Вісник Харківського національного університету
Серія «Математичне моделювання. Інформаційні технології. Автоматизовані системи управління»
№ 629, 2004, с. 70-85

УДК 519.6+681.3.06

Математическая модель стиля Software Science для метрического анализа сложных наукоёмких программ

В. О. Мищенко
Харьковский национальный университет им. В.Н. Каразина, Украина

The new model of software energy is developed in Software Science style. It is free from roles of any program languages. The model shall be applied to the software metric analysis of the software developed for numeric experiments on the base of the Discreet Singularity Method. It is possible to utilize this model to analyze software energy in any complicated case.

1. Качество программных систем и задачи развития научных метрик

Рассматриваются вопросы прогноза и управления качеством таких наукоёмких программных систем, как средства проведения и анализа результатов численных экспериментов современной электродинамики. Наличие серьёзных проблем в данной области, отмеченное в [1], связано с бесперспективностью для программной индустрии разработки сложных экспериментальных систем, спрос на которые со стороны исследователей может быть острым, но редко – платежеспособным. Поэтому разработчиками программного обеспечения (ПО) выступают сами будущие потребители, которые весьма стеснены в ресурсах, и, подчас, имеют недостаточные навыки в области разработки сложных программных систем. В последнее десятилетие, казалось бы, появился выход из положения за счёт доступности высокоэффективных программных библиотек, организованных на основе принципов объектно-ориентированного программирования. За счёт этого реже стали срываться сроки разработки, почти не встречаются невразумительные интерфейсы пользователя. Благодаря росту производительности доступных компьютеров не так часто возникает необходимость в специальных программных решениях. Но сопровождаемая этим нетривиальная структуризация программ, рост их размеров усугубляют проблему адекватности сложных теоретически надёжных алгоритмов их фактической реализации. При наличии дефектов реализации шансы на их обнаружение в современных программах вычислительного назначения можно связывать только с систематическим оцениванием состояния разработки программной системы для своевременных решений о дополнительном контроле.

Ранее для прогноза трудоёмкости разработки ПО численных экспериментов методами дискретных особенностей удалось использовать математическую модель, отправляясь от понятия работы в программировании [2] из науки о программах М.Холстеда [3]. Этой наукой предлагается также прогностическая модель появления ошибок программирования.

В числе средств индикации качества ПО в отношении его надёжности важную роль играют качественные и количественные характеристики надёжности готового продукта или процесса его разработки [4]. Среди таких характеристик стандарт ISO982 называет и научные метрики Холстеда.

Отметим, что эта группа характеристик оказывается одной из наиболее разносторонних. Согласно [4], среди указанных в данном стандарте 39 метрик только две группы – научные метрики и метрики точности тестов - могут использоваться для трёх разных целей. Холстедовские метрики применимы для оценки «оставшегося количества дефектов» и «сложности» продукта, а также «рисков, преимуществ, стоимости» процесса разработки.

К главным преимуществам подхода Software Science – науки о программах – мы относим простоту практического применения, гибкость и потенциальную развиваемость. Надежда на развиваемость связана с подходом к поиску характеристик ПО и процесса его разработки в форме аналогов систем физических величин и моделей естественных процессов.

Однако подход науки о программах, быстро развивавшейся в 70-е годы прошлого века, сопряжен также с определёнными проблемами и в 80-е годы подвергался критике. Информация по этому поводу и полная библиография содержатся в монографии [5]. Замечания с позиций 90-х годов имеются в [6].

В [2] развиваемость науки о программах подтвердилась уточнением и практическим использованием аналогов внутренней энергии и работы в термодинамике на основе холстедовского понятия «работы в программировании». Но подсчёт необходимых при этом параметров программ столкнулся с недостаточностью определения этих величин в [3] при работе с программами сложной структуры. Приходится «по ходу» обобщить правила подсчёта параметров. Хорошо, что в программах, которая рассматривались в [2], число модулей в программе в среднем около 5, а содержательная инкапсуляция данных и методов обработки не применялись. Планируя производить подобные подсчёты для современных программ того же назначения, полагаться на упомянутые частные обобщения нельзя, поскольку показатель модульности увеличился на порядок, активно используются шаблоны, классы, конструкции, обслуживающие распараллеливание программ. То, что указанные трудности носят принципиальный, а не технический характер, мы обнаружили, в частности, при вычислении параметров программы, описанной в работе [7].

Таким образом, дальнейшее использование научных метрик в целях контроля за надёжностью разработки наукоёмких программных средств сложной структуры нуждается в новой математической модели энергетики современного ПО.

2. Цель настоящей работы

Для многомодульных программных систем, активно использующих объектно-ориентированное и параллельное программирование, и имеющих модули с управляемой внутри них видимостью других модулей, решаем следующую задачу.

