидно, что перечисленные аспекты расположены в порядке убывания влияния на них формального качества кода. Различные виды преобразования кода по-разному связаны с его метрикой: *рефакторинг***,** как правило, сопровождается улучшением метрики, а *оптимизация и реинжиниринг* меняют структуру кода таком образом, что его метрические характеристики не всегда сопоставимы.

Стилистические метрики

Стилистика кода позволяет сохранять дополнительную информацию о структуре кода в самом его оформлении. Современные IDE помогают программисту ее соблюдать. Стилистику наиболее просто контролировать, можно найти множество частных рекомендаций по оформлению кода [3-1]. И наконец, стилистика – это стандарт оформления, которого должна придерживаться команда разработчиков, чтобы, как минимум, не вызывать дискомфорта при чтении чужого кода.

К стилистике причисляются правила, которые имеют самое разное отношение к языку и практике его применения: эстетика, синтаксис и семантика языка, шаблоны конструирования:

- комментарии по сложным участкам кода, *шапки* методов – часть программной документации;
- форматирование текста программы, соответствующее синтаксической структуре кода, например, отступ по уровню вложенности - позволяет визуально контролировать синтаксис операторов;
- соглашения об именах переменных, методов, интерфейсов и аббревиатуры – позволяют по имени определить принадлежность к определенному функционалу, свойства и особенности именованного объекта. Например, закрытые свойства класса рекомендуется сопровождать префиксом «m» (*mFirstValue*), имена методов содержат аббревиатуры выполняемых действий, типа возвращаемого результата или его формата (*createParamList, getSelectedRecords*);
- предупреждение возможных ошибок при редактировании кода. Например, тело цикла и условного оператора рекомендуется заключать в фигурные скобки, даже если оно состоит из одного оператора на случай будущей замены оператора последовательностью;
- синтаксически допустимые конструкции, которые не согласуются с хорошим стилем программирования. Например, не рекомендуется перехватывать исключения всех типов конструкцией catch *Throwable(ee)*, а делать это отдельно по разным источникам.

Замечание по теме. Любая стилистика имеет свои эргономические и эстетические обоснования. Например, мне удобнее форматирование, обеспечивающее компактное представление кода (рис.3-47). Это противоречит некоторым положениям стилистики, например, размещение открывающейся фигурной скобки на отдельной строке, зато позволяет видеть одновременно на экране больший участок текста.

```
//=====================================================================
private Throwable eout=null;
private DBItem[] getRecords(DBItem item, final Vector<DBField> ff, String sql
    eout=null;
    final Class cls=item.getClass();
    final Vector<DBItem> out=new Vector();
    srv.selectMany(sql, new DBRecordCallBack(){
        @Override
        public void procRecord(ResultSet rs) {
            try {
                DBItem it=(DBItem)cls.newInstance();
                it.setFields(ff);
                it.loadDBValues(rs);
                out.add(it);
                } catch(Throwable ee){ eout=ee; }
            }
        });
    if (eout!=null) throw new SQLException(eout.getMessage());
    DBItem xx[]=new DBItem[out.size()];
    out.toArray(xx);
    return xx;
    }
```

Рис.3-47. Компактное форматирование кода

Замечание по теме. К стилистике относятся также варианты выбора эквивалентных синтаксических конструкций языка. Например, тело цикла со значительным количеством *break и continue* будет более компактным, нежели с эквивалентными *if...else*, к тому же будет содержать меньше фигурных скобок. Однако, такая бейсик-подобная стилистика не очень приветствуется в структурном программировании.

Количественные метрики

Следующая группа метрик, за неимением лучшего, используется для оценки объема и сложности программного кода и далее вплоть до трудоемкости программного проекта. Общеизвестной здесь является **SLOC** (*Source Lines of Code*) - количество строк исходного кода, операторов, комментариев.

Применимость SLOC обусловлена разными факторами:

- парадокс меры состоит в том, что более примитивные решения в коде дают более объемный и, следовательно, трудоемкий с точки зрения метрики код . В то же время компактное изящное будет проигрывать. Поэтому при сравнении двух функционально идентичных программ мера работает с *точностью до наоборот*. В то же самое время компактное решение является *запутанным* с точки зрения других метрик;
- SLOC может использоваться как удельная мера модульности: строк кода, операторов на один модуль или файл, на одну функцию, процент строк с комментариями. Предпочтительными являются короткие функции и модули;
- готовые закрытые решения и сторонние средства - библиотеки, фреймворки - автоматически понижают SLOC за счет реализованного ими функционала;

- визуальные средства конструирования, производящие код или структурные описания, плохо вписываются в метрику SLOC;
- в однородном технологическом процессе с линейкой однородных продуктов и стабильной командой разработчиков SLOC может использоваться как метрика процесса производства.

Программный код формально можно рассматривать как обычный текст на основе ограниченного набора слов - словаря. **Мера Холстеда** рассматривает программу с позиций теории информации как некоторое сообщение, информационная мера которого используется как метрика кода программы. Основные определения меры:

- **n1** – словарь действий - операторов (ключевые слова, знаки операций, операторы, символы-разделители);
- **n2** – словарь сущностей – операндов (имена типов данных, переменные, константы);
- **N1** – количество операторов;
- **N2** – количество операндов;
- **n1', n2'** – теоретический словарь программы – словари действий и сущностей всего языка;
- **n1 + n2** – словарь программы;
- **N = N1 + N2** – длина программы;
- **$V = N \log_2(n)$** – объем программы;
- **$V'=N' \log_2 n'$** — теоретический объем программы;
- **$N' = n1 \log_2(n1) + n2 \log_2(n2)$** – теоретическая длина программы.

