ОСНОВЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНЫХ СИСТЕМ

Рассматривается содержание этапа проектирования и его место в жизненном цикле конструирования программных систем. Дается обзор архитектурных моделей ПО, обсуждаются классические проектные характеристики: модульность, информационная закрытость, сложность, связность, сцепление и метрики для их оценки.

Особенности процесса синтеза программных систем

Если в ходе анализа ищется ответ на вопрос: «Что должна делать будущая система?», то в процессе синтеза формируется ответ на вопрос: «Каким образом система будет реализовывать предъявляемые к ней требования?». Выделяют три этапа синтеза: проектирование ПС, кодирование ПС, тестирование ПС (рис. 1).

Рассмотрим информационные потоки процесса синтеза.

Этап проектирования питают требования к ПС, представленные информационной, функциональной и поведенческой моделями анализа. Иными словами, модели анализа поставляют этапу проектирования исходные сведения для работы. Информационная модель описывает информацию, которую, по мнению заказчика, должна обрабатывать ПС. Функциональная модель определяет перечень функций обработки. Поведенческая модель фиксирует желаемую динамику системы (режимы ее работы). На выходе этапа проектирования — разработка данных, разработка архитектуры и процедурная разработка ПС.

Разработка данных — это результат преобразования информационной модели анализа в структуры данных, которые потребуются для реализации программной системы.

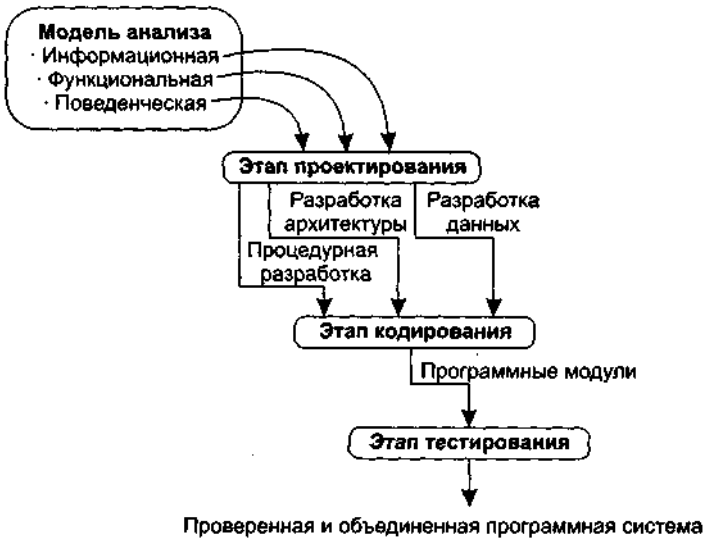

Рис. 1. Информационные потоки процесса синтеза ПС

Разработка архитектуры выделяет основные структурные компоненты и фиксирует связи между ними.

Процедурная разработка описывает последовательность действий в структурных компонентах, то есть определяет их содержание.

Далее создаются тексты программных модулей, проводится тестирование для объединения и проверки ПС. На проектирование, кодирование и тестирование приходится более 75% стоимости конструирования ПС. Принятые здесь решения оказывают решающее воздействие на успех реализации ПС и легкость, с которой ПС будет сопровождаться.

Следует отметить, что решения, принимаемые в ходе проектирования, делают его стержневым этапом процесса синтеза. Важность проектирования можно определить одним словом — качество. Проектирование — этап, на котором «выращивается» качество разработки ПС. Справедлива следующая аксиома разработки: может быть плохая ПС при хорошем проектировании, но не может быть хорошей ПС при плохом проектировании.

Особенности этапа проектирования

Проектирование — итерационный процесс, при помощи которого требования к ПС транслируются в инженерные представления ПС. Вначале эти представления дают только концептуальную информацию (на высоком уровне абстракции), последующие уточнения приводят к формам, которые близки к текстам на языках программирования.

