МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Лекция №3
Структурная парадигма проектирования информационных систем

Методы и средства проектирования информационно-аналитических систем
(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень: специалитет
(бакалавриат, магистратура, специалитет)

Форма обучения: очная
(очная, очно-заочная, заочная)

Направление(-я) подготовки: 10.05.04 «Информационно-аналитические системы безопасности»
(код(-ы) и наименование(-я))

Институт: Институт кибербезопасности и цифровых технологий (ИКБ)
(полное и краткое наименование)

Кафедра: Информационно-аналитические системы кибербезопасности (КБ-2)
(полное и краткое наименование кафедры, реализующей дисциплину (модуль))

Используются в данной редакции с учебного года 2023/24
(учебный год цифрами)

Проверено и согласовано «____» _____20___г.

_____ _____
(подпись директора Института/Филиала с расшифровкой)

Москва 2024 г.

Учебные вопросы:

1. Структурная парадигма проектирования информационных систем.
2. История структурного анализа и проектирования.
3. Метод структурного анализа и проектирования SADT.
4. Методология функционального моделирования IDEF0

1. Структурная парадигма проектирования информационных систем

В настоящее время известны две парадигмы проектирования, использующие два различных подхода к описанию систем:

– *структурная* (процессно-ориентированная), основанная на каскадной (водопадной) модели жизненного цикла ИС;

– *объектно-ориентированная*, основанная на итеративной модели жизненного цикла ИС.

Структурный анализ (Structured Analysis, SA) и **структурное проектирование** (Structured Design, SD) – результат появившегося в 1970-х структурного программирования, развивался из классического системного анализа. Методологии структурного анализа используют каскадную (водопадную) модель ЖЦ ИС; самые известные и используемые методологии структурно анализа – SADDT, SSADM. Методы структурного анализа дорабатывались и используются уже на протяжении многих лет.

Сравнительно позже появились и стали невероятно популярны объектно-ориентированные языки. По мере нарастания их популярности была разработана методология помощи программисту в разработке приложений с использованием объектно-ориентированных языков. Эта методология стала известна как объектно-ориентированный анализ и проектирование (object-oriented analysis and design, OOAD).

OOAD – это подход к инженерии ПО, моделирующий систему как группу взаимодействующих объектов. Объектно-ориентированный анализ (Object-oriented analysis, OOA) использует методы объектного моделирования для анализа функциональных требований к системе.

Основные преимущества структурных методологий:

- основаны на классическом системном анализе и процессно-ориентированы, подходят для описания любых (не только информационных) систем, идеальны для исследования предметной области, реинжиниринга бизнес процессов;
- понятное и относительно простое визуальное представление системы, легко понимаемое как пользователями, так и разработчиками;
- акцент на командной работе;
- четко определенные этапы, что облегчает управление проектом и позволяет разрабатывать системы лучшего качества;
- допускают использование средств проверки требований;
- SSAD и IDEF0 – классические, широко известные методологии в области проектирования ИС, существующие на протяжении длительного времени и достаточно «зрелые».

Недостатки структурных методологий:

- поскольку процессно-ориентированы, то соответственно, игнорируют нефункциональные требования: не идеальные решения при использовании объектно-ориентированных языков программирования, т.к. изначально создавались для структурных языков;
- поскольку SSAD и SSADM неитеративны, то изменение требований может привести к перезапуску всего процесса разработки;
- возможны трудности при определении глубины декомпозиции – момента, когда нужно остановиться и переходить к реализации модели.

Структурным анализом принято называть метод исследования системы, которое начинается с ее общего обзора и затем детализируется, приобретая иерархическую структуру с все большим числом уровней. Решение трудных проблем путем их разбиения на множество меньших независимых задач (так называемых «черных ящиков») и организация этих задач в древовидные иерархические структуры значительно повышают понимание сложных систем.

В инженерии ПО (software engineering), Структурный анализ (Structured Analysis, SA) и одноименное с ним Структурное проектирование (Structured Design, SD) – это методы для анализа и

преобразования бизнес-требований в спецификации и, в конечном счете, в компьютерные программы, конфигурации аппаратного обеспечения и связанные с ними ручные процедуры.

Структурный анализ, СА (Structured Analysis, SA) и Структурное проектирование, СП (Structured Design, SD) являются фундаментальными инструментами системного анализа и развивались из классического системного анализа 1960-70-х годов.

Структурный подход заключается в поэтапной декомпозиции системы при сохранении целостного о ней представления Основные принципы структурного подхода (первые два являются основными):

1) **«разделяй и властвуй»** – принцип решения сложных проблем путем их разбиения на множество меньших независимых задач, легких для понимания и решения;
2) **иерархического упорядочивания** – принцип организации составных частей проблемы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.
3) **абстрагирования** – заключается в выделении существенных аспектов системы и отвлечения от несущественных;
4) **формализации** – заключается в необходимости строгого методического подхода к решению проблемы;
5) **непротиворечивости** – заключается в обоснованности и согласованности элементов.

2. История структурного анализа и проектирования

Структурный анализ – это часть серии структурных методов, представляющих набор методологий анализа, проектирования и программирования, которые были разработаны в ответ на проблемы, с которыми столкнулся мир ПО в период с 1960 по 1980 гг. В этот период большинство программ было создано на Cobol и Fortran, потом на C и BASIC.

