ПРАКТИЧЕСКАЯ РАБОТА

Измерение и анализ эксплуатационных характеристик качества программного обеспечения

Содержание: название, формулировка темы и цели, время выполнения, необходимое обеспечение, теоретическая часть, практическая часть, технология выполнения, описание заданий, рефлексия самоконтроля, контрольные вопросы

Время выполнения: 2 ч.

Материально-техническое обеспечение: персональный компьютер (системный блок, монитор, клавиатура, мышь)

Программное обеспечение: ОС Windows, интегрированная среда разработки, офисные программы Microsoft Office.

Учебно-методическое обеспечение: конспект лекций, описание работы, встроенная справочная система ОС Windows и сред разработки.

Порядок выполнения практической работы: изучить теоретический материал, выполнить задания, составить отчёт с выводами о проделанной работе, ответить на контрольные вопросы, сдать выполненную работу и отчёт.

Требования к отчёту: отчёт должен быть оформлен в соответствии с «ГОСТ 19.XXX ЕСПД».

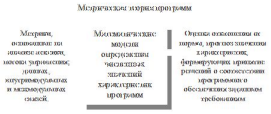
Цель: приобретение навыков обнаружения вируса и устранения последствий его влияния.

Теоретическая часть

Критерий качества

Критерий качества характеризует степень, в которой программе присуще оцениваемое свойство. Критерии качества включают следующие характеристики: экономичность, документированность, гибкость, модульность, надёжность, обоснованность, тестируемость, ясность, точность, модифицируемость, эффективность, легкость сопровождения и т. д. Для измерения характеристик и критериев качества используют метрики.

Метрика программного обеспечения



Метрика программного обеспечения (англ. Software metric) - это некая мера определенного свойства программного обеспечения или же его спецификаций. Как известно, мера - это средство измерения. Важно понять, что мера - это числовое значение. Таким образом, метрика программного обеспечения будет показывать некое числовое значение определенного свойства ПО. Ни одной универсальной метрики не существует. Любые контролируемые метрические характеристики программы должны контролироваться либо в зависимости друг от друга, либо в зависимости от конкретной задачи, кроме того, можно применять гибридные меры, однако они так же зависят от более простых метрик и также не могут быть универсальными. Строго говоря, любая метрика - это лишь показатель, который сильно зависит от языка и стиля программирования, поэтому ни одну меру нельзя возводить в абсолют и принимать какие-либо решения, основываясь только на ней. В исследовании метрик ПО различают два основных направления: поиск метрик, характеризующих наиболее специфические свойства программ. Группы метрик:

Метрики, оценивающие отклонение от нормы характеристик исходных проектных материалов, устанавливают полноту заданных технических характеристик исходного кода.
Метрики, позволяющие прогнозировать качество разрабатываемого ПО заданы на множестве возможных вариантов решений поставленной задачи и их реализации и определяют качество ПО, которое будет достигнуто в итоге.
Метрики, по которым принимается решение о соответствии конечного ПО заданным требованиям, позволяют оценить соответствие разработки заданным требованиям.

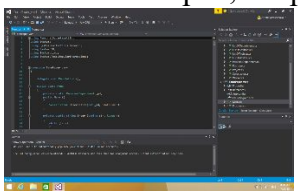
Метрические шкалы

В зависимости от характеристик и особенностей применяемых метрик им ставятся в соответствие различные измерительные шкалы. Номинальной шкале соответствуют метрики, классифицирующие программы на типы по признаку наличия или отсутствия некоторой характеристики без учета градаций. Порядковой шкале соответствуют метрики, позволяющие ранжировать некоторое характеристики путем сравнения с опорными значениями, т.е. измерение по этой шкале фактически определяет взаимное положение конкретных программ. Интервальной шкале соответствуют метрики, которые показывают не

только относительное положение программ, но и то, как далеко они отстоят друг от друга. Относительной шкале соответствуют метрики, позволяющие не только расположить программы определенным образом и оценить их положение относительно друг друга, но и определить, как далеко оценки отстоят от границы, начиная с которой характеристика может быть измерена.

Метрики программного обеспечения в Visual Studio

Стоит заметить сразу, что метрики подвергаются критике. Это, как минимум, поверхностно и неточно. Сперва, откроем проект для изучения:



Далее, мы можем видеть вкладку Analyze:

В этой вкладке мы видим «Calculate Code Metrics for ...». Это нам и нужно. Разница лишь в том, что будет анализироваться. Или же выбранные проекты в Solution Explorer, или же сразу весь Solution. После нажатия придется немного подождать. Время зависит от конфигурации Вашего компьютера. Когда анализ будет завершен, Вы увидите внизу окно:

Здесь будет видна иерархия всего Solution. В моем случае это отдельная dll библиотека и проект. Когда развернем библиотеку, мы увидим следующий уровень иерархии, и т. д.:

Maintainability Index - это комплексный показатель качества кода. Эта метрика рассчитывается по следующей формуле:

MI = MAX(0, (171 - 5.2 * ln(HV) - 0.23 * CC - 16.2 * ln(LoC)) * 100 / 171)	HV - Halstead Volume, вычислительная сложность. Чем больше операторов, тем больше значение этой метрики; CC - Cyclomatic Complexity; LoC - количество строк кода
Cyclomatic Complexity	Показывает структурную сложность кода. Иными словами, количество различных ветвей кода. Считается на основе операторов в Вашем коде, строя графы переходов от одного оператора к другому. К примеру, оператор if-else увеличит эту метрику, потому что здесь будут разные ветви выполнения
Depth of Inheritance	Глубина наследования. Для каждого класса эта метрика показывает, насколько глубоко он в цепочке наследования
Class Coupling	Указывает на зависимость классов друг от друга. Проект с множеством зависимостей очень трудно и дорого поддерживать
Lines of Code	Количество строк кода. Напрямую используется редко. В наши дни, с множеством разнообразных как подходов к программированию, так и языков, эта метрика дает нам мало полезной информации. Если брать во внимание отдельный метод, то можно разбить его на несколько методов поменьше

Изначально стоит обращать внимание на Maintainability Index. Старайтесь придерживать его около 70-90. Это значительно облегчит сопровождения кода как Вами, так и другими программистами. Иногда стоит оставить его на уровне 50-60, так как переписать некоторые участки кода бывает очень затратным. Оценивайте здраво как код, так и Ваши возможности с затратами. Стоит также уделить много внимания Class Coupling. Эта метрика должна быть как можно меньшей. Ведь она так же способствует поддержке кода. Для оптимизации возможно придется пересматривать дизайн проекта и некоторые архитектурные решения. Теперь стоит уделить внимание Cyclomatic Complexity. Эта метрика показывает сложность кода, а это так же влияет на поддержку кода в будущем. Иногда приходится переписывать куски кода, которые писали до Вас другие люди, так как Вы просто не можете понять, что, как и зачем в этом методе. Конечно, этому еще способствует стиль кода и идея, но не забывайте о Cyclomatic Complexity при рефакторинге. Вы, наверняка, заметили, что мы использовали на практике не все метрики, но они могут быть частью остальных, как в случае с Maintainability Index. Но стоит понимать, что оценивать качество работы программиста, исходя из метрик, нельзя. Это очень неточно и поверхностно. Иногда просто нет другого способа решения задачи, а иногда это бывает затратным. Также есть человеческий фактор, о котором не

стоит забывать. Метрики бывают искаженными, ведь программист может стремится написать не эффективное и правильно решение, а оптимизировать показатели этих же метрик.

Практическая часть

Метрическая оценка качества программного обеспечения:

- Количественные метрики (метрики размера программ);
- Метрики сложности потока управления программы;
- Метрики сложности потока управления данными;
- Метрики сложности потока управления и данных программы;
- Объектно-ориентированные метрики;
- Метрики надежности;
- Гибридные метрики.

КОЛИЧЕСТВЕННЫЕ МЕТРИКИ (МЕТРИКИ РАЗМЕРА ПРОГРАММ)

Оценки первой группы наиболее просты и, очевидно, поэтому получили широкое распространение. Традиционной характеристикой размера программ является количество строк исходного текста. Под строкой понимается любой оператор программы, поскольку именно оператор, а не отдельно взятая строка является тем интеллектуальным "квантом" программы, опираясь на который можно строить метрики сложности ее создания. Непосредственное измерение размера программы, несмотря на свою простоту, дает хорошие результаты. Конечно, оценка размера программы недостаточна для принятия решения о ее сложности, но вполне применима для классификации программ, существенно различающихся объемами. При уменьшении различий в объеме программ на первый план выдвигаются оценки других факторов, оказывающих влияние на сложность. Таким образом, оценка размера программы есть оценка по номинальной шкале, на основе которой определяются только категории программ без уточнения оценки для каждой категории.

Размерно-ориентированные метрики

Размерно-ориентированные метрики прямо измеряют программный продукт и процесс его разработки. Основываются такие метрики на LOC-оценках. Этот вид метрик косвенно измеряет программный продукт и процесс его разработки. Вместо подсчета LOC-оценок при этом рассматривается не размер, а функциональность или полезность продукта. Наибольшее распространение в практике создания программного обеспечения получили размерно-ориентированные метрики.

Формулы размерно-ориентированных метрик:

Общие трудозатраты (в человеко-месяцах, человеко-часах)
Объем программы (в тысячах строках исходного кода -LOC)
Стоимость разработки
Объем документации
Ошибки, обнаруженные в течение года эксплуатации
Количество людей, работавших над изделием
Срок разработки

На основе этих данных обычно подсчитываются простые метрики для оценки производительности труда (KLOC/человеко-месяц) и качества изделия. Эти метрики не универсальны и спорны, особенно это относится к такому показателю как LOC, который существенно зависит от используемого языка программирования.

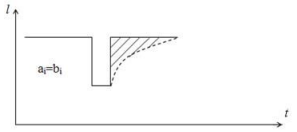
Метрика флуктуации длин программной документации

Программная документация, на самом деле, неоднородна по своему содержанию. С одной стороны, встречаются документы, сильно зависящие от текста программы. С другой стороны, существует документация, отражающая постановку задачи, имеются спецификации и ведомости, а также материалы, относящиеся к испытаниям программ. Целесообразно использовать данную метрику для оценки качества, прежде всего эксплуатационной документации. Исключая формуляр и ведомость эксплуатационных документов. Отдельно оцениваются техническое задание с пояснительной запиской и описание программ.