Основной принцип меры: символы программы учитываются в мере линейно, а их многообразие – как степень соответствующей двойки или логарифм по основанию 2. Пример расчета метрики для функции двоичного поиска приведен на рис.3-48.

```
//------Двоичный поиск в упорядоченном массиве
int binary(int c[], int n, int val){  // Возвращает индекс найденного
int a,b,m;                            // Левая, правая границы и
    for(a=0,b=n-1; a <= b;) {         // середина
    m = (a + b)/2;                    // Середина интервала
    if (c[m] == val)                  // Значение найдено -
    return m;                         // вернуть индекс найденного
    if (c[m] > val)
        b = m-1;                      // Выбрать левую половину
    else
        a = m+1;                      // Выбрать правую половину
    }
return -1; }                          // Значение не найдено
```

```
n1 = 18     - (,), «,» ,{,},=,<=,==,+,-,/,[.],while,if,else,return,«;»;
n2 = 11     - binary,int,c,n,m,val,a,b,1,2,0;
N1 = 51     - количество операторов;
N2 = 38     - количество операндов;
n  = 29     - словарь программы;
N = 89      - длина программы;
N' = 113    - теоретическая длина программы (словарь);
V = 433     - объем программы;
```

Рис.3-48. Мера Холстеда для функции двоичного поиска

Замечание по теме. В любом случае в оценке возникают разные нюансы, связанные с синтаксисом языка. Например, как считать синтаксически связанные элементы - парные скобки, или символ «*запятая*» в различных вариантах использования.

Сложность потоков управления и данных

Сложность структуры кода можно оценить, рассматривая поток управления и поток данных программы и их взаимоотношения. Каждый из них можно представить в виде графа, а уже сам граф оценивать как структуру определенной сложности. Самая простая метрика определяется для потока управления.

Граф потока управления отображает возможную связность по управлению - переходы между различными точками программы. Его проще всего построить на основе блок-схемы программы (рис.3-49):

- вершины графа – уникальные дуги блок-схемы, *точки останова*;
- дуги графа соединяют вершины, если между ними имеется путь через условие (ромб) или действие (прямоугольник).

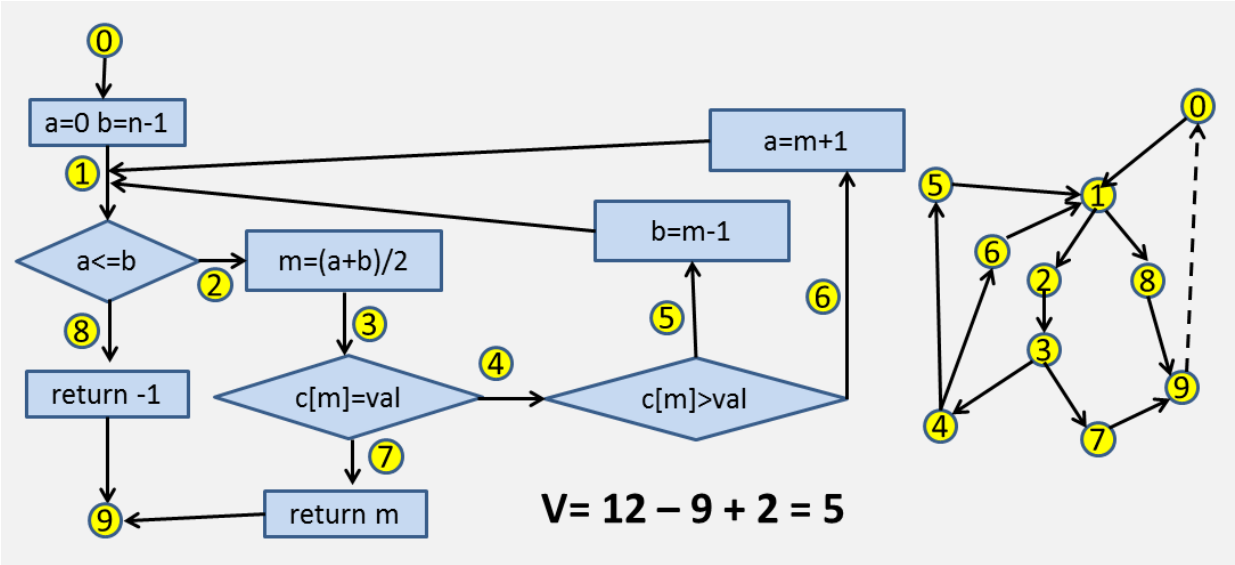


Рис. 3-49. Граф потока управления

Для ориентированного графа, каковым является граф потока управления, известна оценка его сложности - **цикломатическая сложность графа** в виде **V = E - N + 2P**, где **E** — количество дуг, **N** — количество вершин, **P** — число компонент связности. Число компонентов связности графа можно рассматривать как количество дуг, которые необходимо добавить для преобразования графа в сильно связный. Сильно связным называется граф, любые две вершины которого взаимно достижимы. Для графов корректных программ, т. е. графов, не имеющих недостижимых участков от точки входа и *висячих* точек входа и выхода, сильно связный граф, как правило, получается путем замыкания дугой вершины, обозначающей конец программы, на вершину, обозначающую точку входа в эту программу. В результате получается **V = E – N + 2.** Для нашего примера **V= 12 – 10 + 2 = 4.** Свойства такой оценки:

- длина линейных участков на оценку не влияет;
- для *пустой* программы V = 1.

Формально эта метрика указывает превышение числа дуг над числом вершин, а поскольку количество дуг возрастает только при проверке условий, то данную метрику можно рассматривать как показатель *ветвистости* или *запутанности* кода.

Метрики связей модульного кода

Модуль, состоящий из компонент - методы классы, библиотека функций- может быть охарактеризован с точки зрения способов взаимодействия компонент между собой - **связность (cohesion)** и способа обращения к ним извне **сцепление (coupling)** (рис.3-50)**.** Характеристики являются качественными, они отражают *способ связи* компонент.