Построить в духе Software Science [3] математическую модель, свободную от правил конкретных языков программирования, предназначенную для определения в процессе разработки сложной программной системы её энергетических характеристик в смысле [2].

3. Моделирование внутрипрограммных связей и энергетики состояния

Определение 1. Следующие условия 1^0-7^0 описывают *схемы программных систем* (сокр. *СПС*).

1^0. Схема программной системы P представляется множеством *модулей*, которое распадается на непересекающиеся подмножества: I - *интерфейсных* и R - *реализующих* модулей. На модулях системы P определены четыре бинарные отношения, описанные в пунктах 2^0-5^0.

2^0. Отношение «*имеет тело»* является частично определённой функцией на I со значениями в R. Эта функция разным интерфейсным модулям ставит в соответствие разные реализующие модули («тела»). Она обозначается префиксным символом b («тело»). В наших обозначениях интерфейсный модуль M имеет тело bM, если он его вообще имеет.

3^0. Отношение *«имеет родителя»* является частично определённой функцией p на I со значениями в I, причём соответствующее отображение не порождает зацикленных траекторий:

$$p^n M = M \Rightarrow n = 0 \qquad (1)$$

Функциональный префикс p читается как «родитель». Любой элемент M из прообраза $p^{-1}M'$ будет называться наследником модуля M'

4^0. Отношение *«имеет надмодуль»* является частично определённой функцией s на R со значениями в R, причём соответствующее отображение (модуль $\mapsto$ надмодуль) не порождает зацикленных траекторий:

$$s^n B = B \Rightarrow n = 0\,. \qquad (2)$$

Тело не должно имееть надмодулей:

$$\neg\exists B, M : sbM = B\,. \qquad (3)$$

Далее любой элемент B прообраза $p^{-1}B'$ будет называться субмодулем модуля B'.

5^0. Отношение *«зависит от»* может рассматриваться как частично определённая многозначная функция m на P со значениями в I. Обычно удобнее смотреть на неё как на всюду определённую функцию со значениями в множестве подмножеств I. Три отношения: транзитивное замыкание отношения «имеет родителя», произведение транзитивного замыкания отношения «*имеет надмодуль*» на обратное к отношению «имеет тело» и отношение *«зависит от»*, - не должны пересекаться.

(Примечание. Смысл последнего требования связан с интерпретацией. А именно, первые два из указанных отношений ($\overline{p}$ и $b^{-1}\overline{s}$) подразумеваются, как более сильные в сравнении с отношением «зависит от» (m).)

6^0. С каждым модулем связывается непустое множество *блоков* (называемых блоками данного модуля). Один из них, называемый *собственным* блоком модуля, является списком *программных символов* (сокр. ПС), а остальные – непересекающимися подсписками этого списка. Дополнение объединения всех внутренних блоков, если оно не тривиально, иногда удобно рассматривать как *остаточный* блок данного модуля. Блоки, подобно модулям, делятся на интерфейсные и реализующие, а в числе последних могут быть тела интерфейсных блоков. Указанное подразделение блоков должно быть согласовано с соответствующим подразделением модулей и описанными выше

отношениями на модулях, как указывается в следующем пункте. Собственный блок модуля X обозначается $< X >$.

7^0. На множестве внутренних интерфейсных блоков каждого модуля X задана частично определённая функция b_X со значениями в множестве внутренних блоков всех модулей схемы P. Эта функция ставит в соответствие блоку X_i его *тело* – некоторый реализующий блок $b(X_i) = b_X(X_i)$. Разным блокам в качестве тел могут соответствовать только разные блоки. Блоки, связанные с интерфейсным модулем, могут быть только интерфейсными. Телом интерфейсного блока M_i модуля $M \in I$ может служить только блок, соответствующий телу этого модуля ($b(M_i) \subseteq < bM >$) или собственный блок субмодуля тела. Если внутренний блок связан с $B \in R$, то тело этого блока может быть связано либо с B, либо с его субмодулем $B' \in R$ ($B = sB'$), и в последнем случае $b(B_i) =< B' >$.

8^0. Собственные блоки разных модулей не пересекаются.

Содержательно программные символы - это элементы исходных текстов программ, традиционно записанных на языках программирования, или представленных в графической или смешанной форме. Поэтому блок - направленный нелинейный список. После определения необходимых атрибутов СПС его можно заменить множеством ПС с указанием частот их вхождения. В [3] правила выделения в текстах программ программных символов, непосредственно применимые к простым процедурным языкам, имеются на уровне разбора примеров. На практике нужные правила несложно формализовать. Неизбежный при этом у разных исследователей разнобой, по нашему мнению и опыту, не существенен для выводов по результатам программометрических расчётов.