Остальные документы чаще всего отражают изменения в перечисленных документах или неявно вызывают эти изменения. Представляют интерес сам подход, дающий возможность оценивать динамику характеристик. Ведь если вместо текущей длины документа, использовать какую - либо иную характеристику, например, длину программы, то все исходные предпосылки останутся в силе. Измеряя в определенные моменты времени значения исследуемой характеристики, мы можем делать выводы об ее динамике. По этим данным, выявляя резкие скачки в переходном процессе, можно судить, например, о потери стабильности используемого ПО и о целесообразности.

Исходным является предположение о том, что чем меньше изменений и корректировок вносится в программную документацию, тем более четко были сформулированы решаемые задачи на всех этапах работ. Неточности и неясности при создании послужат причиной увеличения количества корректировок и изменений в документации. И, напротив, демпфированный (гашение нежелательных колебаний) переходный процесс с немногочисленными изменениями длин документов - естественное следствие хорошо обдуманной идеи, хорошо проведенного анализа, проектирования и ясной структуры программ. Эти взаимосвязи и являются основными для данного метода оценки, суть которого состоит в следующем. Предположим, что документация изменяется в декретные моменты времени ti, i=1, 2, … , n. Тогда в любой момент t_i текущая длина документа l_i может быть определена как: $l_i = l_i - 1 + a_i - b_i$; где $l_0=0$, (l_i-1) - длина документа в предыдущий момент времени; a_i - добавляемая часть документа; bi - исключаемая часть документа. Далее вводится d_i, представляющее собой отклонение текущей длины документа l_i от оконечного значения l_n: $d_i=l_n - l_i$. Затем рассчитывается интервал по модулю этого отклонения на интервале от t_i до t_n, представленный в виде суммы:

$H_n = \sum_{i=1}^{n-1} \left| d_i \right| \left(t_{i+1} - t_i \right)$. Значение H_n представляет собой оценку переходного процесса для интервала времени от t_1 до t_n. Однако H_n не учитывает изменений типа a_i = bi, хотя они, бесспорно, влияют на ход дальнейшего процесса. Чтобы отразить влияние изменений такого рода, называемых в дальнейшем импульсными, вводится функция экспоненциальная, отражающая функцию отклика. Пример импульсного изменения длины программного документа:



Заштрихованная область представляет собой дополнение к оценке H, отражающие влияние импульсного изменения длины документов и вычисляемое как:

$$\int_{t_i}^{\infty} a_i e^{-\alpha^{-1}(t-t_i)} dt = \alpha a_i = \alpha b_i, \alpha > 0.$$

Т. о., оценка длины документа пропорциональна значению импульсного изменения длины ai=bi с коэффициентом пропорциональности α. В принципе импульсное изменение длины документа присутствует и при $a_i \neq b_i$. Формула метрики флуктуации длин программной документации:

$H_n^{'} = \sum_{i=1}^{n-1} [\left| d_i \right| (t_{i+1} - t_i) + \alpha c_i],$	$c_i = \min \{a_i, b_i\}.$

Используя конечное значение длины документа, можно записать: $H_n^{''} = H_n^{'} / l_n.$

LOC-метрика

Количество строк исходного кода (Lines of Code - LOC, Source Lines of Code - SLOC) является наиболее простым и распространенным способом оценки объема работ по проекту. Изначально данный показатель возник как способ оценки объема работы по проекту, в котором применялись языки программирования, обладающие достаточно простой структурой: "одна строка кода = одна команда языка". Также давно известно, что одну и ту же функциональность можно написать разным количеством строк, а если возьмем язык высокого уровня (C++, Java), то возможно и в одной строке написать функционал 5-6 строк - это не проблема. И это было бы полбеды: современные средства программирования сами генерируют тысячи строк кода на пустяковую операцию. Потому метод LOC является только оценочным методом (который надо принимать к сведению, но не опираться в оценках) и никак не обязательным. В зависимости от того, каким образом учитывается сходный код, выделяют два основных показателя SLOC:

1.	Количество "физических" строк кода - SLOC (используемые аббревиатуры LOC, SLOC, KLOC, KSLOC, DSLOC) - определяется как общее число строк исходного кода, включая комментарии и пустые строки (при измерении показателя на количество пустых строк, как правило, вводится ограничение - при подсчете учитывается число пустых строк, которое не превышает 25% общего числа строк в измеряемом блоке кода)
2.	Количество "логических" строк кода - SLOC (используемые аббревиатуры LSI, DSI, KDSI, где "SI" - source instructions) - определяется как количество команд и зависит от используемого языка программирования. В том случае, если язык не допускает размещение нескольких команд на одной строке, то количество "логических" SLOC будет соответствовать числу "физических", за исключением числа пустых строк и строк комментариев. В том случае, если язык программирования поддерживает размещение нескольких команд на одной строке, то одна физическая строка должна быть учтена как несколько логических, если она содержит более одной команды языка

Для метрики SLOC существует большое число производных, призванных получить отдельные показатели проекта, основными среди которых являются:

Число пустых строк
Число строк, содержащих комментарии
Процент комментариев (отношение строк кода к строкам комментария)
Среднее число строк для функций (классов, файлов)
Среднее число строк, содержащих исходный код для функций (классов, файлов)
Среднее число строк для модулей
Среднее число строк для программы (с модулями: $F = \sum F_i/i$, без - F)

Метрика стилистики и понятности программ

Иногда важно не просто посчитать количество строк комментариев в коде и просто соотнести с логическими строчками кода, а узнать плотность комментариев. Примеры: код сначала был документирован хорошо, затем - плохо; шапка функции или класса документирована и комментирована, а код - нет. Суть метрики проста: код разбивается на N-равные куски и для каждого из них определяется F_i.

Метрика стилистики и понятности фрагментов программы: $F_i = SIGN (N_{комм.i} / N_i - 0,1)$	i - количество равных фрагментов; $N_{комм.i}$ - количество комментариев в фрагменте; N_i - количество строк кода в фрагменте

Формула метрики стилистики и понятности программы:

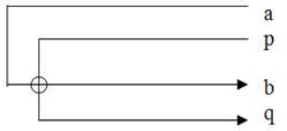
Метрика стилистики и понятности программы: $F = \sum F_i$	F_i - значения метрики стилистики и понятности фрагментов программы

Метрика «Подсчёт точек пересечения»

Авторы метрики «Подсчёт точек пересечения» - Вудвард и Хенел. Метрика ориентирована на анализ программ, при создании которых использовалось неструктурное кодирование (Assembler и Fortran). Вводя эту метрику, ее авторы стремились оценить скорее взаимосвязи между физическими местоположениями управляющих переходов, чем просто их последовательность в программе. Удобство метрики в том, что ее легко представить численными соотношениями. Структурное кодирование предполагает использование ограниченного множества управляющих структур в качестве первичных элементов любой программы. В классическом структурном кодировании оперируют только тремя такими структурами: следованием операторов, развилкой из операторов, циклом над оператором. Все эти разновидности изображаются простейшими планарными графами программ. По правилам структурного кодирования любая программа составляется путем упомянутых структур или помещения одной структуры в другую. Эти операции не нарушает планарности графа всей программы. В графе программы, где каждому оператору соответствует вершина, то есть не исключены линейные участки, при передаче управления от вершины a и b номер оператора a = min(a, b), а номер оператора b = max(a, b). Точка пересечения дуг появляется, если

min(a, b) < min(p, q) < max(a, b) & max(p, q) > max(a, b); min(a, b) < max(p, q) < max(a, b) & max(p, q) < max(a, b).

Иными словами, точка пересечения дуг графа программы возникает в случае выхода управления за пределы пары вершин (a, b):



Количество точек пересечения дуг графа программы дает характеристику неструктурированности программы. Эта характеристика хорошо согласуется с определением Маккейба для отклонений от структурного построения программ.

Метрики Холстеда

При применении метрик Холстеда частично компенсируются недостатки, связанные с возможностью записи одной и той же функциональности разным количеством строк и операторов. Метрики Холстеда основаны на подсчете некоторых единиц в коде программы и основаны на показателях:

n_1	Число уникальных операторов программы, включая символы-разделители, имена процедур и знаки операций (словарь операторов)
n_2	Число уникальных операндов программы (словарь операндов)
N_1	Общее число операторов в программе
N_2	Общее число операндов (элемент данных, над которым выполняется операция) в программе
n_1'	Теоретическое число уникальных операторов
n_2'	Теоретическое число уникальных операндов
n^*	Теоретический словарь программы

Учитывая введенные обозначения, можно определить:

$n=n_1+n_2$	Словарь программы
$N=N_1+N_2$	Длина программы
$n'=n_1'+n_2'$	Теоретический словарь программы
$N'= n_1*\log_2(n_1) + n_2*\log_2(n_2)$	Теоретическая длина программы (для стилистически корректных программ отклонение N от N' не превышает 10%)
$V=N*\log_2 n$	Объем программы
$V'=N'*\log_2 n'$	Теоретический объем программы
$L=V'/V$	Уровень качества программирования, для идеальной программы L = 1
$L'= (2*n_2)/ (n_1*N_2)$	Уровень качества программирования, основанный лишь на параметрах реальной программы без учета теоретических параметров
EC=V/(L')2	Сложность понимания программы
D=1/ L'	Трудоемкость кодирования программы
$y' = V/ D_2$	Уровень языка выражения
I=V/D	Информационное содержание программы, данная характеристика позволяет определить умственные затраты на создание программы
$E=N' * \log_2(n/L)$	Оценка необходимых интеллектуальных усилий при разработке программы, характеризующая число требуемых элементарных решений при написании программы

Метрика длины по Холстеду

$\tilde{N}=F(\eta_1, \eta_2)$ определяет вид функции, где $\tilde{N}$ - теоретическое (предсказываемое) значение.