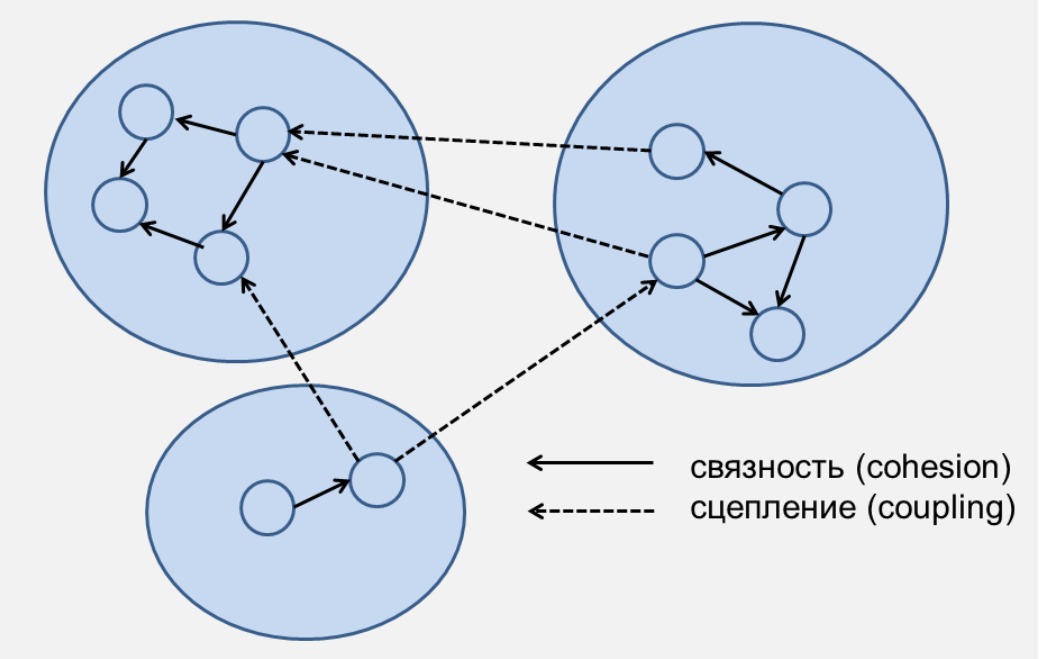


Рис. 3-50. Связность и сцепление

Виды связности обозначаются весами, отражающими степень связности компонент:

- *совпадение* (СВ=0) – компоненты не имеют ничего общего, кроме факта нахождения в общем модуле;
- *логическая* (СВ=1) – компоненты в рамках общего функционала, но не связаны между собой, например, обработка ошибок различных типов;
- *временная* (СВ=3) - используются на одной фазе процесса, в один период времени - инициализация, завершение;
- *процедурная* (СВ=5) – используются в рамках одного сценария, имеют определенный порядок вызова, т.е. могут быть связаны между собой по результату;
- *коммуникативная* (СВ=7) – работают с общей структурой данных;
- *последовательная* (СВ=9) – результат первого является входом второго;
- *функциональная* (СВ=10) – один модуль вызывает другой.

В ООП вводится *объектная связность* - в рамках класса с общими свойствами и функциональностью, она может быть любой из перечисленных, но для класса естественным выглядит использование трех последних.

Виды сцепления также обозначаются весами:

- сцепление по данным (СЦ=1) – вызов с параметрами – примитивными типами данных;
- сцепление по образцу (СЦ=3) – вызов с параметрами-объектами, структурами данных;
- сцепление по управлению (СЦ=4) - модуль устанавливает флаги в другом модуле, управляя его поведением;

- сцепление по внешним ссылкам (СЦ=5) - модули используют один и тот же глобальный элемент данных или ссылку на него;
- сцепление по общей области (СЦ=7) - модули разделяют одну и ту же глобальную структуру данных или используют ссылку на общий объект;
- сцепление по содержанию (СЦ=9) - один модуль прямо исполняет часть кода другого модуля.

Сама по себе степень связности или сцепления не является показателем хорошего или плохого кода. Они лишь являются индикаторами, сигнализирующими о сложностях с *управляемостью кода*. Высокая степень связности и сцепления говорит о том, что существуют косвенные способы повлиять на поведение одного компонента из другого, что может быть причиной трудно обнаруживаемых и локализуемых ошибок.

Замечание по теме. Способы сцепления и связности имеют различные исторические варианты реализации в языках программирования. *Сцепление по содержанию* поддерживалось механизмом **сопрограмм:** имелось два логически независимых потока управления, оба передавали управление друг другу по принципу *пинг-понга*: *A – B – продолжение A – продолжение B*. На уровне архитектуры это можно было сделать с помощью команды вызова процедуры *call* с аргументом – адресом возврата в стеке. В настоящее время сцепление по содержанию реализовано в шаблоне проектирования *обратный вызов* (см. 3.3). Указатели на функции в Си и анонимные функции (*лямбда-выражения*) также используют способ исполнения фрагментов стороннего кода в текущем компоненте.

Объектно-ориентированные метрики

Определенные выше понятия связности и сцепления связаны исторически со структурно-функциональным подходом - функции как основной компонент модуля. В ООП класс представляет собой единство данных и методов, поэтому связность компонент класса можно оценивать как со стороны данных - разделение их методами, так и со стороны методов - использование общих данных (рис.3-51).