Следующие примеры подтверждают рациональность данного определения. Они также подготавливают формулировку последующих определений и правил подсчёта метрических характеристик, затрагивая тему видимости описаний.

Пример 1. Представим себе простую программу на языке Паскаль такой структуры:

```
program SP (Input, Output);
const  N=20;
var A: array [1..N] of Real;
 X: Real;
 I: Integer;
begin
 for I:= 1 to N do
    Read  A[I];
 (* other statements *)
end.
```

Её можно представить СПС, состоящей из одного реализующего модуля SP, с которым связан только его собственный блок:

$< SP >=$ (program • (•) SP Input , *и так далее* end .)

где ПС отделяются один от другого в списке серией пробелов.

Если в программе SP описать процедуру PRNT, то множество связанных с нею блоков можно пополнить ещё одним реализующим блоком SP_{PRNT}. Вместо этого модифицируем описания нашей программы далее таким образом:

```
program SP (Input, Output);
const  N=20;
type   T= array [1..N] of Real;
var A: T;
(* other variables *)
procedure PRNT (K: Integer; X: Real);FORWARD;
procedure MIN (X,Y,Z: Real: var M: Real);EXTERNAL;
procedure PRNT;(*block*)
 begin (* statements *) end;
begin  (* statements *)
end.
```

Тогда необходима отдельно компилируемая часть:

```
procedure (*body*) MIN (X,Y,Z: Real: var M: Real);
 (* declarations *)
begin  (* statements *)
end;
```

Теперь в числе внутренних блоков модуля SP будут: SP_{PRNT} - интерфейсный и $SP_{PRNT\text{-}block} = b(SP_{PRNT})$ - реализующий (тело). Можно было бы предположить наличие ещё одного внутреннего интерфейсного блока SP_{MIN}, но тогда пришлось бы считать, что его телом является собственный блок модуля $body\text{-}MIN$. А это некорректно, поскольку по правилам виртовского Паскаля из отдельно компилируемой процедуры не видны описания её программы-хозяина. Следовательно, правильнее принять, что SP зависит от описанного внутри него заголовка процедуры MIN, как от самостоятельного интерфейсного модуля MIN, телом которого служит отдельно компилируемый модуль $body\text{-}MIN = bMIN$. Итак, описания-заголовки функций $PRNT$ и MIN синтаксически выглядят совершенно аналогично, а в СПС интерпретируются по-разному. В данном случае это следствие ограниченности средств модульности в классическом Паскале.

Пример 2. Представим себе программу на языке C/C++ следующей структкры::

```
/* P.h ------------------------------------------------- */
 const int n= 20;
 typedef float [n]  T;
 void inpt (T);
void prnt (T);
/* end P.h ------------------------------------------------ */
/* S.c ------------------------------------------------- */
 #include "S.h"
 void sort (T a);
 { /* sorting */
 }
/* end P.h ------------------------------------------------ */
/* P.c ------------------------------------------------- */
 #include "S.h"
 extern void sort (T);
void prnt (T a)
{ sort (a);
```

```
   /* printing */
}
void inpt (T a)
{ /* reading */
}
/* end P.c ------------------------------------------- */
/* B.c ----------------------------------------------- */
#include "S.h"
int main ()
{ T A;
   inpt (A);
   prnt (A);
}
/* end B.c ------------------------------------------- */
```

Рассматривая файлы P.h и P.c как модули, приходим к выводу, что первый можно считать интерфейсным, обозначая, например, P, а второй – его телом bP. При этом с первым связываются объявления функций, как внутренние блоки P_{inpt}, P_{prnt} (интерфейсные), а со вторым – реализующие их тела $b(P_{inpt})$ и $b(P_{prnt})$ (определения тех же функций). Файл B.c представим реализующим модулем B, который зависит от P. Остаётся дать интерпретацию файлу S.c . По аналогии с предыдущим примером можно было бы описание функции sort в P. c счесть интерфейсным модулем, от которого зависит bP. Телом этого блока был бы собственный блок модуля, представляющего файл S.c. Но более естественно интерпретировать S.c как субмодуль модуля bP. Собственный блок этого субмодуля есть тело внутреннего блока, представляющего описание функции sort. Дело в том, что в данном случае директивы #include "S.h" обеспечивают разработку определения sort в контексте тех же базовых описаний, что и модуля bP. В то же время «зависимость от» модуля, являющегося функцией или процедурой, обычно предполагает только возможность вызова этой функции или процедуры (срав. с пунктом 5^0 определения 1. и предыдущим примером). Эту же 4-файловую программу можно представить с помощью СПС иначе, если, например, вспомнить, что по правилам данного языка она логически эквивалентна 3-файловой за счет «присоединения» файлов директивой #include. Такая неоднозначность связана с высокой универсальностью языка программирования при относительной скупости средств её достижения.