Нижнюю границу функции F можно найти из условия обязательности использования всех элементов словаря η, а верхнюю - из отсутствия дубликатов в строках программы (приняв количество строк = N/η): $\eta < N < \eta^{\eta+1}$. Учитывая, что операторы и операнды чередуются, верхняя граница уточняется: $\tilde{N} \leq \eta \cdot \eta_1^{\eta 1} \cdot \eta_2^{\eta 2}$. В конечном итоге можно прийти к уравнению длины, которое можно рассматривать как первую гипотезу Холстеда: Методология проверки гипотезы длины заключается в построении графика $\tilde{N} = f(N)$. Если точки

группируются около средней линии $\tilde{N} = N$, то это указывает на статистическую правильность гипотезы. Каждая точка графика получалась из анализа (вычисления $\tilde{N}$ и N) по "хорошим» алгоритмам. Исследовались около 400 программ на Фортране с длинами до 10000 и на языке PL/I (120 программ с длиной до 100000). Коэффициент корреляции величин N и $\tilde{N}$ составил 0,95 - 0,98 (коэффициент корреляции определяется как тангенс угла наклона прямой, аппроксимирующей экспериментальные данные по методу наименьших квадратов):

Формула метрики длины по Холстеду:

$$\tilde{N} = \eta_1 \cdot \log_2\eta_1 + \eta_2 \cdot \log_2\eta_2 \quad (1)$$

Метрика объема по Холстеду

Важной характеристикой алгоритма является его размер. При переводе данного алгоритма с одного языка на другой его размер меняется. Аналогично алгоритмы, написанные на одном языке, имеют разные размеры. Изучение всех этих изменений количественными методами требует, чтобы размер был измеримой величиной.

Кроме того, метрическая характеристика размера должна быть применима к широкому кругу возможных языков без потери общности или объективности и, следовательно, не зависеть от набора символов, требуемых для выражения алгоритма. Тогда на ней не будет отражаться число символов, практически используемых при представлении операторов или имен операндов.

Решение этой проблемы связано с тем, что в любом конкретном случае можно определить абсолютный минимум длины представления самого длинного оператора или имени операнда, если ее выражать в двоичных разрядах, или битах. Минимальная длина зависит только от числа элементов в словаре η. Например, словарь из восьми различных элементов требует восемь различных знаков, или, что то же самое, восемь возможных значений двоичного числа, состоящего из трех разрядов. В общем случае $\log_2\eta$ есть минимальная длина (в битах) всей программы. Тогда может быть определена соответствующая метрическая характеристика размера любой реализации какого-либо алгоритма, называемая объемом V. Очевидно, что если алгоритм переводится с одного языка на другой, то его объем меняется. Например, при переводе алгоритма с Фортрана в машинный код какой-либо конкретной машины его объем увеличится. С другой стороны, алгоритм может быть выражен на более развитом языке, нежели тот, на котором он написан, и в этом случае его объем уменьшится.

Формула метрики объема (в битах) по Холстеду:

$$V = N \log_2\eta \quad (2)$$

, где N - длина реализации; η - ее словарь

Метрика потенциального объема по Холстеду

Выражение алгоритма в наиболее сжатой форме предполагает прежде всего существование языка, в котором требуемая операция уже определена или реализована, возможно, в виде процедуры или подпрограммы. Для реализации данного алгоритма в таком языке требуются лишь имена операндов для его аргументов и результатов.

Обозначив соответствующие программные параметры наиболее сжатой формы алгоритма звездочками, из уравнения (2) получим, что минимальный (или потенциальный) объем: $V^* = (N^*_1+N^*_2)\log_2(\eta^*_1+\eta^*_2)$. Но в минимальной форме ни операторы, ни операнды не требуют повторений, поэтому: $^*_1=\eta^*_1$ и $N^*_2=\eta^*$, что дает нам: $V^* = (\eta^*_1+\eta^*_2)\log_2(\eta^*_1+\eta^*_2)$. Кроме того, известно минимально возможное число операторов η^*_1 для любого алгоритма. Каждый алгоритм должен включать один оператор для имени функции или процедуры и один в качестве символа присваивания или группировки, т. е. $\eta^*_1 = 2$. Потенциальный объем V* любого алгоритма не должен зависеть от языка, на котором он может быть выражен. Если η^*_2 расценивается как число единых по смыслу (не избыточных) операндов, то V* оказывается наиболее полезной мерой содержания алгоритма.

Формула метрики потенциального объема по Холстеду:

$V^* = (2+ \eta^*_2)\log_2(2+\eta^*_2)$, где η^*_2 - число различных входных и выходных параметров (3)

Метрика уровня реализации по Холстеду

После введения понятия объема в теории вводится понятие уровня реализации (уровень качества программирования) L:

$L = V^*/V, \quad (L \leq 1)$ (4)

Нередко требуется определить уровень программы непосредственно из реализации, не зная, чему равен ее потенциальный объем V^* и не ссылаясь на возможное обращение к ней в виде вызова процедуры. Это можно сделать, рассмотрев отдельно влияние операторов и операндов на уровень программы. Разумно предположить, что чем больше число простых операторов, используемых в реализации, тем ниже ее уровень. Но наименьшее возможное число простых операторов равно двум, и эти два оператора суть символ функции и оператор присваивания или группировки, т.е. $\eta^*_1 = 2$. С другой стороны, число простых операторов беспредельно, поскольку на него нет ограничений ни в каком языке, в котором разрешено определение новых вложенных процедур, подпрограмм или переходов к помеченным участкам. Отсюда получаем следующее отношение для операторов: $L \approx \eta^*_1/ \eta_1$. Операнды же не дают однозначного минимума по всем возможным алгоритмам, поэтому к ним требуется иной подход. В этом случае достаточно заметить, что всякое повторение имени операнда является указанием на более низкий уровень реализации. Этот эффект можно измерить, взяв отношение числа простых операндов к общему числу операндов: $L \approx \eta_2/N_2$. Объединяя два предыдущих уравнения, приходим к уравнению уровня метрики уровня реализации программы по Холстеду. Уровень, измеряемый уравнением метрики уровня реализации по Холстед, служит аппроксимацией уравнения (4). На самом деле, конечно, его можно взять в качестве альтернативного определения уровня программы.

Формула метрики уровня реализации по Холстеду:

$L^\wedge = (\eta^*_1/ \eta_1)\cdot(\eta_2/N_2)$ (5)

Метрика интеллектуального содержания по Холстеду

Располагая характеристикой $L^\wedge$, Холстед вводит характеристику I, которую рассматривает как интеллектуальное содержание конкретного алгоритма, инвариантное по отношению к используемым языкам реализации:

$$I = L^\wedge \cdot V \qquad (6)$$

Однако, по мнению самого автора, термин интеллектуальность не совсем удачен. Преобразуя выражение (6) с учетом (4, 5), получаем:

$$I = L^\wedge \cdot V \approx L \cdot V = V^* \cdot V/V = V^*$$

Эквивалентность I и V* свидетельствует о том, что мы имеем дело с характеристикой информативности программы.

Метрика оценки необходимых интеллектуальных усилий по написанию программы по Холстеду

Введение характеристики I позволяет определить умственные затраты на создание программы. Процесс создания программы условно можно представить как: осмысление предложения известного алгоритма; запись предложения алгоритма в терминах используемого языка программирования, т. е. поиск в словаре языка соответствующей инструкции, ее смысловое наполнение и запись. Используя эту формализацию в методике Холстеда, можно сказать, что написание программы по заранее известному алгоритму есть Ñ-кратная выборка операторов и операндов из словаря программы η, причем число сравнений (по аналогии с алгоритмами сортировки) составит $\log_2(\eta)$. Если учесть, что каждая выборка-сравнение содержит, в свою очередь, ряд мысленных элементарных решений, то можно поставить в соответствие содержательной нагрузке каждой конструкции программы сложность и число этих элементарных решений. Количественно это можно характеризовать с помощью характеристики L, поскольку 1/L имеет смысл рассматривать как средний коэффициент сложности, влияющий на скорость выборки для данной программы. Тогда оценка необходимых интеллектуальных усилий по написанию программы может быть измерена как:

$E = Ñ * \log_2(\eta/L)$ (7)

Т. о., E характеризует число требуемых элементарных решений при написании программы. Однако следует заметить, что E адекватно характеризует лишь начальные усилия по написанию программ, поскольку при построении E не учитываются отладочные работы, которые требуют интеллектуальных затрат иного характера. Суть интерпретации этой характеристики состоит в оценке не затрат на разработку программы, а затрат на восприятие готовой программы. При этом вместо теоретической длины программы $\hat{N}$ используется ее реальная длина:

$$E' = N * \log_2(\eta/L) \qquad (8)$$

Характеристика E' введена исходя из предположения, что интеллектуальные усилия на написание и восприятие программы очень близки по своей природе. Однако если при написании программы стилистические погрешности в тексте практически не должны отражаться на интеллектуальной трудоемкости процесса, то при попытке понять такую программу их присутствие может привести к серьезным осложнениям. Преобразуя формулу (8) с учетом выражений (2) и (4), получаем оценку необходимых интеллектуальных усилий по написанию программы. Такое представление E', а соответственно и E, так как E=E', наглядно иллюстрирует целесообразность разбиения программ на отдельные модули, поскольку интеллектуальные затраты оказываются пропорциональными квадрату объема программы, который всегда больше суммы квадратов объемов отдельных модулей.

Формула метрики оценки необходимых интеллектуальных усилий по написанию программы по Холстеду:

$E = V^2/V^*$

ABC-метрика

ABC-метрика (Fitzpatrick) основана на подсчете присваиваний значений переменным (Assignment), явных передач управления за пределы области видимости, т. е. вызовов функций (Branch), илогических проверок (Condition). Мера записывается тройкой значений, например, ABC = <7, 4, 2>, но для оценки сложности программы вычисляется одно число, как квадратный корень из суммы квадратов A, B, C. Она может иметь нулевое значение для некоторых непустых программных единиц.

Формула ABC-метрики:

$ABC = \sqrt{(A^2 + B^2 + C^2)}$	A – количество присваиваний значений переменным; B - количество вызовов функций; C – количество логических проверок

МЕТРИКИ СЛОЖНОСТИ ПОТОКА УПРАВЛЕНИЯ ПРОГРАММ

Вторая наиболее представительная группа оценок сложности программ - метрики сложности потока управления программ. Как правило, с помощью этих оценок оперируют либо плотностью управляющих переходов внутри программ, либо взаимосвязями этих переходов. И в том и в другом случае стало традиционным представление программ в виде управляющего ориентированного графа G = (V, E), где V - вершины, соответствующие операторам, а E - дуги, соответствующие переходам.