Обычно в проектировании выделяют две ступени: предварительное проектирование и детальное проектирование. Предварительное проектирование формирует абстракции архитектурного уровня, детальное проектирование уточняет эти абстракции, добавляет подробности алгоритмического уровня. Кроме того, во многих случаях выделяют интерфейсное проектирование, цель которого — сформировать графический интерфейс пользователя (GUI).

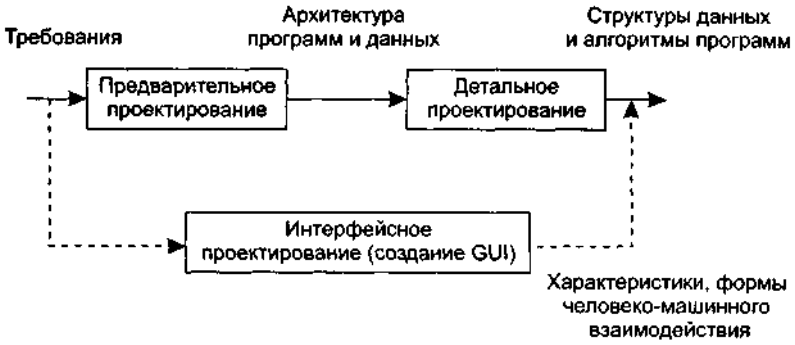

Рис. 2. Информационные связи процесса проектирования

Предварительное проектирование обеспечивает:

- идентификацию подсистем;
- определение основных принципов управления подсистемами, взаимодействия подсистем.

Предварительное проектирование включает три типа деятельности:

1. *Структурирование системы.* Система структурируется на несколько подсистем, где под подсистемой понимается независимый программный компонент. Определяются взаимодействия подсистем.
2. *Моделирование управления.* Определяется модель связей управления между частями системы.
3. *Декомпозиция подсистем на модули.* Каждая подсистема разбивается на модули. Определяются типы модулей и межмодульные соединения.

Рассмотрим вопросы структурирования, моделирования и декомпозиции более подробно.

Структурирование системы

Известны четыре модели системного структурирования:

- модель хранилища данных;
- модель клиент-сервер;
- трехуровневая модель;
- модель абстрактной машины.

В *модели хранилища данных* (рис. 3) подсистемы разделяют данные, находящиеся в общей памяти. Как правило, данные образуют БД. Предусматривается система управления этой базой.

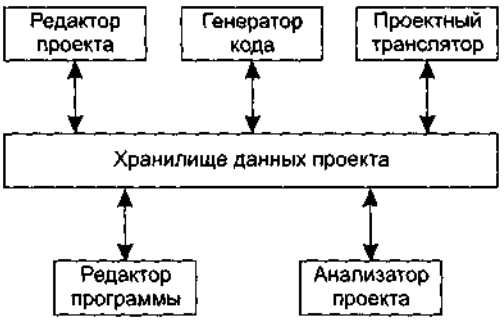

Рис. 3. Модель хранилища данных

Модель клиент-сервер используется для распределенных систем, где данные распределены по серверам (рис. 4). Для передачи данных применяют сетевой протокол, например TCP/IP.

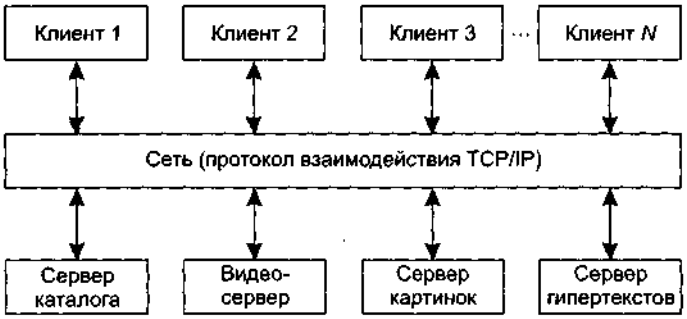

Рис. 4. Модель клиент-сервер

Трехуровневая модель является развитием модели клиент-сервер (рис. 5).

Рис. 5. Трехуровневая модель

Уровень графического интерфейса пользователя запускается на машине клиента. Бизнес-логику образуют модули, осуществляющие функциональные обязанности системы. Этот уровень запускается на сервере приложения. Реляционная СУБД хранит данные, требуемые уровню бизнес-логики. Этот уровень запускается на втором сервере — сервере базы данных.