Пример 3. С помощью тех же заголовочных файлов можно проиллюстрировать отношение «имеет родителя»:

```
/* C.h ----------------------------------(модуль C) */
 const int n= 20;
/* end C.h ------------------------------------------ */
/* T.h ----------------------------------(модуль  T) */
#include "C.h"
 typedef float [n]  T;
/* end T.h -------------------------------------------*/
/* P.h ----------------------------------(модуль  P) */
#include "T.h"
 void inpt (T);
void prnt (T);
/* end P.h ------------------------------------------ */
```

Здесь P имеет родителя T, а T имеет родителя C.

Пример 4.

В языке Ада хватает средств, чтобы каждый вид отношений продемонстрировать с помощью отдельного средства:

```
   function I (String) return Positive is--modulo I
   ------------------------------------------ modulo bI
   -- declarations
   begin –statements
   end I; ----------------------------------------------
   with I;
   package C is --------------------- modulo C without body
    N: constant Natural:= I("#3");
   end C; ----------------------------------------------
   package C.T is  --------------------- modulo C.T
    type L is array(1..N) of Float;
    procedure Inp;
    procedure Out;
   end C.T; ---------------------------------------
   package body C.T is  ---------------------- modulo bC.T
    procedure Inp is
    -- declarations
    begin –statements
   end Inp;
    procedure Out is separate;
   end C.T; ---------------------------------------------
   with C.T; use C.T;
   procedure S (X: in out L);                -- modulo S
   ------------------------------------------------------
   separate (C.T)
   procedure Out is ---------- submodulo of bC.T
   -- declarations
   begin
   –statements
   end Out; ------------------------------------
   with C.T, S; use C.T;
   procedure Main is                          -- modulo Main
   ------------------------------------------- modulo bMain
   -- declarations
      procedure QS (X: in out L) is
      begin –statements of QS
      end QS;
   begin
   –statements of Main
   Bl:begin --potentially erroneous statements
   exception --handlers
   end Bl;
   --others statements and hadlers
   end Main; ------------------------------------------------
   with C;
   procedure S(X: in out L) is            -- modulo bS
   -- declarations
   begin –statements
end S; ---------------------------------------------
```

Модульная структура пояснена в комментариях.

Пример 5. Выше схемы программной структуры основывались на использовании в программах раздельной компиляции и внутренних блоков.

Однако построение СПС может исходить из других принципов, например, из логики объектно-ориентированного программирования, если она доминирует в данной программе. Например,

```
// file Lists.cpp --------------
class E {/*declarations*/}

class L
{public
   L(int);   void   inp(E);   E
out();void sort();
   friend bool P(E, L);
private
   int LEN:= 0;
   void inv();
   /*others*/
};

class S: public L
{public
   S(int); E out(); inline int
size() {return LEN;};
}

bool P(E, L) {if ( L_len~=0) /*
… */ }
```

```
/*The realization of the
complex of L-declarations: */
L::L(int n) {/* … */}

void L::inp(E) {/* … */}

E L::out(){/* … */}

void L::inv(){/* … */}

/*The realization of the
complex of S-declarations:*/
S::S(int n) {/* … */}

E S::out(){/* … */}/*-------*/
// end of file
// file sort.cpp that contains
// the most complex function
void L::sort(){/* … */}// the
realization of the member sort
of the class S
// end of file
```

Соответствующая СПС определяется в следующей таблице. Она могла быть такой же без выделения функции L::sort в отдельно компилируемый файл.

Таблица 1. Элементы СПС программы примера 5

Модуль	Первые 3 ПС собственного блока	Отношения с др.модулями	Внутр.блоки	Отношения с др.блоками
E (интерфейс)	class E {}			
L (интерфейс)	class L {}	зависит от E	L::L, L::inp, L::out, L::sort, L::inv (интерф.)	
S (интерфейс)	class S :	имеет родителя L	S:: S, S::out, S::size (интерф.)	
P (интерфейс)	bool P ()	имеет родителя L, зависит от E		
P (реализ.)	{} if ()	bP - тело P		
L (реализ.)	L :: L	bL - тело L	L::L , L::inp , L::out , L::inv (реализ.)	тела : b(L::L), b(L::inp), b(L::out), b(L::inv)

S (реализ.)	S :: S	bS - тело S	S::S , S::out (реализ.)	тела : b(S::S), b(S::out)
L::sort (реализ.)	void L ::	субмодуль: s(sort)=bS		<L::sort> = bL_{sort} - тело

Такой же (с точностью до пограммных символов) СПС может быть представлена аналогичная объектно-ориентированная программа на языке Ада. Но правила данного языка налагают условия согласованности логической структуры программы со структурой её компиляции (модули E, L, L.S, L.P, L.Sort обязательно должны компилироваться раздельно).