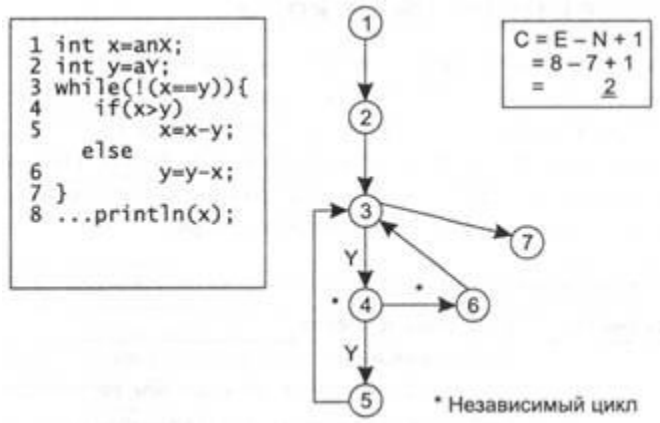
Метрика Мак-Кейба

Метрика Маккейба определяет минимальное количество тестовых прогонов программы, необходимых для тестирования всех ее ветвей (разветвлений).

Показатель цикломатической сложности является одним из наиболее распространенных показателей оценки сложности программных проектов. Метрика Маккейба определяет минимальное количество тестовых прогонов программы, необходимых для тестирования всех ее ветвей (разветвлений). Данный показатель был разработан ученым Мак-Кейбом в 1976 г., относится к группе показателей оценки сложности потока управления программой и вычисляется на основе графа управляющей логики программы (control flow graph). Данный граф строится в виде ориентированного графа, в котором вычислительные операторы или выражения представляются в виде узлов, а передача управления между узлами - в виде дуг. Показатель цикломатической сложности позволяет не только произвести оценку трудоемкости реализации отдельных элементов программного проекта и скорректировать общие показатели оценки длительности и стоимости проекта, но и оценить связанные риски и принять необходимые управленческие решения. Число компонентов связности графа можно рассматривать как количество дуг, которые необходимо добавить для преобразования графа в сильносвязный.

Сильносвязным называется граф, любые две вершины которого взаимно достижимы. Для графов корректных программ, т. е. графов, не имеющих недостижимых от точки входа участков 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 (как правило, P = 1).

Способы вычисления цикломатической сложности: вычисление цикломатической сложности C; вычислении количества замкнутых областей, образованных ребрами, которые нельзя разбить на меньшие области:



По сути C определяет число линейно независимых контуров в сильносвязном графе. Иначе говоря, цикломатическое число Мак-Кейба показывает требуемое количество проходов для покрытия всех контуров сильно связного графа или количество тестовых прогонов программы, необходимых для исчерпывающего тестирования по критерию "работает каждая ветвь". Цикломатическое число зависит только от количества предикатов, сложность которых при этом не учитывается. Например, имеется два оператора условия:

IF X>0 THEN X=A; ELSE X=0; IF ((X>0) AND (FLAG=1)) or ((X=0 AND (FLAG=0)) THEN X=A; ELSE X=0;

Оба оператора предполагают единственное ветвление и могут быть представлены одним и тем же графом. Очевидно, цикломатическое число будет для обоих операторов одинаковым, не отражающим сложности предикатов, что весьма существенно при оценке программ. Цикломатическое число Мак-Кейба показывает требуемое количество проходов для покрытия всех контуров сильносвязанного графа или количества тестовых прогонов программы, необходимых для исчерпывающего тестирования по принципу «работает каждая ветвь». Показатель цикломатической сложности может быть рассчитан для модуля, метода и других структурных единиц программы.

Модификации показателя цикломатической сложности:

- «Модифицированная» цикломатическая сложность - рассматривает не каждое ветвление оператора множественного выбора (switch), а весь оператор как единое целое;
- «Строгая» цикломатическая сложность - включает логические операторы;
- «Упрощенное» вычисление цикломатической сложности - предусматривает вычисление не на основе графа, а на основе подсчета управляющих операторов.

Формула вычисления цикломатической сложности по Мак-Кейбу:

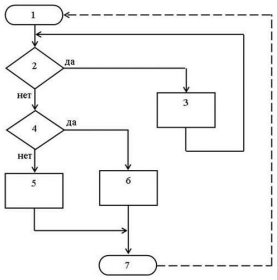
Z(G) = E - N + 2*P	N - число узлов (операторов программы) на графе управляющей логики; E - число ребер (ребро соединяет узел т с узлом п, если оператор п следует сразу за оператором m); P - число компонентов связности графа

В процессе автоматизированного вычисления показателя цикломатической сложности, как правило, применяется упрощенный подход, в соответствии с которым построение графа не осуществляется, а вычисление показателя производится на основании подсчета числа операторов управляющей логики (if, switch и т. д.) и возможного количества путей исполнения программы.

Упрощённая формула вычисления цикломатической сложности по Мак-Кейбу:

$Z(G)_{упр.}$ = E- N + 2	E - число ребер (ребро соединяет узел т с узлом п, если оператор п следует сразу за оператором m); N - число узлов (операторов программы) на графе управляющей логики

Пример № 1. Метрика Маккейба для программы, представленной схемой алгоритма. Действия, выполняемые блоками программы, в примере не показаны. Внутри каждого блока помещены их номера. Компонент связности графа обозначен штриховой дугой.

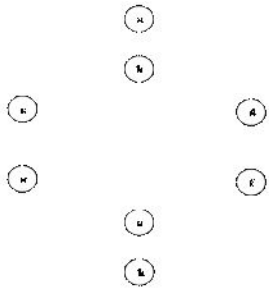


Число дуг E = 8, число вершин N = 7, P = 1. Цикломатическое число Мак-Кейба Z(G) = 8 – 7 +2 = 3. Значение метрики Мак-Кейба показывает, что в схеме алгоритма можно выделить три базисных независимых пути (называемых также линейно независимыми контурами):

Путь	Вариант № 1	Вариант № 2
1.	1 – 2 (нет) – 4 (нет) – 5 – 7	1 – 2 (да) – 3 – 2 (нет) – 4 (нет) – 5 –
2.	1 – 2 (да) – 3 – 2 (нет) – 4 (нет) – 5 – 7	1 – 2 (нет) – 4 (нет) – 5 – 7
3.	1 – 2 (нет) – 4 (да) – 6 – 7	1 – 2 (да) – 3 – 2 (нет) – 4 (да) – 6 – 7

Для тестирования совокупности базисных независимых путей исследуемой программы необходимо выполнить минимально три тестовых прогона.

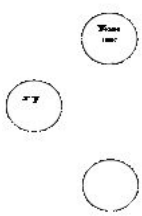
Пример № 2. Метрика Маккейба для программы, представленной схемой алгоритма. Действия, выполняемые блоками программы, в примере не показаны. Внутри каждого блока помещены буквы.



Число дуг E = 10, число вершин N = 8, P = 1. Цикломатическое число Мак-Кейба Z(G) = 10 – 8 +2 = 4.

№ п/п	Линейно независимые контуры сильносвязного графа
1.	a-b-c-g-e-h-a
2.	a-b-c--e-h-a
3.	a-b-d-f-e-h-a
4.	a-b-d-e-h-a

Цикломатическое число зависит только от количества предикатов, сложность которых при этом не учитывается. Например, имеется два оператора условия:

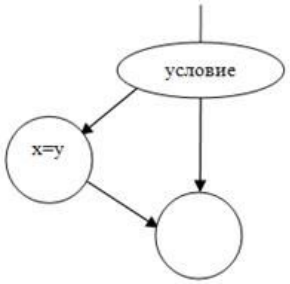
IF X>0 THEN X=A ELSE	IF (X>0 & FLAG = '1'B) (X=0 & FLAG = '0'B) THEN X=A ELSE
	

Оба оператора предполагают единственное ветвление и могут быть представлены одним и тем же графом. Очевидно, цикломатическое число будет для обоих операторов одинаковым, не отражающим сложности предикатов, что весьма существенно при оценке программ.

Метрика Майерса

Майерс предложил расширение метрики Мак-Кейба. Суть подхода Майерса состоит в представлении метрики сложности программ в виде интервала [Z(G), Z(G)+h]. Для простого предиката

h=0, а для n-местных предикатов h=n-1. Т. о., первому оператору соответствует интервал [2,2], а второму - [2,6]. По идее такая метрика позволяет различать программы, представленные одинаковыми графами. Информация о результатах использования этого метода отсутствует, поэтому ничего нельзя сказать о его применимости.



Метрика Джилба

Одной из наиболее простых, но, как показывает практика, достаточно эффективных оценок сложности программ является метрика Джилба, в которой логическая сложность программы определяется как насыщенность программы выражениями типа IF-THEN-ELSE.

Формула метрики Джилба:

$C_L = C_A / N$	N - количество операторов; C_A - число операторов вида IF - THEN - ELSE

МЕТРИКИ СЛОЖНОСТИ ПОТОКА ДАННЫХ

Другая группа метрик сложности программ - метрики сложности потока данных, т. е. использования, конфигурации и размещения данных в программах. Следует отметить, что метрики основаны на анализе исходных текстов программ и графов, что обеспечивает единый подход к автоматизации их расчета.

Метрика обращения к глобальным переменным

Пара "модуль-глобальная переменная" обозначается как (p, r), где p - модуль, имеющий доступ к глобальной переменной r. В зависимости от наличия в программе реального обращения к переменной r формируются два типа пар "модуль-глобальная переменная": фактические и возможные. Возможное обращение к r с помощью p показывает, что область существования r включает в себя p. Характеристика A_{up} говорит о том, сколько раз модули U_p действительно получали доступ к глобальным переменным, а число P_{up} - сколько раз они могли бы получить доступ. Метрика показывает приближенную вероятность ссылки произвольного модуля на произвольную глобальную переменную (отношение числа фактических обращений к возможным). Очевидно, чем выше эта вероятность, тем выше вероятность "несанкционированного" изменения какой-либо переменной, что может существенно осложнить работы, связанные с модификацией программы.

Формула метрики обращения к глобальным переменным:

$R = A_{up}/P_{up}$.

Пример. Программа с 3 глобальными переменными (x, y, z) и есть 3 подпрограммы. В первой подпрограмме используется x, во второй подпрограмме - y, z, а в третьей - ничего.
Решение: $P_{осн} = 3_{перем} * 3_{подпрог} = 9$; $A_{осн} = 1 + 2 + 0 = 3$; $R = 3/9=1/3$.