Преимущества трехуровневой модели:

- упрощается такая модификация уровня, которая не влияет на другие уровни;
- отделение прикладных функций от функций управления БД упрощает оптимизацию всей системы.

Модель абстрактной машины отображает многослойную систему (рис. 6).

Каждый текущий слой реализуется с использованием средств, обеспечиваемых слоем-фундаментом.

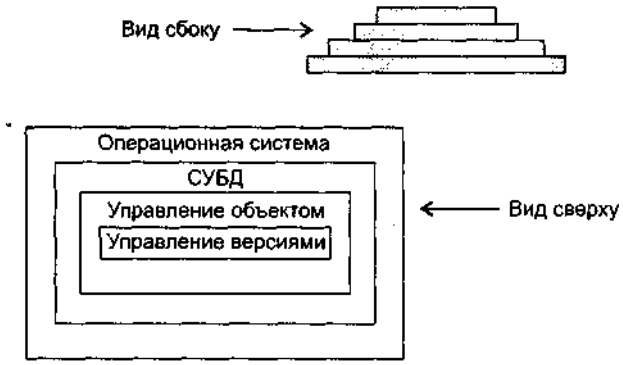

Рис. 6. Модель абстрактной машины

Моделирование управления

Известны два типа моделей управления:

- модель централизованного управления;
- модель событийного управления.

В модели централизованного управления одна подсистема выделяется как системный контроллер. Ее обязанности — руководить работой других подсистем. Различают две разновидности моделей централизованного управления: *модель вызов-возврат* (рис. 7) и *Модель менеджера* (рис. 8), которая используется в системах параллельной обработки.

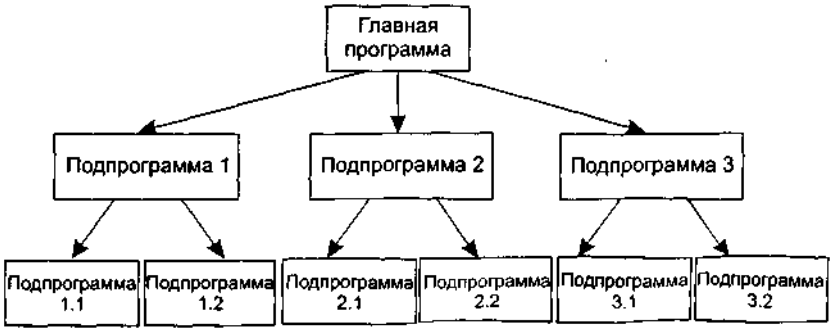

Рис. 7. Модель вызов-возврат

В модели событийного управления системой управляют внешние события. Используются две разновидности модели событийного управления: широковещательная модель и модель, управляемая прерываниями.

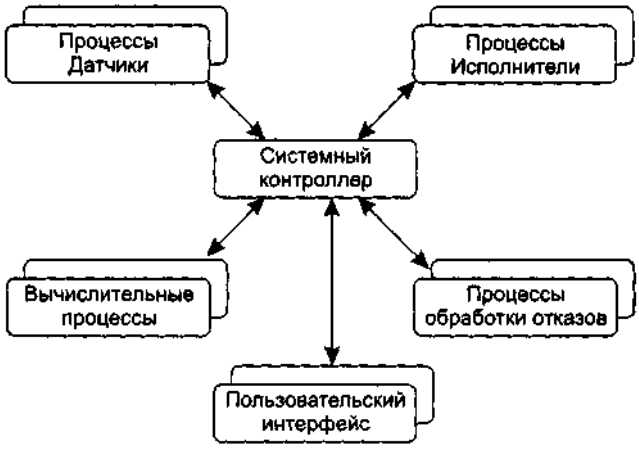

Рис. 8. Модель менеджера

В *широковещательной модели* (рис. 9) каждая подсистема уведомляет обработчика о своем интересе к конкретным событиям. Когда событие происходит, обработчик пересылает его подсистеме, которая может обработать это событие. Функции управления в обработчик не встраиваются.