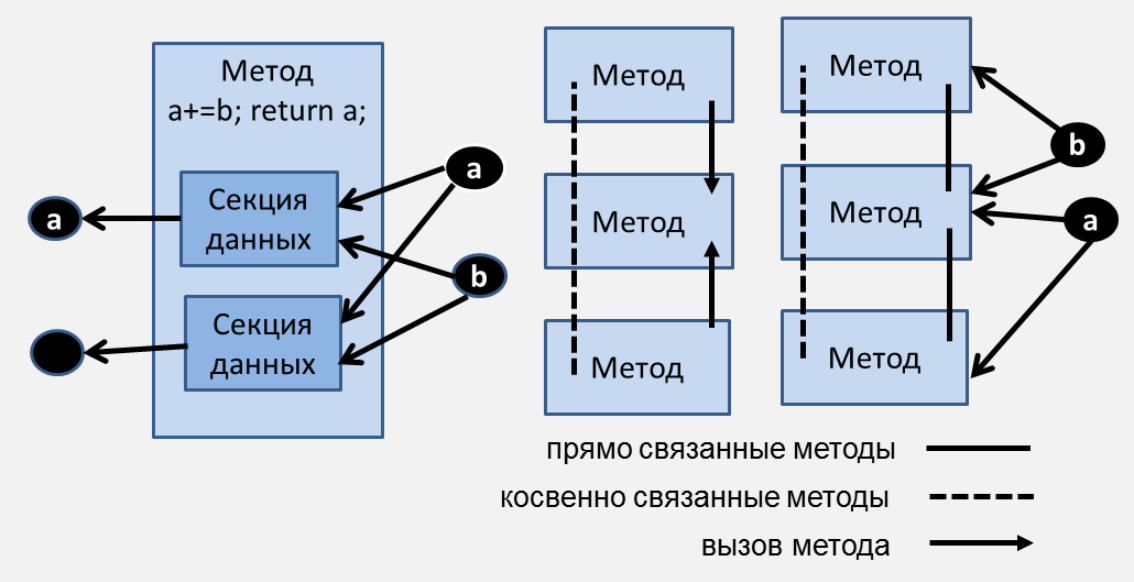


Рис.3-51. Связность класса по данным по методам

Метрика связности (cohesion) класса по данным

В метрике связности по данным базируется на следующих определениях:

- *токен (token)* – элемент данных класса - переменные, ссылки и константы;

- *секция данных* – набор токенов для вычисления одного из выходных параметров метода - результата метода или элемента данных класса. Количество секций данных метода - количество изменяемых им токенов плюс результат метода;
- *склеенный токен* – разделяемый токен, который используется более чем в одной секции данных;
- *сильно склеенный токен* – токен, который используется во всех секциях данных;
- *слабая связанность по данным (***WDC** – *Weak Data Cohesion)* – доля склеенных токенов;
- *сильная связанность по данным* (**SDC** – *Strong Data Cohesion*) – доля сильно склеенных токенов.

Финальным параметром является *клейкость данных (DA - Data Adhesiveness),* определяемая как доля дуг *токен-секция* по отношению максимально возможному количеству, т.е. **DA = N / (S * T),** где N - количество дуг *токен-секция,* S – количество секций, T – количество токенов.

Имеются технологические нюансы: конструкторы, set/get-методы в метрике не учитываются, при вызове метода A из метода B секции метода A учитываются в B.

В качестве примера рассмотрим крайние случаи минимальной и максимальной связности.

Вырожденный случай, когда методы не связаны с данными вообще, дает нулевые значения всех параметров. Минимальная связность, имеющая смысл, будет у класса, в котором каждый элемент данных связан с методом, его изменяющим и не возвращающим результата, т.е. в форме 1-1-соответствия:

- токенов – N, секций/методов – N;
- SDC = 0, WDC = 0, DA = N / (N * N) = 1/ N.

Если же методы будут возвращать измененные значения, то получим:

- токенов – N, методов – N, секций – 2N, каждый токен связан с двумя секциями данных;
- SDC = 0, WDC = N/N = 1, DA = 2N / (2N * N) = 1/ N.

Максимальная связность имеет место при использовании всеми методами, не возвращающими результат, единственного токена**:**

- токенов – 1, секций/методов – N;
- SDC = 1, WDC = 1, DA = N / (N * 1) = 1.

Метрика связности класса по методам

Связность по методам выглядит проще и рассматривает только факт использования элемента данных класса безотносительно к его изменению или причастности к формированию результата. Выглядит это так:

- пара *прямо связанных методов* имеет общий элемент данных, NT – количество прямо связанных пар;
- пара *косвенно связанных методов* имеют общий метод, с которым они связаны прямо, либо вызывают его. NL - количество косвенно связанных пар;

- **NP = N * (N-1) / 2 –** общее количество пар методов;
- **TCC = NT / NP** *сильная связность класса* (*Tight Class Cohesion)*;
- **LCC = (NT + NL) / NP** - *слабая связность класса* (*Loose Class Cohesion*).

Из технологических нюансов: связность может определяться как с учетом наследования, так и без него, т.е. для текущего класса.

Прагматические объектно-ориентированные метрики

В качестве метрик используются также наборы прямых структурных характеристик кода класса, которые могут применяться для комплексной оценки качества кода класса. В некоторых случаях имеют место рекомендации по их допустимым значениям.

Метрики Чидамбера и Кемерера предназначены для комплексной оценки качества класса и включают в себя:

1. взвешенные методы на класс - **WMC** (*Weighted Methods Per Class*) – метрика количества и сложности методов: суммарная нормированная сложность кода методов, например, цикломатическая сложность потока управления, общее количество методов. Варианты: учет унаследованных или только собственных методов;
2. высота дерева наследования - **DIT** *(Depth of Inheritance Tree*) - максимальная длина пути, количество вершин – классов по дереву наследования;
3. количество детей - **NOC** (*Number of children*) – среднее количество прямых наследований класса;
4. сцепление между классами - **CBO** (*Coupling Between Object classes*) – общее количество вызовов методов и использования свойств объектов других классов в коде класса;
5. *отклик класса* - **RFC** (*Response For a Class*) – количество методов класса плюс количество методов других классов, вызываемых из данного класса. В отличие от CBO учитываются только связи по управлению;
6. *недостаток связности в методах* **LCOM** (*Lack of Cohesion in Methods*) - количество пар несвязанных методов (не использующих совместно хотя бы один элемент данных класса) минус количество пар связанных методов. Если значение становится отрицательным, то оно приравнивается 0.

Метрика Мартина предназначена для оценки кода проекта в целом с точки зрения сцепления - внешних связей между категориями классов. Категория классов – группа функционально связанных классов, обычно оформленных в виде пакета:

1. *центростремительное сцепление* **Ca** – количество классов вне категории, зависящих от классов этой категории;
2. *центробежное сцепление* **Ce** - количество классов внутри категории, которые зависят от классов вне этой категории;
3. *нестабильность* **I = Ce / (Ca+Ce);**
4. *абстрактность* **A** – доля абстрактных классов;
5. в категории *сбалансированы абстрактность и нестабильность*, если **I+A=1**, тогда она относится к главной последовательности. На этой основе вводятся две метрики:
   - **D=|(A+I-1)/√2|** - расстояние до главной последовательности
   - **Dn=|A+I-2|** - нормализированной расстояние до главной последователь-ности.