Предыдущие примеры подсказывают рецепты трактовки наиболее популярных средств структуризации и объектной ориентации языков программирования с точки зрения СПС. Однако интерпретация в форме СПС программ, использующих весьма специфические возможности языков программирования, а также поверхностный взгляд на использованные в программах специальные средства может приводить к определённым трудностям. Выходом служит разумное «огрубление» специфических особенностей программы при её представлении в форме СПС.

Пример 6. В C++ программе из предыдущего примера предполагалось, что функция P является другом только класса L , но на практике весьма характерно, что функция (например, бинарная операция пользователя) выступает другом нескольких классов, в нашем случае - E, L. Тогда можно, например, считать P самостоятельным модулем, который зависит от всех этих модулей-классов (в нашем случае от E и L). Если функция-друг проста (как часто бывает), то потеря информации о том, что P имеет доступ к приватным частям E и L мало сказывается на метрических характеристиках. Хуже дело, если активно используется множественное наследование классов. Здесь замена связи «имеет родителя» на «зависит от» не столь безобидна. Мы не сочли целесообразным приспосабливать определение СПС к множественному наследованию C++, учитывая, что связанные с ним сложности в программировании вызвали к жизни тенденцию избавляться в более современных языках и системах программирования от его использования вообще.

П*ример 7.* Рассмотрим фрагмент программы на языке Ада:

```
task T is--------   --------------------------- the modulo
 entry E (X: T1; Y: T2; Z,W:  out T3); -- the block E
end T;-----------------------------------------------------

task body T is ---------------------- the body of the modulo
 A: T1; A: T2; C,D:  out T3
begin
loop   C:= Expretion_1; D:= Expretion_2; -- …
    accept E (X: T1; Y: T2; Z,W:  out T3) do -- accept statement
       A:= X; B:= Y; Z:= C; W:= D; -- Is this the body of E?
    end E; ------------------------------------------------
…
end loop;
end T;-----------------------------------------------------
```

Поскольку задача в Аде может рассматриваться как процедура, выполняемая (параллельно с другими задачами) на отдельном процессоре, а описание входа

трактуется в точности как описание процедуры, то может показаться, что оператор принятия входа E в теле задачи T может рассматриваться как тело входа E. Это была бы ошибка, так как операторов принятия одного входа может быть сколько угодно, а их обрабатывающие части не обязаны совпадать. Однако было бы хуже пытаться рассматривать в качестве тела объединение таких операторов. Правильное решение – задачный вход вообще не имеет тела: операторы принятия в практике реального программирования тривиальны (так или иначе, но производят копирование параметров).

Определение 2. Скажем, что *статус блоков СПС определён*, если верно следующее. Множества внутренних блоков каждого модуля СПС из тех модулей, что не являются ничьим телом и ничьим субмодулем, разбиты на непересекающиеся группы, каждая из которых помечена, как *автономная*, *структурная* или *усложнённаяя*. При этом автономная группа, если она есть, то только одна, и она содержит остаточный блок данного модуля, если он не пуст.

Отметим, что статус усложненной может иметь, например, группа функций, связанных взаимной рекурсией.

Пример 8. Мы видели, что конструкция пакета в языке Ада обычно является кандидатом на представление модулем СПС. Рассмотрим пакет, в котором определяются несколько типов:

```
package P is ------------------ ------------------ modulo P
type T1 is private; -- 1
---------- groop G1:
function Init_1 (t: integer) return T1;
function "+" (Left, Right: T1) return T1;
-- …
type T2 is private; -- 2
------------- groop G2:
function Init_2 (x: float) return T2;
function "+" (Left, Right: T2) return T2;
-- …
-- next blocks belong to a groop X:
function T1_to_T2 (t: T1) return T2;
function T2_to_T1 (x: T2) return T1;
Convert_Error: exception; -- 3
end P; ----------------------         ----------------------
```

Блок, представляющий функцию Init_1 и прочие, которые представляют операции над типом T1, естественно отнести к одной структурной группе (G1), аналогично для операций над типом T2. Программные символы описаний --1, --2, --3 входят в остаточный блок, обозначаемый |P>. К какой группе отнести блоки операций преобразования типов, решается в зависимости от конкретных обстоятельств. Если эти преобразования (как обычно) тривиальны, то их можно счесть автономными и положить $X = (T1\_to\_T2 \quad T2\_to\_T1 \quad |P>)$. В противном случае, если бы имелись другие совместные нетривиальные операции над типами T1, T2, то или $X = (T1\_to\_T2 \quad T2\_to\_T1)$ - ещё одна структурная группа, или структурная группа вообще одна: $G1 = G2 = X$.