Спен-метрика спена

Определение спена основывается на локализации обращений к данным внутри каждой программной секции. Идентификатор, появившийся n раз, имеет спен = n - 1. При большом спене усложняется тестирование и отладка. Усреднённый по всем идентификаторам, спен даёт сложность программы.

Формула спен-метрики:

Спен $S_{ср}$ = числу утверждений, содержащих данный идентификатор, между его первым и последним появлением в тексте программы

Пример: Пусть в программе имеется 7 идентификаторов: 2 (спен=12), 2 (спен=41), 3 (спен=80). Тогда спен-метрика $S_{ср} = (12*2 + 41 * 2 + 80 * 3)/7 \approx 50$.

Метрика Чепина

Существует несколько ее модификаций. Рассмотрим более простой, а с точки зрения практического использования – достаточно эффективный вариант этой метрики. Суть метода состоит в оценке информационной прочности отдельно взятого программного модуля с помощью анализа характера использования переменных из списка ввода-вывода. Все множество переменных, составляющих список ввода-вывода, разбивается на четыре функциональные группы.

Множество «P»	Вводимые переменные для расчетов и для обеспечения вывода. Примером может служить используемая в программах лексического анализатора переменная, содержащая строку исходного текста программы, то есть сама переменная не модифицируется, а только содержит исходную информацию
Множество «M»	Модифицируемые или создаваемые внутри программы переменные.
Множество «C»	переменные, участвующие в управлении работой программного модуля (управляющие переменные)
Множество «T»	Не используемые в программе (“паразитные”) переменные. Поскольку каждая переменная может выполнять одновременно несколько функций, необходимо учитывать ее в каждой соответствующей функциональной группе

Далее вводится значение метрики Чепина: $Q = a_1P + a_2M + a_3C + a_4T$, где a_1, a_2, a_3, a_4 – весовые коэффициенты. Весовые коэффициенты использованы для отражения различного влияния на сложность программы каждой функциональной группы. Наибольший вес, равный трем, имеет функциональная группа C, так как она влияет на поток управления программы. Весовые коэффициенты остальных групп распределяются следующим образом: $a_1=1$; $a_2=2$; $a_3=3$, $a_4=0.5$. Весовой коэффициент группы T не равен нулю, поскольку “паразитные” переменные не увеличивают сложности потока данных программы, но иногда затрудняют ее понимание.

Формула метрики Чепина:

Q = P + 2*M + 3*C + 0.5*T	P - вводимые переменные для расчетов и для обеспечения вывода; M - модифицируемые или создаваемые внутри программы переменные; C - управляющие переменные; T - не используемые в программе (“паразитные”) переменные

Метрика уровня комментированности программ

Наиболее простой метрикой стилистики и понятности программ является оценка уровня комментированности программы (насыщенность программы комментариями).

Формула метрики уровня комментированности программ:

F = Nком/Nстр,	$N_{ком}$ - число комментариев в программе; $N_{стр}$ - число строк/операторов исходного текста

Исходя из практического опыта принято считать, что $F \geq 0.1$, т. е. на каждые десять строк программы должен приходиться минимум один комментарий. Как показывают исследования, комментарии распределяются по тексту программы неравномерно: в начале программы их избыток, а в середине или в конце - недостаток. Это объясняется тем, что в начале программы, как правило, расположены операторы описания идентификаторов, требующие более "плотного" комментирования. Кроме того, в начале программы также расположены "шапки", содержащие общие сведения об исполнителе, характере, функциональном назначении программы и т. п. Уровень комментированности программы считается нормальным, если выполняется условие: $F \approx n$. В противном случае какой-либо фрагмент программы дополняется комментариями до номинального уровня. Такая насыщенность компенсирует недостаток комментариев в теле программы, и поэтому формула недостаточно точно отражает комментированность

функциональной части текста программы. Более удачен вариант, когда вся программа разбивается на N равных сегментов и для каждого из них определяется F_i = SIGN ($N_{комм.i}$ / N_i - 0,1).

Общая формула метрики уровня комментированности программ:

$F = \sum F_i$	F_i = SIGN ($N_{комм.i}$ / N_i - 0,1)

Метрика Чена

Топологическая мера Чена выражает сложность программы через число пересечений границ между областями, образуемыми графом программы. Этот подход применим только к структурированным программам, допускающим лишь последовательное соединение управляющих конструкций. Для неструктурированных программ мера Чена существенно зависит от условных и безусловных переходов, в этом случае можно указать верхнюю (= m + 1 где m - число логических операторов при их взаимной вложенности) и нижнюю (= 2) меры. Когда управляющий граф программы имеет только одну компоненту связности, мера Чена совпадает с цикломатической мерой Мак-Кейба.

Формула метрики Чена:

M(G) = (Z(G), C, Q)	Z(G) - цикломатическая сложность Маккейба$ C - количество условий, необходимых для покрытия управляющего графа минимальным числом маршрутов$ Q - степень связности структуры программы и ее протяженности

Метрики Харрисона-Мейджела

Метрики учитывают уровень вложенности и протяженность программы. Каждой вершине присваивается своя сложность в соответствии с оператором, который она изображает. Эта начальная сложность вершины может вычисляться любым способом, включая использование мер Холстеда. Выделим для каждой предикатной вершины подграф, порожденный вершинами, которые являются концами исходящих из нее дуг, а также вершинами, достижимыми из каждой такой вершины (нижняя граница подграфа), и вершинами, лежащими на путях из предикатной вершины в какую-нибудь нижнюю границу. Этот подграф называется сферой влияния предикатной вершины. Приведенной сложностью предикатной вершины называется сумма начальных или приведенных сложностей вершин, входящих в ее сферу влияния, плюс первичная сложность самой предикатной вершины. Функциональная мера (SCOPE) программы - это сумма приведенных сложностей всех вершин управляющего графа. Функциональным отношением (SCORT) называется отношение числа вершин в управляющем графе к его функциональной сложности, причем из числа вершин исключаются терминальные. SCORT может принимать разные значения для графов с одинаковым цикломатическим числом.

Метрика Пивоварского:

В метрике Пивоварского вычисляется модифицированная цикломатическая сложность C^*. Отличие модифицированной цикломатической сложности C^* от цикломатической сложности C: оператор Case с n-выходами рассматривается как один логический оператор, а не как n - 1 операторов.

Формула метрики Пивоварского:

$C_{piv} = C^* + \sum P_i$	C^* - модифицированная цикломатическая сложность; P_i - глубина вложенности i - ой предикатной вершины

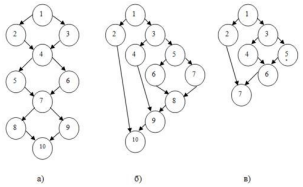
Метрика Вудворда

Метрика Вудворда использует меру Вудфорда - количество пересечений дуг управляющего графа. Так как в хорошо структурированной программе таких ситуаций возникать не должно, то данная метрика применяется в основном в слабо структурированных языках (Ассемблер, Фортран). Точка пересечения возникает при выходе управления за пределы двух вершин, являющихся последовательными операторами.

Метод граничных значений

Введем несколько дополнительных понятий, связанных с графом программы. Пусть G = (V, E) - ориентированный граф программы с единственной начальной и единственной конечной вершинами. В этом графе число входящих в вершину дуг называется отрицательной степенью вершины, а число исходящих из вершины дуг - положительной степенью вершины. Тогда набор вершин графа можно

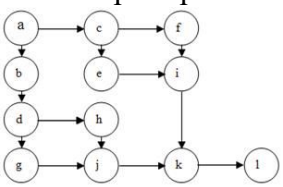
разбить на две группы: вершины, у которых положительная степень ≤ 1; вершины, у которых положительная степень ≥ 2. Вершины первой группы назовем принимающими вершинами, а вершины второй - вершинами отбора. Для получения оценки по методу граничных значений необходимо разбить граф G на максимальное число подграфов G', удовлетворяющих следующим условиям: вход в подграф осуществляется только через вершину отбора; каждый подграф включает вершину (называемую в дальнейшем нижней границей подграфа), в которую можно попасть из любой другой вершины подграфа. Например, вершина отбора, соединенная сама с собой дугой-петлей, образует подграф. Преимущество метода граничных значений состоит в его чувствительности к конструктивному контексту программ (графам программ с одинаковым цикломатическим числом):



Формула относительной граничной сложности программы:

$S_0 = 1 - (\upsilon - 1)/S_a$	υ – общее число вершин графа программы; S_a - абсолютная граничная сложность программы

Пример. Анализ сложности программы методом граничных значений



Число вершин, образующих подграфы программы равно скорректированной сложности вершины отбора:

Подграфы программы:

Характеристики подграфов программы	Вершина отбора			
	a	b	c	d
Вершины перехода	b, c	b, d	e, f	g, h
Скорректированная сложность вершины графа	10	2	3	3
Вершины подграфа	b, c, d, e, f, g, R, i, j	b	e, f	g, h
Нижняя граница подграфа	k	d	i	j

Скорректированная сложность вершин графа программы:

Вершина графа программы	Скорректированная сложность вершины графа	Вершина графа программы	Скорректированная сложность вершины графа
a	10	g	1
b	2	h	1
c	3	i	1
d	3	j	1
e	1	k	1
f	1	l	0

Каждая принимающая вершина имеют скорректированную сложность, равно 1, кроме конечной вершины, скорректированная сложность которой равна 0. Скорректированные сложности всех вершин графа G суммируются, образуя абсолютную граничную сложность программы. После этого определяется относительная граничная сложность программы: $S_0 = 1 – (11/25) = 0,56$.

Метрика Шнейдевинда

Метрика Шнейдевинда выражается через число возможных путей в управляющем графе.

МЕТРИКИ СЛОЖНОСТИ ПОТОКА УПРАВЛЕНИЯ И ДАННЫХ ПРОГРАММЫ

Метрики сложности потока управления и данных программы близки как к классу количественных метрик, классу метрик сложности потока управления программы, так и к классу метрик сложности потока управления данными (строго говоря, данный класс метрик и класс метрик сложности потока управления программы являются одним и тем же классом - топологическими метриками, но имеет смысл разделить их в данном контексте для большей ясности). Данный класс метрик устанавливает сложность структуры программы как на основе количественных подсчетов, так и на основе анализа управляющих структур.