Метрики Лоренца и Кидда содержат как характеристик классов, так и системы в целом. К тому же они содержат рекомендации для диапазонов значений:

1. *размер класса* - **CS** (*Class Size*) – количество методов, в т.ч. унаследованных и закрытых плюс количество свойств. CS ≤ 20 методов;
2. *количество методов, переопределяемых подклассом* - **NOO** (*Number of Operations Overridden by a Subclass*). NOO ≤ 3;
3. *количество операций, добавленных подклассом* - **NOA** (*Number of Operations Added by a Subclass*) Для CS = 20 и DIT = 6, NOA ≤ 4;
4. *индекс специализации* – **SI** (*Specialization Index*), **SI = NOO * L / M**, где L – уровень наследования, M – общее количество методов. SI $\leq 0{,}15$;
5. *средний размер операции* **AOS** (*Average Operation Size*). AOS $\leq$ 9, SLOC-метрика кода;
6. *сложность операции* **OC** (*Operation Complexity*) – одна из метрик сложности метода - цикломатическая, Холстеда. Авторами предлагается собственная калькуляция с весами для различных типов операций;
7. *среднее количество параметров на операцию* **ANP** (*Average Number of Parameters per operation*). ANP = 0,7;
8. *количество описаний сценариев* **NSS** (*Number of Scenario Scripts*) – комплексный показатель для классов, управляющих поведеним;
9. *количество ключевых классов* **NKC** (*Number of Key Classes*). NKC > 0,2 от общего количества классов системы;
10. *количество подсистем* **NSUB** (*Number of SUBsystem*) NSUB > 3.

Запутывающие преобразования

Запутывание, обфускация - формальные преобразования кода, не меняющие функционал, с целью защиты от повторного использования кода сторонними лицами. Обфускация производится как над исходным текстом, так и над скомпилированным кодом - внутренним представлением. Как правило, она сопровождается снижением качества кода, ухудшением его метрик. Способы запутывания нацелены на то, чтобы усложнить анализ программного кода при его реинжиниринге как в исходном тексте, так и после восстановления (дизамссемблирования, декомпиляции) из внутреннего представления:

- **запутывание форматирования исходного текста** (*layout obfuscation*) - стилистические преобразования - изменения имен, форматирования, удаление комментариев, оказывает влияние только стилистику кода и на SLOC-метрики;
- **запутывание данных** (*data obfuscation*):
  - *преобразования размещения* переменных, изменение их классов памяти (локальные, глобальные, объектные) и формы кодирования значений вплоть до шифрования;
  - *преобразования агрегации* – преобразование структур данных, *слияние* нескольких скалярных переменных в одну, использование неочевидных представлений данных, сжатие, шифрование;
- **запутывание управления** (*control obfuscation*):
  - *преобразование агрегации* - вставка (*inline*) и вынос (*outline*) функций, кпирование кода, *раскрутка циклов* в последовательности шагов, переупорядочение операторов;

- *преобразование вычислений* - вставка *мертвого* кода, диспетчеризация – замена прямого потока управления вызовом или передачей управления фрагментам кода;
- *табличная интерпретация* – автоматные модели поведения программы, переменные состояния;

- **превентивные трансформации** (*preventive transformation*) – преобразования, усложняющие декомпиляцию или деобфускацию двоичного представления.

Запутывание упоминается здесь еще и потому, что некоторые его способы используются в преобразовании кода с совершенно другими целями, например:

- повышение эффективности системы за счет оптимизации алгоритмов, структур данных;
- защита программного кода и хранимых данных от несанкционированного доступа и расшифровки.

Скорее всего, такие преобразования будут сопровождаться усложнением кода и ухудшением его метрик.

Замечание по теме. Очень может быть, что изящное и компактное решение, которое не просматривается напрямую в коде, по своей природе является результатом запутывания очевидного решения, в котором логика *зашита в код*. Использование таблиц решений, переменных состояния, автоматных моделей и т.п. вместо многоэтажных условных конструкций делают код более *управляемым,* но менее качественным с точки зрения его запутанности.

Средства контроля качества кода

Контроль качества кода поддерживается различными средствами:

- автономные программные продукты;
- облачные сервисы;
- плагины, встраиваемые в средства разработки.

Автономные программные продукты - SonarQube

SonarQube – свободно распространяемый проект с открытым кодом, использующий стилистические и количественные метрики качества кода.

Особенности программной архитектуры (рис.3-52): написан на Java, поставляется в виде jar-файла, запускается в режиме командной строки. В корневом каталоге проекта необходимо создать конфигурационный файл с именем, версией проекта, языком разработки и другими параметрами. Результаты анализа кода записываются в БД MySQL. Просмотр производится через web-интерфейс обычным браузером по *URL http:// localhost:9000*, в качестве эмулятора сайта выступает компонента *wrapper.exe*, которая запускается отдельным командным файлом.