Рис. 9. Широковещательная модель

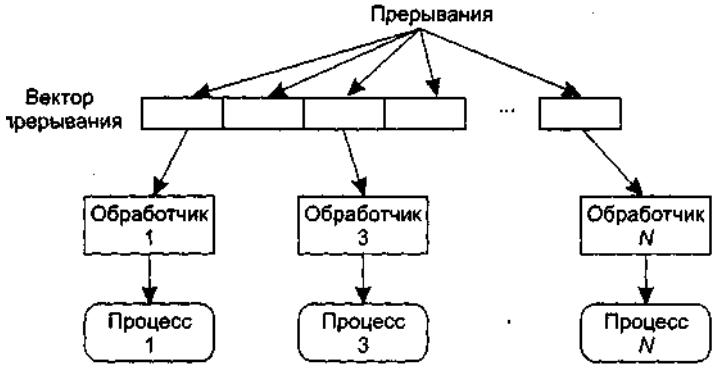

Рис. 10. Модель, управляемая прерываниями

В модели, управляемой прерываниями (рис. 10), все прерывания разбиты на группы — типы, которые образуют вектор прерываний. Для каждого типа прерывания есть свой обработчик. Каждый обработчик реагирует на свой тип прерывания и запускает свой процесс.

Декомпозиция подсистем на модули

Известны два типа моделей модульной декомпозиции:

- модель потока данных;
- модель объектов.

В основе модели потока данных лежит разбиение по функциям.

Модель объектов основана на слабо сцепленных сущностях, имеющих собственные наборы данных, состояния и наборы операций.

Очевидно, что выбор типа декомпозиции должен определяться сложностью разбиваемой подсистемы.

Модульность

Модуль — фрагмент программного текста, являющийся строительным блоком для физической структуры системы. Как правило, модуль состоит из интерфейсной части и части-реализации.

Модульность — свойство системы, которая может подвергаться декомпозиции на ряд внутренне связанных и слабо зависящих друг от друга модулей.

По определению Г. Майерса, модульность — свойство ПО, обеспечивающее интеллектуальную возможность создания сколь угодно сложной программы. Проиллюстрируем эту точку зрения.

Пусть $C(x)$ — функция сложности решения проблемы x, $T(x)$ — функция затрат времени на решение проблемы x. Для двух проблем p_1 и p_2 из соотношения $C(p_1) > C(p_2)$ следует, что

$$T(p_1)>T(p_2). \quad (1)$$

Этот вывод интуитивно ясен: решение сложной проблемы требует большего времени.

Далее. Из практики решения проблем человеком следует:

$$C(p_1+ p_2)>C(p_1) + C(p_2).$$

Отсюда с учетом соотношения (1) запишем:

$$T(p_1+p_2)>T(p_1) + T(p_2). \quad (2)$$

Соотношение (2) приводит к заключению— сложную проблему легче решить, разделив ее на управляемые части. Результат, выраженный неравенством (2), имеет важное значение для модульности и ПО. Фактически, это аргумент в пользу модульности.

Однако здесь не учитываются затраты на межмодульный интерфейс. Как показано на рис. 11, с увеличением количества модулей (и уменьшением их размера) эти затраты также растут.

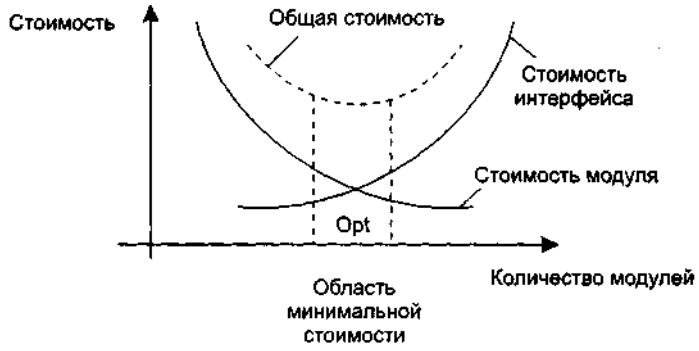

Рис. 11. Затраты на модульность

Таким образом, существует оптимальное количество модулей Opt, которое приводит к минимальной стоимости разработки. Увы, у нас нет необходимого опыта для гарантированного предсказания Opt. Впрочем, разработчики знают, что оптимальный модуль должен удовлетворять двум критериям:

- снаружи он проще, чем внутри;
- его проще использовать, чем построить.