Метрика тестирующей М-Меры

Тестирующей мерой М называется мера сложности, удовлетворяющая следующим условиям:

- Мера возрастает с глубиной вложенности и учитывает протяженность программы. К тестирующей мере близко примыкает мера на основе регулярных вложений. Идея этой меры сложности программ состоит в подсчете суммарного числа символов (операндов, операторов, скобок) в регулярном выражении с минимально необходимым числом скобок, описывающим управляющий граф программы. Все меры этой

группы чувствительны к вложенности управляющих конструкций и к протяженности программы. Однако возрастает уровень трудоемкости вычислений.

- Связанность модулей программы. Если модули сильно связанны, то программа становится трудномодифицируемой и тяжелой в понимании. Данная мера не выражается численно. Виды связанности модулей:
- Связанность по данным - если модули взаимодействуют через передачу параметров и при этом каждый параметр является элементарным информационным объектом. Это наиболее предпочтительный тип связанности (сцепления).
- Связанность по структуре данных - если один модуль посылает другому составной информационный объект (структуру) для обмена данными.
- Связанность по управлению - если один посылает другому информационный объект - флаг, предназначенный для управления его внутренней логикой. Модули связаны по общей области в том случае, если они ссылаются на одну и туже область глобальных данных. Связанность (сцепление) по общей области является нежелательным, так как, во-первых, ошибка в модуле, использующем глобальную область, может неожиданно проявиться в любом другом модуле; во-вторых, такие программы трудны для понимания, так как программисту трудно определить какие именно данные используются конкретным модулем.
- Связанность по содержимому - если один из модулей ссылается внутрь другого. Это недопустимый тип сцепления, так как полностью противоречит принципу модульности, т.е. представления модуля в виде черного ящика.
- Внешняя связанность - два модуля используют внешние данные, например, коммуникационный протокол.
- Связанность при помощи сообщений - наиболее свободный вид связанности, модули напрямую не связаны друг с другом, о сообщаются через сообщения, не имеющие параметров.
- Отсутствие связанности - модули не взаимодействуют между собой.
- Подклассовая связанность - отношение между классом-родителем и классом-потомком, причем потомок связан с родителем, а родитель с потомком - нет.
- Связанность по времени - два действия сгруппированы в одном модуле лишь потому, что ввиду обстоятельств они происходят в одно время.
- Мера Колофелло может быть определена как количество изменений, которые требуется произвести в модулях, отличных от модуля, стабильность которого проверяется, при этом эти изменения должны касаться проверяемого модуля.

Метрика Мак-Клура

Метрика Мак-Клура ориентирована на программы, хорошо структурированные, составленные из иерархических модулей, задающих функциональную спецификацию и структуру управления. Также подразумевается, что в каждом модуле одна точка входа и одна точка выхода, модуль выполняет ровно одну функцию, а вызов модулей осуществляется в соответствии с иерархической системой управления, которая задаёт отношение вызова на множестве модулей программы. Выделяются три этапа вычисления данной метрики:

1. Для каждой управляющей переменной i вычисляется значенит её сложностной функции C(i) по формуле: C(i) = (D(i) * J(i))/n, где D(i) - величина, измеряющая сферу действия переменной i. J(i) - мера сложности взаимодействия модулей через переменную i, n - число отдельных модулей в схеме разбиения.
2. Для всех модулей, входящих в сферу разбиения, определяется значение их сложностных функций M(P) по формуле $M(P) = f_p * X(P) + g_p * Y(P)$, где f_p и g_p - соответственно, число модулей, непосредственно предшествующих и непосредственно следующих за модулем P, X(P) - сложность обращения к модулю P, Y(P) - сложность управления вызовом из модуля P других модулей.
3. Общая сложность MP иерархической схемы разбиения программы на модули задаётся формулой: MP = СУММА(M(P)) по всем возможным значениям P - модулям программы.

Метрика Берлингера

Метрика Берлингера основана на информационной концепции - мере Берлингера. Недостатком данной метрики является то, что программа, содержащая много уникальных символов, но в малом количестве, будет иметь такую же сложность как программа, содержащая малое количество уникальных символов, но в большом количестве.

Формула метрики Берлингера:

$M= \sum f_i * \log_2 p_i$	f_i - частота появления i-го символа, pi - вероятность его появления

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ МЕТРИКИ

В связи с развитием объектно-ориентированных языков программирования появился новый класс метрик, также называемый объектно-ориентированными метриками.

Метрики Мартина

В 1995 г. Мартин предложил пять основных принципов дизайна классов в объектно-ориентированном проектировании (так называемая методология SOLID). Прежде чем начать рассмотрение метрик Мартина необходимо ввести понятие категории классов. В реальности класс может достаточно редко быть повторно использован изолированно от других классов. Практически каждый класс имеет группу классов, с которыми он работает в кооперации, и от которых он не может быть легко отделен. Для повторного использования таких классов необходимо повторно использовать всю группу классов. Такая группа классов сильно связна и называется категорией классов. В программах класс редко может быть повторно использован изолированно от других классов. Почти всегда класс имеет группу классов, с которыми он работает во взаимодействии и от которых его достаточно трудно отделить. Для повторного использования таких классов необходимо заново использовать всю группу классов. Связность такой группы классов (она называется категорией классов) достаточно высока, и для ее существования должны быть соблюдены следующие условия:

1.	Классы в пределах категории закрыты от любых попыток изменения отдельных экземпляров. Это означает, что если одному классу необходимо измениться, то весьма вероятно изменение всех классов в этой категории. Если любой из классов открыт для некоторого вида изменений, они все открыты для этого вида изменений
2.	Классы в категории повторно используются только вместе. Они настолько взаимозависимы, что не могут быть отделены друг от друга. Поэтому если предпринимается попытка повторного использования одного класса в категории, то все другие классы этой категории также повторно используются вместе с таким классом
3.	Классы в категории обеспечивают некоторую общую функцию или достигают некоторую общую цель

Ответственность, независимость и стабильность категории, по мнению Мартина, могут быть измерены путем подсчета некоторых зависимостей, взаимодействующих с этой категорией. Поскольку данные зависимости не являются основными, опустим их рассмотрение. Мартин предложил следующие метрики для оценки характеристик программы при объектно-ориентированном программировании.

Формулы базовых метрик Мартина:

Название	Формула	Описание	Значения
Центростремительное сцепление)	C_a	Количество классов вне этой категории, которые зависят от классов внутри этой категории	
Центробежное сцепление	C_e	Количество классов внутри этой категории, которые зависят от классов вне этой категории	
Нестабильность	I:: $I = C_e / (C_a+C_e)$.		[0,1]; I = 0 - max максимально стабильная категория; I = 1 - maxo не стабильную категория
Абстрактность	$A = n_A / n_{All}$	n_A – количество абстрактных классов в категории; n_{All} –	[0, 1];

Название	Формула	Описание	Значения
		общее количество классов в категории	0 - категория полностью конкретна, 1 - категория полностью абстрактна

Теперь на основе приведенных метрик Мартина можно построить график, на котором отражена зависимость между абстрактностью и нестабильностью. Если на нем построить прямую, задаваемую формулой I+A=1, то на этой прямой будут лежать категории, имеющие наилучшую сбалансированность между абстрактностью и нестабильностью. Эта прямая называется главной последовательностью. Практически для любых категорий верно то, что чем ближе они находятся к главной последовательности, тем лучше. Далее можно ввести еще две метрики.

Формулы производных метрик Мартина:

Расстояние до главной последовательности $D=\|(A+I-1) / (\sqrt{2})\|$
Нормализированной расстояние до главной последовательности $D_n=\|A+I-2\|$

Метрики Лоренца и Кидда

Комплексный набор метрик, предложенный в 1994 г. Лоренцем и Киддом, имеет практическую направленность на использование в промышленной разработке ПО. Набор включает 10 метрик, которые, в свою очередь, классифицируют в следующие группы:

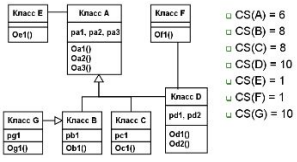
Метрики размера	Подсчёт свойств и операций для отдельных классов, а также их средних значений для всей объектно-ориентированной системы
Метрики наследования	Способы повторного использования операций в иерархии классов
Внутренние метрики	Связность и кодирование
Внешние метрики	Сцепление и повторное использование

- Метрики размера

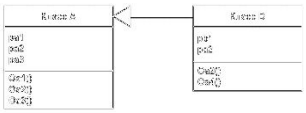
Размер класса CS (Class Size) — общий размер класса определяется на основании определения следующих показателей: общее количество операций; количество свойств. Метрика Чидамбера и Кемерера является взвешенной метрикой размера класса Указанные измерения в обоих случаях следует проводить с учетом частных (приватных) и наследуемых экземплярных операций, которые инкапсулируются внутри класса:

$CS_{=}=C_{\Sigma}+S_{\Sigma}$	C_{Σ} - количество инкапсулированных классом методов (операций); S_{Σ} - количество инкапсулированных классом свойств.

Большие значения метрики CS указывают на то, что класс имеет слишком много обязанностей, что уменьшает возможность повторного использования класса, усложняет его реализацию и тестирование. При определении размера класса больший удельный вес придают унаследованным (публичным) операциям и свойствам, потому что приватные операции и свойства обеспечивают специализацию и являются более локализованными в проекте. Могут вычисляться средние количества свойств и операций класса. Чем меньше среднее значение размера CS, тем больше вероятность повторного использования класса. Рекомендуемое значение метрики S_S ограничено сверху 20 инкапсулированными методами и свойствами ($C_S < 20$ методов).



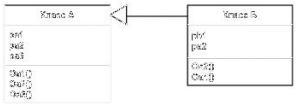
Количество операций, переопределяемых подклассом NOO (Number of Operations Overridden by a Subclass). Переопределением называют случай, когда подкласс замещает операцию, унаследованную от суперкласса, своей собственной версией. Большие значения NOO указывают на возникшие проблемы проектирования. Понятно, что подкласс должен расширять операции суперкласса, что проявляется в виде новых имен операций. Если же значение метрики NOO достаточно велико, то это означает нарушение разработчиком абстракции суперкласса. Это явление ослабляет иерархию классов, усложняет тестирование и модификацию программного обеспечения. Рекомендуемое значение метрики: $NOO \leq 3$ метода.