Пример 9. Может сложиться впечатление, что группировка блоков – это дополнительное средство для уточнения схемы программной структуры в тех случаях, когда программа не содержит однозначных синтаксических признаков фактически имеющейся структурности. В следующем эскизе фрагмента

программы на языке C++ группировка помогает снять определённые синтаксические рамки, сохраняя при этом в СПС некоторую информацию об имевшейся логической структуре.

```
#include math.h
void f ()
{ class P
   {   float x, y, z;
public: P(float a, float b, float c) {x=a; y=b; z=c;};
 float X() {return x};
 // …
   }
   class V : public P
   {   float len () {return sqrt(x*x+y*y+z*z)};
// …
   }
   //others
}
```

Здесь классы P и V локальны и служат удобству обозначений в теле функции f. Нет смысла представлять их внутренними блоками модуля f . Лучше представить блоками в составе автономной группы модуля f функции P::P, P::X, ... и V::len, ... Если бы они обрабатывали данные-члены классов более сложным образом (что, впрочим, не характерно для локальных классов), то можно было бы присвоить данной группе статус структурной

Определение 3. *Выявление параметров ввода-вывода* блока означает (в соответствии со смыслом представляемого им кода) сопоставление ему величины $\eta^* > 1$ по следующим правилам.

1^0. Интерфейсному блоку и его телу сопоставляется одна и та же величина.

2^0. Пусть часть программы, представленной интерфейсным блоком, имеет p описанных параметров (любого вида и типа). И пусть j - число объектов внешнего окружения программы, которые не представлены явно параметрами, но с которыми можно непосредственно работать через представленные в интерфейсе операции (если такие есть). Тогда

$$\eta^* = 2 + p + j \tag{4}$$

3^0. Пусть реализующий блок, не являющийся ничьим телом, непосредственно осуществляет ввод-вывод p разнотипных объектов, а также однотипный обмен с j внешними объектами. Тогда η^* снова определяется (5).

Рассмотрим для модуля или блока M величину его *объема* по Холстеду

$$V_M = V(N, \eta) = N \log \eta \ , \tag{5}$$

где N - общее число ПС данного блока или блока $< M >$ (если M модуль);

η - число различных ПС (мощность использованного алфавита ПС).

Потенциальный объём определён в [3] зависимостью от числа выявленных параметров ввода-вывода η^*:

$$V_M^* = V^*(\eta^*) = V(\eta^*, \eta^*) \tag{6}$$

Холстед предложил также количественную характеристику λ *уровня языка* программирования, статистический подход к её оценке и ввел для одномодульных неструктурированных программ выражение

$$E(M) = E^*(V_M^*, \lambda) = (V_M^*)^3 / \lambda^2 \quad , \tag{7}$$

которое рассматривал как оценку работы по программированию.

В [2] (7) трактуется как аналог внутренней энергии. Мы обобщим (7) так, чтобы с усложнением внутренней структуры модуля его энергия возрастала.

Определение 4. *Спецификационной энергией* СПС (сокр. – *энергией*) называется неотрицательная величина E, которая присваивается, как СПС в целом, так и её элементам (модулям, группам и блокам автономной группы) и вычисляется следующим образом.

1^0. Спецификационная энергия СПС равна сумме спецификационных энергий составляющих её модулей.

2^0. Спецификационная энергия модуля, который является телом или субмодулем другого модуля полагается равной нулю.

3^0 Спецификационная энергия интерфейсного модуля или реализующего модуля, который не является телом или субмодулем другого модуля, равна сумме спецификационных энергий составляющих его групп.

4^0. Спецификационная энергия группы $B_1, B_2, \ldots$ блоков одного из указанных в 3^0 модулей вычисляется при условии выявленности параметров ввода-вывода этих блоков и в зависимости от статуса группы. Если группа имеет статус автономной, то её энергия равна сумме энергий блоков и равна

$$E(\{B_i\}) = \sum_i E(B_i) = \sum_i E^*(V_i^*, \lambda_i) \tag{8}$$

Если группа имеет статус структурной, то её энергия вычисляется на основе принципа суммирования потенциальных объёма блоков:

$$E(\{B_i\}) = E^*(\sum_i V_i^*, \lambda) \tag{9}$$

Если группа имеет статус иерархической, то её энергия вычисляется на основе принципа объединения параметров

$$E\{B_i\} = E^*(V^*(\sum_i \eta_i^*), \lambda) \tag{10}$$

Из определений видно, что располагая исходным текстом программы и приняв на основании стилей языка и программы решения, связанные с представлением в форме СПС, спецификационную энергию можно вычислить однозначно. В этом смысле она является функцией состояния программы.