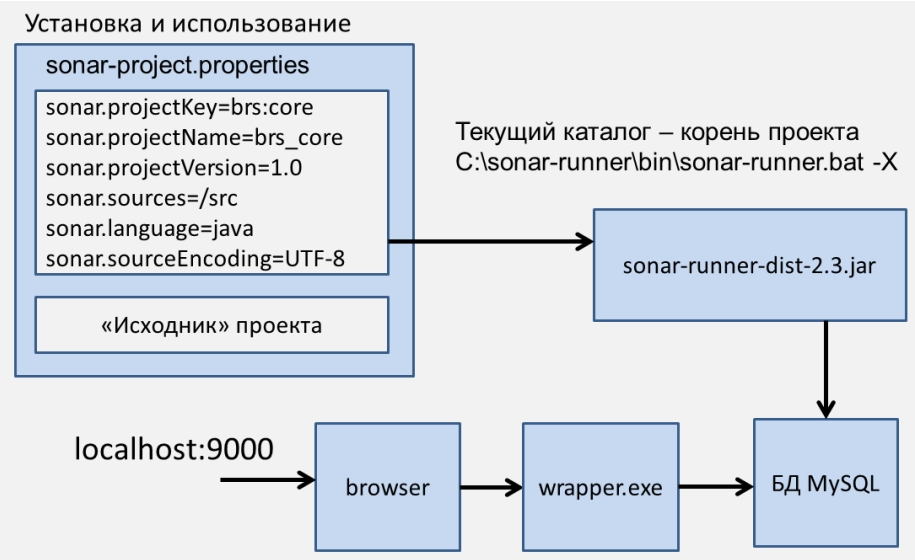


Рис.3-52. Программная архитектура SonarQube

Функционал продукта: собирает статистику по стилистике и количественным метрикам кода для всех компонент - проект, пакет, класс, метод, оценивает его качество (рис.3-53, 3-54):

- объем кода по классам, пакетам, методам в SLOC, документируемость, комментарии;
- поиск дублирования кода - *копипаст*;
- собственная мера оценки сложности кода;
- около 900 правил кодирования: стилистика, минимизирующая ошибки, стандарты оформления кода, предупреждение технологических дефектов и ошибок;
- оценка времени исправления стилистики.

Рис.3-53. Основные характеристики, оценка качества и статистика кода в SonarQube

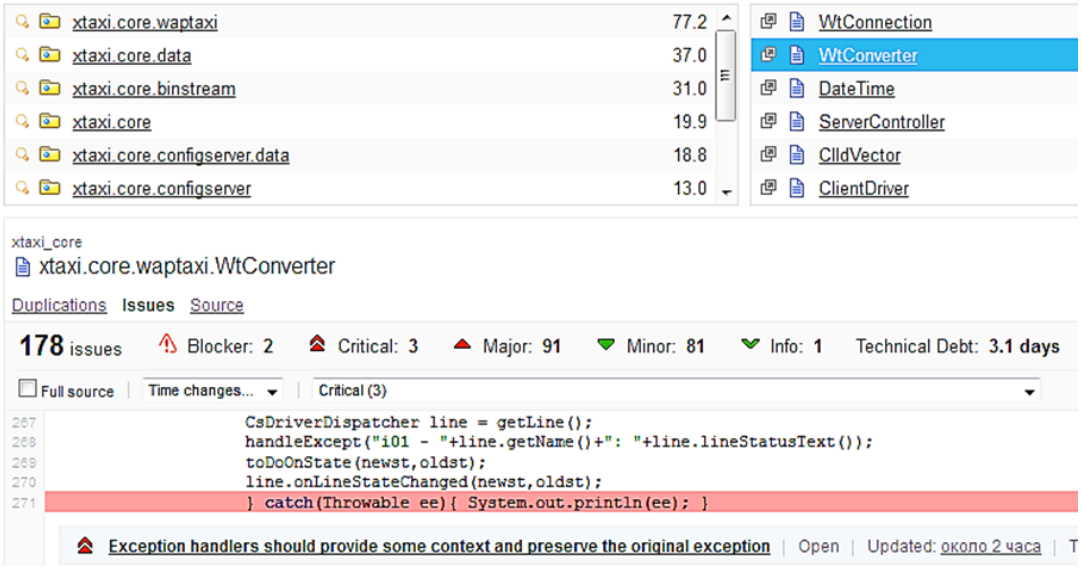
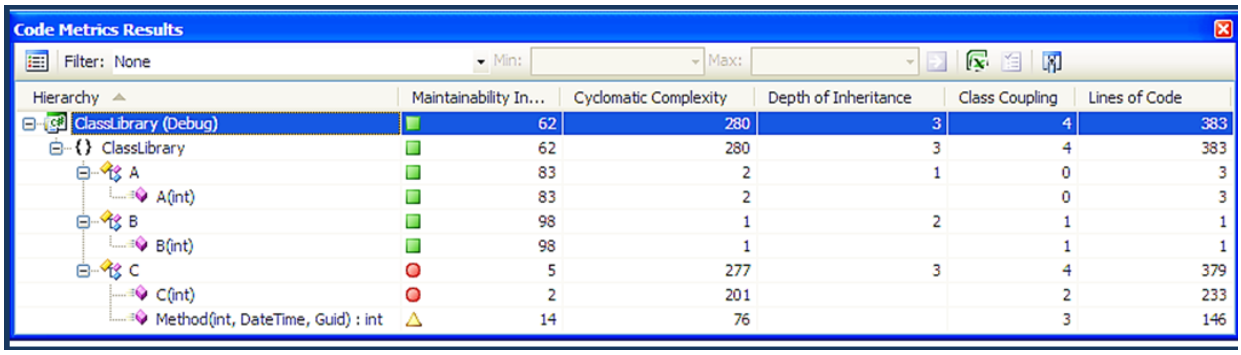


Рис.3-54. Контроль стилистики оформления кода

Средство оценки качества кода в MS Visual Studio

Встроенные средства MS Visual Studio вычисляют комплексный показатель качества кода *Maintainability Index* (рис.3-55).



Hierarchy	Maintainability In...	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
ClassLibrary (Debug)	62	280	3	4	383
ClassLibrary	62	280	3	4	383
A	83	2	1	0	3
A(int)	83	2		0	3
B	98	1	2	1	1
B(int)	98	1		1	1
C	5	277	3	4	379
C(int)	2	201		2	233
Method(int, DateTime, Guid) : int	14	76		3	146