Информационная закрытость

Принцип информационной закрытости (автор — Д. Парнас, 1972) утверждает: содержание модулей должно быть скрыто друг от друга. Как показано на рис. 12, модуль должен определяться и проектироваться так, чтобы его содержимое (процедуры и данные) было недоступно тем модулям, которые не нуждаются в такой информации (клиентам).

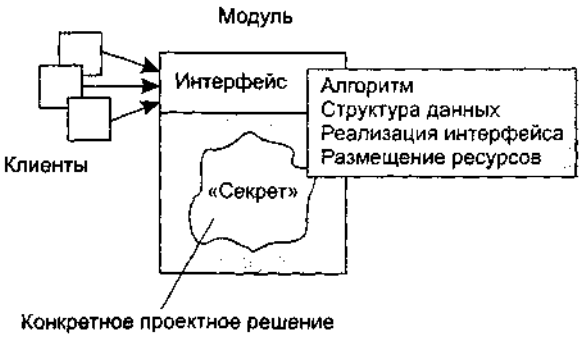

Рис. 12. Информационная закрытость модуля

Информационная закрытость означает следующее:

1) все модули независимы, обмениваются только информацией, необходимой для работы;
2) доступ к операциям и структурам данных модуля ограничен.

Достоинства информационной закрытости:

- обеспечивается возможность разработки модулей различными, независимыми коллективами;
- обеспечивается легкая модификация системы (вероятность распространения ошибок очень мала, так как большинство данных и процедур скрыто от других частей системы).

Идеальный модуль играет роль «черного ящика», содержимое которого невидимо клиентам. Он прост в использовании — количество «ручек и органов управления» им невелико (аналогия с эксплуатацией телевизора). Его легко развивать и корректировать в процессе сопровождения программной системы. Для обеспечения таких возможностей система внутренних и внешних связей модуля должна отвечать особым требованиям. Обсудим характеристики внутренних и внешних связей модуля.

Связность модуля

Связность модуля (Cohesion) — это мера зависимости его частей. Связность — внутренняя характеристика модуля. Чем выше связность модуля, тем лучше результат проектирования, то есть тем «черней» его ящик (капсула, защитная оболочка модуля), тем меньше «ручек управления» на нем находится и тем проще эти «ручки».

Для измерения связности используют понятие силы связности (СС). Существует 7 типов связности:

1. **Связность по совпадению** (СС=0). В модуле отсутствуют явно выраженные внутренние связи
2. **Логическая связность** (СС=1). Части модуля объединены по принципу функционального подобия.
3. **Временная связность** (СС=3). Части модуля не связаны, но необходимы в один и тот же период работы системы.
4. **Процедурная связность** (СС=5). Части модуля связаны порядком выполняемых действий, реализующих некий сценарий поведения.
5. **Коммуникативная связность** (СС=7). Части модуля связаны по данным (работают с одной и той же структурой данных).
6. **Информационная (последовательная) связность** (СС=9). Выходные данные одной части используются как входные в другой части модуля.
7. **Функциональная связность** (СС=10). Части модуля вместе реализуют одну функцию.