Количество операций, добавленных подклассом NOA (Number of Operations Added by a Subclass), определяется количеством добавленных относительно родительского класса собственных методов(операций):

$NOA = N_\Sigma$	N_Σ — количество новых методов класса, добавленных относительно суперкласса

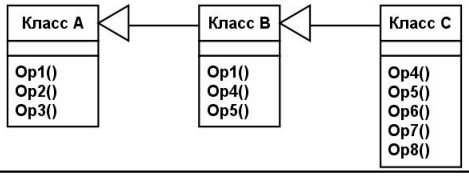
С увеличением NOA подкласс приобретает меньшую общность со своим суперклассом, что требует больших трудозатрат по тестированию и внесению изменений. При увеличении высоты дерева иерархии классов (с ростом значения DIT) должно уменьшаться количество новых методов классов нижних уровней. Для рекомендуемых граничных значений размера класса (CS= 20) и высоты дерева иерархии классов (= 6) значение NOA ограничено значением 4 ($NOA \leq 4$).



Индекс специализации SI (Specialization Index) характеризует грубую оценку степени специализации каждого подкласса при добавлении, удалении или переопределении операций:

$SL = NOO * u / M_{общ}$	u— номер уровня в иерархии, на котором находится подкласс; $M_{общ}$— общее количество методов класса

Чем выше значение метрики SI, тем выше вероятность того, что в иерархии классов есть отдельные экземпляры, нарушающие абстракцию суперкласса. Рекомендуемое значение показателя SI ограничено сверху величиной 0,15 ($SI \leq 0,15$).



$$SI(A) = \frac{0 \cdot 1}{3} = 0 \quad SI(B) = \frac{1 \cdot 2}{5} = 0,4 \quad SI(C) = \frac{2 \cdot 3}{8} = 0,75$$

- Группа метрик «Метрики наследования метрики» предназначена для оценки операций в классах. Как правило, методы бывают небольшими как по размеру, так и по логической сложности. В то же время истинные характеристики операций помогают более глубоко понять особенности создаваемой системы.

Средний размер операции AOS (Average Operation Size) определяется количеством сообщений, порождаемых операцией. В качестве оценки размера может использоваться количество строк программы, однакоLOC-оценки приводят к известным проблемам. Иным вариантом может являться количество сообщений, посланных операцией. Рост значения данного показателя означает, что обязанности размещены в классе не очень удачно. Рекомендуемое значение метрикиAOSне должно превышать 9. Увеличение среднего размера относительно этой границы рассматривают как показатель неудачного проектирования обязанностей класса

Сложность операции OC (Operation Complexity) может быть вычислена на основе стандартных метрик сложности (например, с помощью LOC- или FP-оценок, метрики дипломатической сложности, метрики Холстеда). Лоренц и Кидд предложили вычислять значение OC суммированием оценок с весовыми коэффициентами вариантов действий:

Действие	Вес
Определение (описание) переменной-параметра	0,3
Определение (описание) временной переменной	0,5
Присваивание значения	0,5
Вложенное выражение	0,5
Сообщение без параметров	1
Арифметическая операция	2
Сообщение с параметрами	3
Вызов стандартной функции интерфейса (API)	5

Действие	Вес
Вызов пользовательской функции (простой вызов)	7

Рекомендуемое значение метрики ограничено числом 65 (ОС < 65).
Пример.

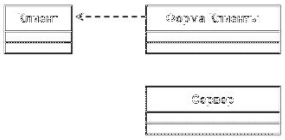
void Push(int item) {
if (top==size)
printf(“Empty stack.\n”);
else
array[top++]=item;
}
	ОС = 0,3 + 2 * 0,5 + 2 = 3,3

Среднее количество параметров на операцию ANP (Average Number of Parameters per operation) определяется отношением числа параметров к количеству операций (методов) класса. Чем больше параметров у операции, тем сложнее взаимодействие между объектами. Поэтому значение метрики ANP должно быть как можно меньшим. Рекомендуемое значение AMP = 0,7.

- Группа метрик «Внутренние метрики» предназначена для оценки показателей процесса разработки ПС. Центральным вопросом в процессе разработки является прогноз размера создаваемого продукта.

Количество описаний сценариев NSS (Number of Scenario Scripts) измеряется или количеством классов, реализующих требования к ПО, или количеством состояний для каждого класса, или количеством методов класса. При своем не совсем обычном способе измерения метрикаNSSявляется достаточно эффективным индикатором размера создаваемой программы. Рекомендуется не менее одного сценария. Рост количества сценариев неминуемо ведет к увеличению размера программы.

Количество ключевых классов NKC (Number of Key Classcs) адекватно характеризует предстоящий объем работы по программированию. Ключевой класс прямо связан с проблемной областью, для которой предназначена система, поэтому, если предметная область специфична, то возможность повторного использования существующего ключевого класса маловероятна и требуется самостоятельная разработка «с нуля». Авторы этого набора метрик полагают, что в типовой объектно-ориентированной системе на долю ключевых классов приходится от 20 до 40% общего количества классов. Остальные классы реализуют общую инфраструктуру (интерфейсы, коммуникации, базы данных). Рекомендуется ограничивать значение метрики снизу значением 0,2. Если значение метрики NKC < 0,2 от общего количества классов системы, следует пересмотреть выделение классов.



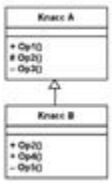
Количество подсистем NSUB (Number of subsystem) определяется непосредственным подсчетом. Количество подсистем обеспечивает понимание таких вопросов, как размещение ресурсов, планирование (с акцентом на параллельную разработку), общие затраты на интеграцию. Рекомендуется выделять в программном комплексе (системе) не менее трех подсистем (NSUB > 3). Количество подсистем характеризует трудоемкость и управляемость проекта.

- Группа метрик «Внешние метрики» целесообразно накапливать по мере выполнения проектов по разработке ПС. Эти данные могут применяться для вычисления показателей производительности. Совместное применение метрик позволяет оценивать необходимые ресурсы на выполнение проекта: затраты, продолжительность разработки, численность привлекаемого персонала и другие характеристики.

Метрики Абреу
Фактор закрытости метода (MHF):

$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$	Mv(Ci)–количество видимых методов в классе Ci (интерфейс класса), Mh(Ci)–количество скрытых методов в классе Ci (реализация класса), Md(Ci)–общее количество методов в классе Ci (унаследованные методы не учитываются), Md(Ci) = Mv(Ci) + Mh(Ci), TC–количество классов в системе.

С увеличением значения метрики уменьшается плотность дефектов в системе и затраты на их устранение. Пример:



$M_h(A) = 2$
$M_a(A) = 3$
$M_h(B) = 1$
$M_a(B) = 3$

$$MHF = \frac{1+2}{3+3} = \frac{3}{6} = 0{,}5$$

Фактор закрытости свойства (AHF):

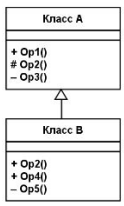
$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$	Av(Ci)–количество видимых свойств в классе Ci (интерфейс класса), Ah(Ci)–количество скрытых свойств в классе Ci (реализация класса), Ad(Ci)–общее количество свойств, определенных в классе Ci (унаследованные свойства не учитываются), Ad(Ci) = Av(Ci) + Ah(Ci), TC–количество классов в системе.

В идеальном случае все свойства должны быть скрыты и доступны только для методов соответствующего класса (AHF = 100 %).

Фактор наследования метода (MIF):

$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$	Mi(Ci)–количество унаследованных и не переопределенных методов в классе Ci, Mo(Ci)–количество унаследованных и переопределенных методов в классе Ci, Mn(Ci)–количество новых (не унаследованных и переопределенных) методов в классе Ci, Md(Ci)–количество методов, определенных в классе Ci, Md(Ci) = Mn(Ci) + Mo(Ci), Ma(Ci)–общее количество методов, доступных в классе Ci, Ma(Ci) = Md(Ci) + Mi(Ci), TC–количество классов в системе.

Значение метрики MIF = 0 указывает, что в системе отсутствует эффективное наследование, например, все унаследованные методы переопределены. С увеличением значения метрики уменьшается плотность дефектов в системе и затраты на их устранение. Очень большие значения MIF (70 – 80 %) приводят к обратному эффекту. Рекомендация – умеренное использование наследования. Пример:



$M_i(A) = 0$
$M_a(A) = 3$
$M_i(B) = 2$
$M_a(B) = 5$

$$MIF = \frac{0+2}{3+5} = \frac{2}{8} = 0{,}25$$

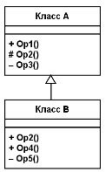
Фактор наследования свойства (AIF):

$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$	Ai(Ci)–количество унаследованных и не переопределенных методов в классе Ci, Ao(Ci)–количество унаследованных и переопределенных методов в классе Ci, An(Ci)–количество новых (не унаследованных и переопределенных) методов в классе Ci, Ad(Ci)–количество методов, определенных в классе Ci, Ad(Ci) = An(Ci) + Ao(Ci), Aa(Ci)–общее количество методов, доступных в классе Ci, Aa(Ci) = Ad(Ci) + Ai(Ci), TC–количество классов в системе.

Фактор полиморфизма (POF):

$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} \lvert M_n(C_i) \times DC(C_i) \rvert}$	Mo(Ci)–количество унаследованных и переопределенных методов в классе Ci, Mn(Ci)–количество новых (не унаследованных и переопределенных) методов в классе Ci, Md(Ci)–общее количество методов, определенных в классе Ci, Md(Ci) = Mn(Ci) + Mo(Ci), DC(Ci)–количество потомков класса Ci, TC–количество классов в системе.

Знаменатель представляет максимальное количество возможных полиморфных ситуаций для класса Ci. Имеется в виду случай, когда все новые методы, определенные в классе Ci, переопределяются во всех его потомках. Умеренное использование полиморфизма уменьшается плотность дефектов в системе и затраты на доработку. Однако при POF > 10 % возможен обратный эффект. Пример:



$M_o(A) = 0$
$M_n(A) = 2$
$DC(A) = 1$
$M_o(B) = 1$
$M_n(B) = 2$
$DC(B) = 0$

$$POF = \frac{0-1}{2 \cdot 1 + 2 \cdot 0} = \frac{1}{2} = 0{,}5$$

Фактор сцепления (COF):

$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i, C_j)\right]}{TC^2 - TC}$	is_client(Ci, Cj) = 1, если класс Ci содержит хотя бы одну не унаследованную ссылку на свойство или метод класса Cj, иначе – is_client(Ci, Cj) = 0. TC–количество классов в системе.