Рис.3-55. Комплексный показатель качества кода в MS Visual Studio

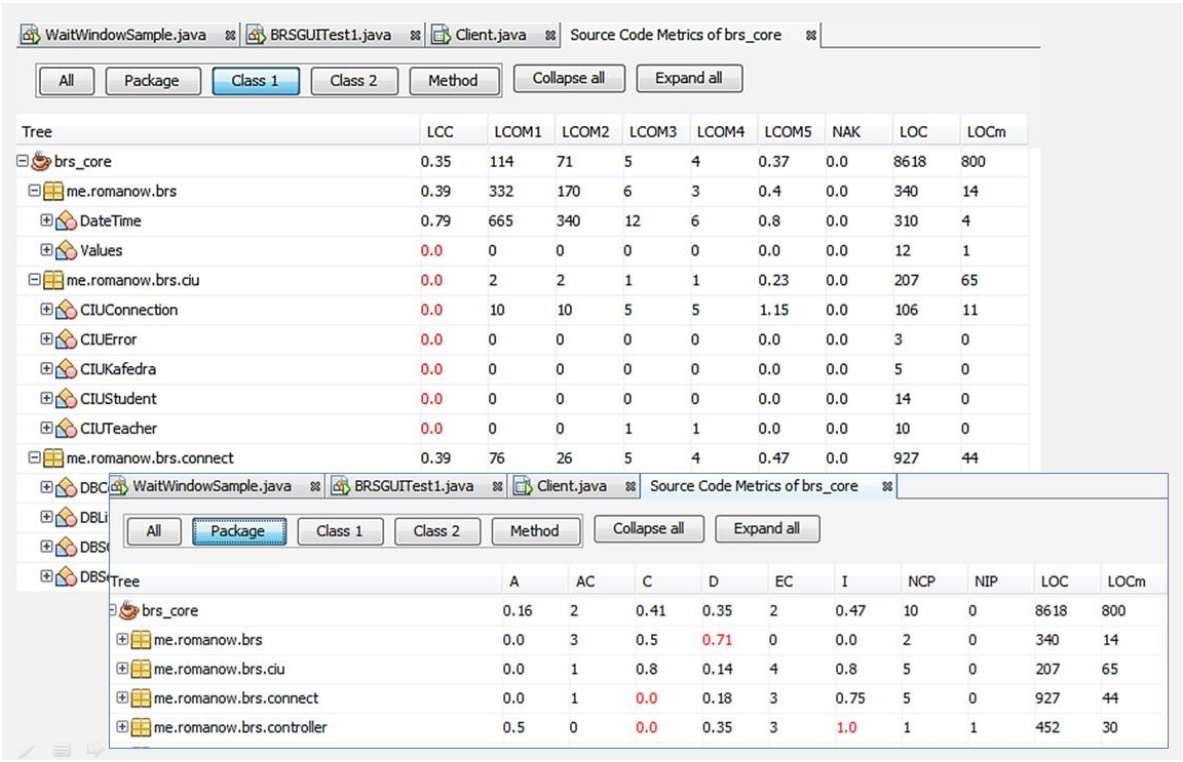
Показатель вычисляется по классам и методам по формуле:

MI = 171 - 5.2 * ln(HV) - 0.23 * CC - 16.2 * ln(LoC)) * 100 / 171, где

- **HV** (*Halstead Volume),* вычислительная сложность метода (класса);
- **CC** (*Cyclomatic Complexity*) – цикломатическая сложность кода;
- **LoC (***Lines of Code*) – количество строк кода, исключая пустые строки, комментарии, строки со скобками, объявление типов и пространств имен.

Плагины оценки кода в NetBeans и IntelliJ IDEA

Плагин, встраиваемый в Java NetBeans [3-11], позволяет получить 33 метрики качества кода для проекта, пакета и класса и экспортировать их в Excel (рис.3-56).



Tree	LCC	LCOM1	LCOM2	LCOM3	LCOM4	LCOM5	NAK	LOC	LOCm
brs_core	0.35	114	71	5	4	0.37	0.0	8618	800
me.romanow.brs	0.39	332	170	6	3	0.4	0.0	340	14
DateTime	0.79	665	340	12	6	0.8	0.0	310	4
Values	0.0	0	0	0	0	0.0	0.0	12	1
me.romanow.brs.ciu	0.0	2	2	1	1	0.23	0.0	207	65
CIUConnection	0.0	10	10	5	5	1.15	0.0	106	11
CIUError	0.0	0	0	0	0	0.0	0.0	3	0
CIUKafedra	0.0	0	0	0	0	0.0	0.0	5	0
CIUStudent	0.0	0	0	0	0	0.0	0.0	14	0
CIUTeacher	0.0	0	0	1	1	0.0	0.0	10	0
me.romanow.brs.connect	0.39	76	26	5	4	0.47	0.0	927	44

Tree	A	AC	C	D	EC	I	NCP	NIP	LOC	LOCm
brs_core	0.16	2	0.41	0.35	2	0.47	10	0	8618	800
me.romanow.brs	0.0	3	0.5	0.71	0	0.0	2	0	340	14
me.romanow.brs.ciu	0.0	1	0.8	0.14	4	0.8	5	0	207	65
me.romanow.brs.connect	0.0	1	0.0	0.18	3	0.75	5	0	927	44
me.romanow.brs.controller	0.5	0	0.0	0.35	3	1.0	1	1	452	30

Рис.3-56. Метрики качества кода на Java в NetBeans

Аналогичный плагин для IntelliJ IDEA, позволяет экспортировать полученные метрики в XML-формате (рис.3-57).