4. Модель процесса разработки ПО в терминах энергетических метрик

Обощим объем программы и работу в программировании [3] так, чтобы они учитывали размер контекста, в котором программируется данный модуль.

Определение 4. При наличии определённой информации о процессе разработки программы, представленной СПС, можно определить *объём разработки* W_M произвольного модуля M, величину *работы программирования* такого модуля и работу программирования СПС в целом:

$$A_M = A(W_M, V^*) = W_M^2 / V_M^*, \tag{11}$$

$$A = \sum_M A_M \tag{12}$$

где W_M определяются следующим образом:

1^0. Пусть M - интерфейсный модуль, который имеет родителя M_0, зависит от модулей $M_1, .. M_m$ (не универсальных) и универсальных модулей $U_1, .. U_n$. Допустим, что $M_1, .. M_{m0}$ разрабатывались раньше M, $M_{m0+1}, .. M_{m1}$ одновременно с M, а $M_{m1+1}, .. M_{m2}$ - позже. Тогда в обозначениях (5)

$$W_M = \sum_1^{m0} V_i + V(N, \eta + \eta_0 + \sum_{m0+1}^{m1} \eta_i) + \sum_{n0+1}^{n1} V_j, \tag{13}$$

где V_i и V_j обозначают объёмы соответственно M_i и U_j, η_i - число различных ПС в $< M_i >$.

2^0. Пусть M - реализующий модуль, который является телом модуля M_0 или имеет надмодуль и служит реализацией блока, водящего в группу M_0 (в последнем случае $M_0 = \{B_0, B_1, ...\}$, $< M >= bB_0$). Модули, от которых зависит M, обозначим так же, как в предыдущем пункте. Тогда

$$W_M = \sum_1^{m0} V_i + V(N, \eta + \overline{\eta}_0 + \sum_{m0+1}^{m1} \eta_i) + \sum_{n0+1}^{n1} \chi_j \cdot V_j, \tag{14}$$

где $\overline{\eta}_0$ - количество различных ПС, присутствующих в M_0 и отсутствующих в M; χ_j - индикатор отсутствия V_j в выражении для объёма M_0 или его предков в соответствии с 1^0 (если M_0 модуль) или в выражениях объёма

надмодулей M любого порядка в соответствии с данным правилом (если M_0 группа блоков).

3^0. Пусть, наконец, M - интерфейсный модуль, разрабатываемый позже ряда модулей, в контексте которых находится. Пусть это будут интерфейсные модули $M_1, .. M_m$ и такие реализующие модули $B_1, .. B_n$, что переходя от них к их надмодулям и т.д., затем к некоторому телу, его предку и т.д., - нельзя встретить модуль, в контексте которого также находится M. Тогда

$$W_M = \sum_1^{\eta} (1 + k_l), \tag{15}$$

где k_l - количество тех модулей из числа $M_1, .. M_m$ и $B_1, .. B_n$, в собственных блоках которых встречается l-й ПС из числа η различных ПС блока $< M >$.

Рассмотрим процесс разработки программной системы P, который отражается последовательными версиями представляющей её СПС, занумерованными от 0 до N. Тогда для работ A_k, подсчитанных для k-х версий с учетом последовательности разработки (и переработки) модулей имеем

$$A(P) = A_N = \sum_1^N \Delta A_k, \tag{16}$$

где $\Delta A_k = A_k - A_{k-1}$, $A_0 = 0$. Но трудозатраты в группе разработчиков D равны

$$A(D) = \sum_1^N |\Delta A_k| \tag{17}$$

и напрашивается ввести коэффициент *трудности разработки*

$$d = A(D)/A(P) \quad \geq 1, \tag{18}$$

отражающий то, как часто и много приходилось «чистить» уже написанный код.

Для разработок программных систем в целях поддержки научных исследований не характерно сохранение промежуточных версий, а тогда коэффициент трудности (18) непредсказуемо занижается. В частности, при $N = 1$ он совершенно неинформативен. Поэтому рассмотрим параметр, позволяющий анализировать процесс разработки в любом случае. Величину

$$\Delta Q = \Delta E - \Delta A \tag{19}$$

назовём *информационным теплом*, и введём две взаимозаменяемые метрики

$$\rho = \Delta A/\Delta E, \quad \kappa = \Delta Q/\Delta E \tag{20}$$

(коэффициент *соответствия сложности* и коэффициент *притока информации*).