Знаменатель соответствует максимально возможному количеству сцеплений. Сцепления отрицательно влияют на качество ПО, их нужно сводить к минимуму.

МЕТРИКИ НАДЕЖНОСТИ

Метрики надёжности – метрики, близкие к количественным, но основанные на количестве ошибок и дефектов в программе. Нет смысла рассматривать особенности каждой из этих метрик, достаточно будет их просто перечислить: количество структурных изменений, произведенных с момента прошлой проверки, количество ошибок, выявленных в ходе просмотра кода, количество ошибок, выявленных при тестировании программы и количество необходимых структурных изменений, необходимых для корректной работы программы. Для больших проектов обычно рассматривают данные показатели в отношении тысячи строк кода, т. е. среднее количество дефектов на тысячу строк кода.

ГИБРИДНЫЕ МЕТРИКИ

Гибридные метрики основываются на более простых метриках и представляют собой их взвешенную сумму.

Метрика Кокола

Метрика Кокола использует базовую метрику M, другие интересные меры M_i, корректно подобранные коэффициенты R_i, функции $M(M_i)$. Общий вид метрики Кокола: $K = (M + R_1 * M(M_1) + \ldots + R_n * M(M_n))/(1 + R_1 + \ldots + R_n)$. Выделяют три модели для мер: Маккейба, Холстеда и SLOC, где в качестве базовой используется мера Холстеда. Эти модели получили название «наилучшая», «случайная» и «линейная».

Формула метрики Кокола:

$K = (M + R_1 * M(M_1) + M(M_2) + M(M_3))/(1 + R_1 + R_2 + R_3)$	M - метрика Холстеда, M_1 - метрика Маккейба; M_2 - метрика Холстеда; M_3 - метрика SLOC; R_1, R_2, R_3 - коэффициенты; $M(M_n)$, $M(M_n)$, $M(M_n)$ - функции

Метрика Зольновского-Симмонса-Тейера

Метрика Зольновского-Симмонса-Тейера представляет собой взвешенную сумму различных индикаторов. Существуют два варианта данной метрики. Используемые метрики в каждом варианте выбираются в зависимости от конкретной задачи, коэффициенты - в зависимости от значения метрики для принятия решения в данном случае.

Формулы метрики Зольновского-Симмонса-Тейера (структура, взаимодействие, объем, данные):

$ZST_1 = a + b + c + d$	a - структура; b - взаимодействие; c - объем; d - данные

$ZST_2 = x + y + z + p$	x - сложность интерфейса; y - вычислительная сложность; z - сложность ввода/вывода; p - сложность ввода/вывода

Примеры решения задач

Пример. Измерение первичных понятий метрики Холстеда для программы на языке Алгол, реализующей алгоритм Евклида для нахождения наибольшего общего делителя (НОД) двух чисел

```
        IF (A = 0)
LAST: BEGIN
        GCD :=B;
        RETURN
        END;
        IF (B = O)
        BEGIN
        GCD : = A;
        RETURN
        END;
HERE: G : = A/B;
        R := A-B*G;
        IF (R = 0) GO TO LAST;
        A: = B;
        B : = R;
        GO TO HERE;
```

В отношении классификации операторов интуитивно ясно, что символ присваивания «:=», знаки равенства =, вычитания -, деления / и умножения * соответствуют их обычному определению. Пара, состоящая из открывающей и закрывающей скобок, классифицируется как единый оператор группировки. Поскольку пара BEGIN ... END выполняет точно такую же группирующую функцию, она классифицируется как такой же оператор. Метки HERE: и LAST: - не переменные и не константы, поэтому они не являются операндами. Следовательно, эти метки должны быть операторами или их составными частями. Комбинация команды GO TO HERE и метки HERE: определяет ход выполнения программы путем задания для нее счетчика или указателя текста. Эта комбинация классифицируется как один оператор. Неиспользуемая метка, с другой стороны, трактуется всего лишь как комментарий, поэтому ее присутствие в программе (или ее части) необязательно. Ограничитель, или точка с запятой, также определяет ход выполнения программы (простым продвижением счетчика), что позволяет классифицировать точку с запятой как оператор. По той же причине все управляющие структуры, например, IF, IF ... THEN ... ... ELSE или DO ... WHILE, классифицируются как операторы. Параметры программы, полученные классификацией и подсчетом:

Характеристики операторов:

Оператор	i	$f_{1,i}$
;	1	9
:=	2	6
( ) или BEGIN... END	3	5
IF	4	3
=	5	3
/	6	1
-	7	1
*	8	1
GO TO HERE	9	1
GO TO LAST	10	1
	η_1= 10	N_1 = 31

Характеристики операндов:

Операнд	j	f_{2j}
B	1	6
A	2	5
O	3	3
R	4	3
G	5	2
GCD	6-	2
	η_1= 6	N_1 = 21

Метрика длины по Холстеду для алгоритма НОД: $\tilde{N} = 10 \log_2 10 + 6 \log_2 6 = 33+16 = 49.$ = 31+21 = 52 (N -реальное = экспериментальное).

Объем по Холстеду для алгоритма НОД: V = (31 + 21) log (10 + 6) = 208 бит.

Чтобы найти потенциальный объем, нам нужно только подсчитать число требуемых входных и выходных параметров. В данном случае это A, B и GCD, так что $\eta^*_2 = 3$. Следовательно, потенциальный объем по Холстеду программы для алгоритма НОД: $V^* = (2 + 3)\log_2(2 + 3) = 11{,}6$ бита.

Задания для самостоятельной работы

1. Написать консольную программу (с циклами и модулями).
2. Оценить программу по следующим метриками: количество строк исходного текста (под строкой понимается любой оператор программы); метрики Холстеда; цикломатическое число Мак-Кейба; метрика Джилба; метрика обращения к глобальным переменным; спен-метрика; метрика Чепина; метрика уровня комментированности программы, метрика Кафура, метрика корректности программы, метрика «Интеллектуальное содержание конкретного алгоритма, инвариантное по отношению к используемым языкам реализации», метрика необходимых интеллектуальных усилий по написанию программы, метрика флуктуации длин программной документации.
3. Измерить теоретическую длину программы, используя аппроксимирующую формулу.

Контрольные вопросы

1. Что такое управление качеством ПО?
2. Что мы понимаем под понятиями свойства и характеристика программы?
3. Перечислить известные характеристики измерительных шкал ПО.
4. Какие существуют группы методов оценки характеристик ПО?
5. Что образует иерархию характеристик качества ПО?
6. Что означает глубина дерева наследования в объектно-ориентированном программировании?
7. О чем говорит чрезмерная связность объектов?
8. Пояснить, что означает отклик на класс?
9. Что понимается под оптимизацией программы?
10. Что такое критерий качества? Перечислите его основные характеристики.
11. Что такое метрика качества программы?
12. Какие два основных направления исследовании метрик ПО существуют?
13. На какие три группы делятся метрики виду информации, получаемой при оценке качества ПО?
14. Какие метрические шкалы существуют?
15. Какие основные группы метрик выделяют при оценке сложности программ?
16. Перечислите метрические характеристики, используемые в метриках Холстеда?
17. Каким образом рассчитывается метрика длины программы?
18. Каким образом определяется метрика объема программы?
19. Чем метрика потенциального объема отличается от метрики объема?
20. Как установить уровень реализации программы?
21. Как определяется интеллектуальное содержание конкретного алгоритма?
22. Каким образом рассчитывается оценка необходимых интеллектуальных усилий по написанию программы?
23. Опишите методику нахождения цикломатического числа Мак-Кейба.
24. От чего зависит цикломатического числа Мак-Кейба и что оно не учитывает?
25. Опишите методику нахождения метрики Джилба.
26. Какие метрики относятся к метрикам сложности потока данных?
27. Что показывает метрика обращения к глобальным переменным?
28. Как определяется метрика Спена?
29. Для чего используется метрика Чепина и каким образом она рассчитывается?
30. Как найти уровень комментированности программы?
31. Каким образом распределяются комментарии в программе?
32. Что такое метрика программного обеспечения?

33. Дать определение понятия "качество ПО".
34. Дать определение характеристики качества программ.
35. Что такое критерий качества программы?
36. Перечислить характеристики критериев качества ПО?
37. Два основных направления в исследовании метрик ПО.
38. Перечислить три группы метрик, различающихся по виду информации, получаемой при оценке качества ПО.
39. Назвать существующие качественные оценки программ.
40. Три основные группы метрик сложности программ.
41. Размерно-ориентированные метрики.
42. Дать понятие метрики Холстеда.
43. Дать понятие метрики Джилба.
44. Определить метрику, получившую название "подсчет точек пересечения".
45. Дать определение качества ПО согласно международным стандартам.
46. Какие метрики можно отнести к метрикам стилистики и понятности программ?
47. Что такое уровень качества программирования?
48. В чем суть метрики корректности программ по Холстеду?
49. Привести аппроксимирующую формулу для определения уровня программы.
50. Определить характеристику интеллектуального содержания конкретного алгоритма.
51. Определить характеристику информативности программы.
52. Как можно измерить оценку необходимых интеллектуальных усилий по написанию программы?
53. Назвать метрики сложности программ, которые базируются на оценке использования конфигурации размещения данных в программе.
54. Дать определение метрики, связывающей сложность программы с обращениями к глобальным переменным.
55. Дать определение спена программы.
56. В чем состоит суть метрики Чепина?
57. Определить метрику Кафура, основанную на учете потока данных программы.
58. Какие программы являются потенциально корректными?
59. Какие ситуации в программе можно считать несовершенствами (стилистическими ошибками)?
60. Какой уровень комментированности программы принято считать достаточным?
61. Что такое теоретическая длина программы?
62. Назвать метрики, которые являются представителями группы метрик сложности потока управления программ.
63. Что такое управляющий граф программы?
64. Что такое цикломатическая сложность графа?
65. Что такое сильносвязный граф?
66. Что такое "модифицированная" цикломатическая сложность графа?
67. Суть подхода Майерса к расширению метрики Маккейба.
68. Оценка сложности программ по методу граничных значений.
69. Что такое отрицательная и положительная степени вершины графа.
70. Какие метрики можно отнести к вариантам цикломатической меры сложности программ?
71. С помощью какой метрики можно посчитать число путей в управляющем графе?