Определение связности модуля

1. Если модуль — единичная проблемно-ориентированная функция, то уровень связности — функциональный; конец алгоритма. В противном случае перейти к пункту 2.
2. Если действия внутри модуля связаны, то перейти к пункту 3. Если действия внутри модуля никак не связаны, то перейти к пункту 6.
3. Если действия внутри модуля связаны данными, то перейти к пункту 4. Если действия внутри модуля связаны потоком управления, перейти к пункту 5.
4. Если порядок действий внутри модуля важен, то уровень связности — информационный. В противном случае уровень связности — коммуникативный. Конец алгоритма.
5. Если порядок действий внутри модуля важен, то уровень связности — процедурный. В противном случае уровень связности — временной. Конец алгоритма.
6. Если действия внутри модуля принадлежат к одной категории, то уровень связности — логический. Если действия внутри модуля не принадлежат к одной категории, то уровень связности — по совпадению. Конец алгоритма.

Возможны более сложные случаи, когда с модулем ассоциируются несколько уровней связности. В этих случаях следует применять одно из двух правил:

Правило параллельной цепи:
Если все действия модуля имеют несколько уровней связности, модулю присваивают самый сильный уровень связности.

Правило последовательной цепи:
Если действия в модуле имеют разные уровни связности, то модулю присваивают самый слабый уровень связности.

Сцепление модулей

Сцепление (Coupling) — мера взаимозависимости модулей по данным. Сцепление — внешняя характеристика модуля, которую желательно уменьшать.

Количественно сцепление измеряется степенью сцепления (СЦ). Выделяют 6 типов сцепления:

1. Сцепление по данным (СЦ=1). Модуль А вызывает модуль В. Все входные и выходные параметры вызываемого модуля – простые элементы данных.
2. Сцепление по образцу (СЦ=3). В качестве параметров используются структуры данных.
3. Сцепление по управлению (СЦ=4). Модуль А явно управляет функционированием модуля В, посылая ему управляющие данные.
4. Сцепление по внешним ссылкам (СЦ=5). Модули А и В ссылаются на один и тот же глобальный элемент данных.
5. Сцепление по общей области (СЦ=7). Модули разделяют одну и ту же глобальную структуру данных.
6. Сцепление по содержанию (СЦ=9). Один модуль прямо ссылается на содержание другого модуля (через его точку входа).
7.

Сложность программной системы

В простейшем случае сложность системы определяется как сумма мер сложности ее модулей. Сложность модуля может вычисляться различными способами.

Например, М. Холстед (1977) предложил меру длины N модуля:

$$N \approx n_1 log_2 (n_1) + n_2 log_2 (n_2),$$

где n_1 — число различных операторов, n_2 — число различных операндов.

В качестве второй метрики М. Холстед рассматривал объем V модуля (количество символов для записи всех операторов и операндов текста программы):

$$V = N \times log_2 (n_1 + n_2).$$

Вместе с тем известно, что любая сложная система состоит из элементов и системы связей между элементами и что игнорировать внутрисистемные связи неразумно.

Том МакКейб (1976) при оценке сложности ПС предложил исходить из топологии внутренних связей. Для этой цели он разработал метрику цикломатической сложности:

$$V(G) = E\text{-}N + 2,$$

где E — количество дуг, а N — количество вершин в управляющем графе ПС. Это был шаг в нужном направлении. Дальнейшее уточнение оценок сложности потребовало, чтобы каждый модуль мог представляться как локальная структура, состоящая из элементов и связей между ними.

Таким образом, при комплексной оценке сложности ПС необходимо рассматривать меру сложности модулей, меру сложности внешних связей (между модулями) и меру сложности внутренних связей (внутри модулей). Традиционно со внешними связями сопоставляют характеристику «сцепление», а с внутренними связями — характеристику «связность».

Характеристики иерархической структуры программной системы

Иерархическая структура программной системы — основной результат предварительного проектирования. Она определяет состав модулей ПС и управляющие отношения между модулями. В этой структуре модуль более высокого уровня (начальник) управляет модулем нижнего уровня (подчиненным).

Иерархическая структура не отражает процедурные особенности программной системы, то есть последовательность операций, их повторение, ветвления и т. д. Рассмотрим основные характеристики иерархической структуры, представленной на рис. 1.