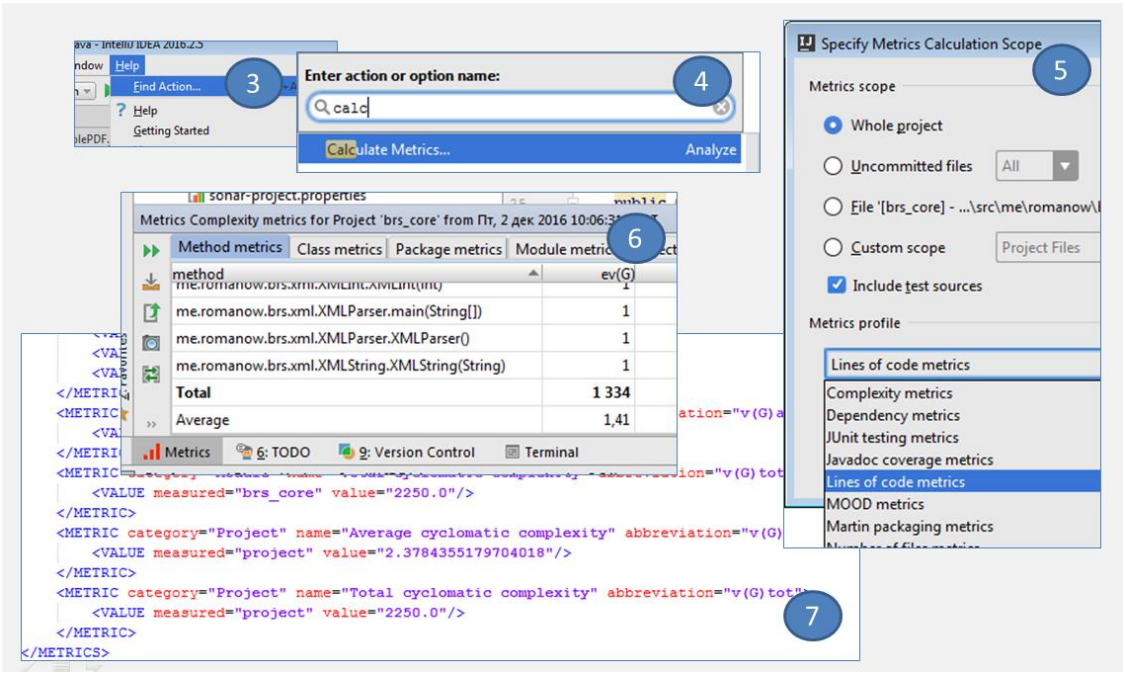


Рис.3-57. Метрики качества кода на Java в IntelilJ IDEA

Online-оценка кода - codenforcer

Коммерческий облачный сервис *codenforcer* [3-12] предоставляет услуги по метрическому сопровождению проектов с исходным текстом на различных языках программирования.. Проект должен быть размещен в репозитории на *github*, с которого он скачивается и анализируется. Предоставляемый сервис:

- нормализованные объектно-ориентированные метрики: 7 метрик для пакета, 2 – для класса;
- рекомендации по исправлению кода;
- оценка стилистики кода;
- поддержка разработки документации к проекту.

В качестве основного набора используются метрики Мартина (рис.3-58): сцепление, центробежное и центростремительное сцепление, нестабильность, абстрактность, расстояние до главной последовательности – все они характеризуют внешние связи (сцепление) классов в пакетах. Также используется метрика Чидамбера и Кемерера – недостаток связности в методах (LCOM) и ряд других:

- *относительная связность* – доля автономных классов, в которых нет обращений вне пакета;
- *ассоциация между классами* - количество случаев использования данным классом элементов классов - свойств, методов, констант, перечислений, нормированное количеством использования собственных элементов класса.

#	Metrics	Application	Assembly	Namespace	Package	Class	Interface	Structure	Enumeration
1	Coupling								
2	Afferent Coupling								
3	Efferent Coupling								
4	Instability								
5	Relational Cohesion								
6	Distance from the Main Sequence								
7	Abstractness								
8	Association Between Classes								
9	Cohesion of LCOM								
10	Cohesion ofLCOM HS								
11	Modularity								
12	OOP Level For Types								
13	OOP Level For Methods								
14	OOP Level For Fields								

- available in codEnforcer metrics
- metrics under development
- metric applicable for Java, C++, C# and PHP
- specific for Java
- specific for C#
- specific for PHP
- specific for C++

Рис.3-58. Виды поддерживаемых метрик

Собранные нормализованные метрики для компонент проекта визуализируются с помощью радиальных диаграмм, даются рекомендации по исправлению кода (рис.3-59).

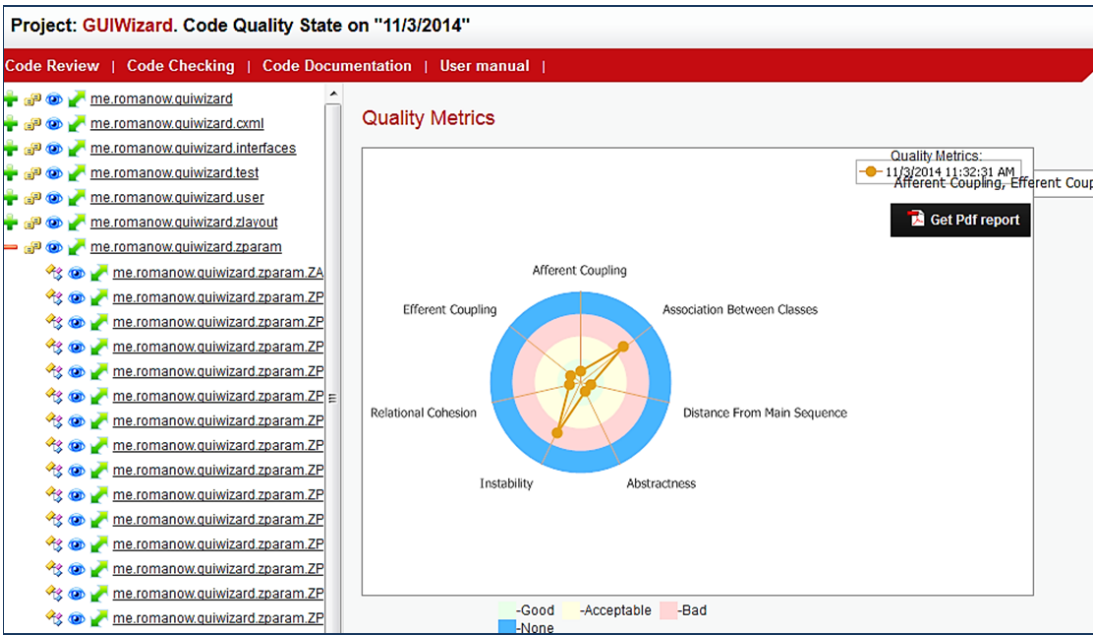


Рис.3-59. Радиальная диаграмма для нормализованных метрик пакета