Если структура и интерфейс P обогащены без особых изменений реализационной части, то $\Delta Q > 0, \rho < 1$, что, таким образом, соответствует закачке информации в P за счёт знаний D. Пусть, напротив, $Q < 0, \rho > 1$. Это возможно при большой работе программирования по сравнению со структурными изменениями. Значит, системные знания D использовались мало, а за счет большой работы по реализации программных решений интенсивно накапливался опыт – информация преимущественно поступала от P к D. Если на каком-то этапе изменение спецификационной энергии соответствует работе программирования, то $\rho = 1, Q = 0$. Это объясняет названия величин (19)-(20).

Вышесказанное предполагает такую нормировку величин, что в среднем

$$E(M) = A_M \tag{21}$$

по представительной выборке готовых программ M, в которых реализация признаётся соответствующей их спецификации по интерфейсу. Очевидно, за такую нормировку должен отвечать коэффициент λ из (7) – «уровень языка». В [3] приводятся результаты статистического определения уровней разных языков, свидетельствующие о том, что данная величина действительно может рассматриваться как характеристика языка программирования. Правда, использовались выборки малых по сегодняшним меркам программ, и стабильность λ с ростом размера программ нуждается в дальнейшей проверке.

Пример 10. По данным (7) видно, что авторы дипломных работ по приложениям методов дискретных особенностей на первых порах сталкивались с трудностями программной реализации ($\rho >> 1$), а по мере накопления коллективного опыта стал расти объём программ и они, напротив, стали источником полезной информации о возможностях разработки более профессионального ПО. При сохранении обнаруженной в (7) регрессионной зависимости A на E величина ρ, уменьшаясь, должна была бы стремиться к предельному значению, равному примерно $0{,}3$.

Для контроля за внутренним качеством ПО при его разработке вводим

$$Q_k = E_k - A_k \quad (k = 1..N) \tag{22}$$

и рекомендуем сопоставить моменты $k(i)$ перемен знака $Q_{k(i)}$ с минуса на плюс этапам структурной перестройки разрабатываемой системы. При отсутствии соответствия между ними следует тщательно проверить, не сталкивается ли разработка с неудачными проектными решениями, не попала ли в полосу массового выявления и текущего исправления ошибок.

5. Заключительные выводы

Построена в стиле Software Science новая модель энергетики процесса разработки сложных программных систем, имеющая акцент на учёте внешних и внутренних связей программных модулей. Модель свободна от правил и допущений конкретных языков программирования. Но, как обосновывается

системой примеров на языках C++ и Ада, требования модели применительно к конкретным программа на конкретных языках приводят к интерпретации программной структуры в форме СПС практически однозначно.

Разработанная модель будет применена в программометрическом анализе качества современного программного обеспечения числительных экспериментов, проводимых на базе методов дискретных особенностей. Именно проблемы с применимостью прежних моделей Software Science и вызвали данную работу к жизни. Нет принципиальных препятствий к использованию её результатов для анализа качества и надёжности сложных систем ПО любого назначения.

Но для применений к большим программным системам, целесообразно провести дальнейшее исследование стабильности используемого холстедовского параметра - уровня языка программирования.

ЛИТЕРАТУРА

1. Гандель Ю.В., Мищенко В.О. Математическое моделирование в электродинамике на базе сингулярных интегральных уравнений и проект программной системы. // Математическое моделирование. Сб.науч.тр./НАН Украины. Ин-т математики. – Киев, 1996. – С.70-74.
2. Мищенко В.О. Применение математического моделирования в системном анализе проекта программного обеспечения методом дискретных особенностей // Труды VII Международного «Методы дискретных особенностей в задачах математической физики». – Феодосия, 1997. – С.117-120.
3. Холстед М.Х. Начала науки о программах.- М.:, Финансы и статистика, 1981,- 128 с.
4. Харченко В.С,, Скляр В.В., Тарасюк О.М, Методы моделирования и оценки качества и надёжности программного обеспечения. – учеб.пособие.- Харьков: Нац. аэрокосм. ун-т «Харьк.авиац.ин-т». 2004.- 159 с.
5. Коган Б.И. Экспериментальные исследования программ.- М.: Наука, 1988.- 184 с.
6. Уточнение математической модели Холстеда и её развитие для метрического описания современных компьютерных программ. // Математическое моделирование. Сб.науч.тр./НАН Украины. Ин-т математики. – Киев, 1996. – С.180-182.
7. Гахов А.В., Мищенко В.О. Вычислительный эксперимент на базе численного решения гиперсингулярного интегрального уравнения для прямоугольной области // Вісник Харківського національного університету. № 595, Серія «Математичне моделювання. Інформаційні технології. Автоматизовані системи управління», вип. 1. – С. 84-91.