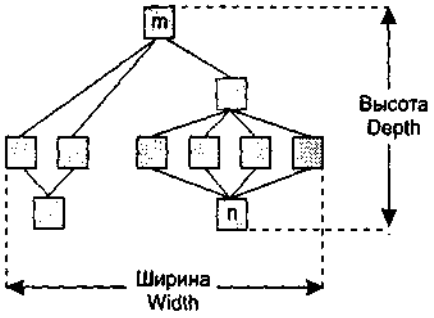

Рис. 1. Иерархическая структура программной системы

Первичными характеристиками являются количество вершин (модулей) и количество ребер (связей между модулями). К ним добавляются две глобальные характеристики — высота и ширина:

- **высота** — количество уровней управления;
- **ширина** — максимальное из количеств модулей, размещенных на уровнях управления.

Локальными характеристиками модулей структуры являются коэффициент объединения по входу и коэффициент разветвления по выходу.

Коэффициент объединения по входу Fan_in(i) — это количество модулей, которые прямо управляют i-м модулем.

Коэффициент разветвления по выходу Fan_out(i) — это количество модулей, которыми прямо управляет i-й модуль.

Возникает вопрос: как оценить качество структуры? Из практики проектирования известно, что лучшее решение обеспечивается иерархической структурой в виде дерева.

Степень отличия реальной проектной структуры от дерева характеризуется невязкой структуры. Как определить невязку?

Вспомним, что полный граф (complete graph) с n вершинами имеет количество ребер

$$e_c = n(n\text{-}1)/2,$$

а дерево (tree) с таким же количеством вершин — существенно меньшее количество ребер

$$e_t = n\text{-}1.$$

Тогда формулу невязки можно построить, сравнивая количество ребер полного графа, реального графа и дерева.

Для проектной структуры с n вершинами и e ребрами невязка определяется по выражению

$$Nev = \frac{e - e_t}{e_c - e_t} = \frac{(e - n + 1) \times 2}{n \times (n-1) - 2 \times (n-1)} = \frac{2 \times (e - n + 1)}{(n-1) \times (n-2)}.$$

Значение невязки лежит в диапазоне от 0 до 1. Если Nev = 0, то проектная структура является деревом, если Nev = 1, то проектная структура — полный граф.

Ясно, что невязка дает грубую оценку структуры. Для увеличения точности оценки следует применить характеристики связности и сцепления.

Хорошая структура должна иметь низкое сцепление и высокую связность.

Л. Констентайн и Э. Йордан (1979) предложили оценивать структуру с помощью коэффициентов Fan_in(i) и Fan_out(i) модулей.

Большое значение Fan_in(i) — свидетельство высокого сцепления, так как является мерой зависимости модуля. Большое значение Fan_out(i) говорит о высокой сложности вызывающего модуля. Причиной является то, что для координации подчиненных модулей требуется сложная логика управления.

Основной недостаток коэффициентов Fan_in(i) и Fan_out(i) состоит в игнорировании веса связи. Здесь рассматриваются только управляющие потоки (вызовы модулей). В то же время информационные потоки, нагружающие ребра структуры, могут существенно изменяться, поэтому нужна мера, которая учитывает не только количество ребер, но и количество информации, проходящей через них.

С. Генри и Д. Кафура (1981) ввели информационные коэффициенты ifan_in(i) и ifan_out(j). Они учитывают количество элементов и структур данных, из которых i-й модуль берет информацию и которые обновляются j-м модулем соответственно.

Информационные коэффициенты суммируются со структурными коэффициентами sfan_in(i) и sfan_out(j), которые учитывают только вызовы модулей.

В результате формируются полные значения коэффициентов:

$$\text{Fan\_in}\ (i) = \text{sfan\_in}\ (i) + \text{ifan\_in}\ (i),$$

$$\text{Fan\_out}\ (j) = \text{sfan\_out}\ (j) + \text{ifan\_out}\ (j).$$

На основе полных коэффициентов модулей вычисляется метрика общей сложности структуры:

$$S = \sum_{i=1}^{n} \text{length}(i) * (\text{Fan\_in}(i) + \text{Fan\_out}(i))^2,$$

где length(i) — оценка размера i-го модуля (в виде LOC- или FP-оценки).