Это было новым положительным сдвигом в направлении проектирования и программирования, но при этом не было стандартных методологий для документирования требований и самих проектов. По мере укрупнения и усложнения систем, все более затруднительным становился процесс их разработки.

Когда большинство специалистов билось над созданием программного обеспечения, немногие старались разрешить более сложную задачу создания крупномасштабных систем, включающих как людей и машины, так и программное обеспечение, аналогичных системам, применяемым в телефонной связи, промышленности, управлении и контроле за вооружением. В то время специалисты, традиционно занимавшиеся созданием крупномасштабных систем, стали осознавать необходимость большей упорядоченности. Таким образом, разработчики начали формализовать процесс создания системы, разбивая его на следующие фазы:

1) анализ – определение того, что система будет делать;
2) проектирование – определение подсистем и их взаимодействие;
3) реализация – разработка подсистем по отдельности;
4) объединение – соединение подсистем в единое целое;
5) тестирование – проверка работы системы;
6) установка – введение системы в действие;
7) функционирование – использование системы.

Эта последовательность всегда выполнялась итерационно, потому что система полностью никогда не удовлетворяла требованиям пользователей, поскольку их требования часто менялись. И, тем не менее, с этой моделью создания системы, ориентированной на управление, постоянно возникали сложности. Эксплуатационные расходы, возникавшие после сдачи системы, стали существенно превышать расходы на ее создание и продолжали расти с огромной скоростью из-за низкого качества исходно созданной системы. Некоторые считали, что рост эксплуатационных расходов обусловлен характером ошибок, допущенных в процессе создания системы. Исследования показали, что большой процент ошибок в системе возник в процессе анализа и проектирования, гораздо меньше их было допущено при реализации и тестировании, а цена (временная и денежная) обнаружения и исправления ошибок становилась выше на более поздних стадиях проекта. Например, исправление ошибки на стадии проектирования стоит в 2 раза, на стадии тестирования – в 10 раз, а на стадии эксплуатации системы – в 100 раз дороже, чем на стадии анализа. На обнаружение ошибок, допущенных на этапе анализа и проектирования, расходуется примерно в 2 раза больше времени, а на их исправление – примерно в 5 раз, чем на ошибки, допущенные на более

поздних стадиях. Кроме того, ошибки анализа и проектирования обнаруживались часто самими пользователями, что вызывало их недовольство.

Традиционные подходы к созданию систем приводили к возникновению многих проблем:

1) не было единого подхода;
2) привлечение пользователя к процессу разработки не контролировалось;
3) проверка на согласованность проводилась нерегулярно или вообще отсутствовала, результаты одного этапа не согласовывались с результатами других;
4) процесс с трудом поддавался оценкам, как качественным, так и количественным.

Утверждалось, что, когда создатели систем пользуются методологиями типа структурного программирования и проектирования сверху вниз, они решают либо не поставленные задачи, либо плохо поставленные, либо хорошо поставленные, но неправильно понятые задачи. Кроме того, ошибки в создании систем становились все менее доступны выявлению с помощью аппаратных средств или программного обеспечения, а наиболее катастрофические ошибки допускались на ранних этапах создания системы. Часто эти ошибки были следствием неполноты функциональных спецификаций или несогласованности между спецификациями и результатами проектирования. Проектировщики знали, что сложность систем возрастает и что определены они часто весьма слабо. Рост объема и сложности систем является жизненной реалией. Эту предпосылку нужно было принять как неизбежную. Но ошибочное определение системы не является неизбежным: оно – результат неадекватности методов создания систем.

Вскоре был выдвинут тезис: совершенствование методов анализа есть ключ к созданию систем, эффективных по стоимости, производительности и надежности. Для решения ключевых проблем традиционного создания систем широкого профиля требовались новые методы, специально предназначенные для использования на ранних стадиях процесса.

Для помощи в управлении большим и сложным ПО с конца 1960 годов появляется множество разнообразных структурных методов программирования, проектирования и анализа. Эти методы на начальных этапах создания системы позволяют гораздо лучше понять рассматриваемую проблему. А это сокращает затраты как на создание, так и на эксплуатацию системы, а кроме того, повышает ее надежность.

В 1960-70 появляются следующие концепции:

- примерно 1967 – Структурное программирование (Structured programming) – Edsger Dijkstra,
- примерно 1975 – Структурное программирование Джексона (Jackson Structured Programming) – Michael A. Jackson.

Структурное программирование приводит к Структурному проектированию, что в свою очередь приводит к Структурному системному анализу:

- примерно 1975 – появление на рынке Метода структурного анализа и проектирования SADT (Structured Analysis and Design Technique) – Douglas T. Ross;
- примерно 1975 – Структурное проектирование (Structured Design) – Larry Constantine, Ed. Yourdon и Wayne Stevens;
- примерно 1978 – Структурный анализ (Structured Analysis) – Tom De-Marco, Yourdon, Gane & Sarson, McMenamin & Palmer;
- в 1979 опубликован Структурный анализ и системная спецификация (Structured Analysis and System Specification) – Tom DeMarco.

В течение 1980х начинают появляться инструменты для автоматизации черчения диаграмм:

- 1981 – опубликована (и в 85-93 получает развитие) Методология IDEF0, основанная на SADT и инструментальных средствах создания диаграмм (разработана Дугласом Т. Россом, Douglas T. Ross).
- в 1983 впервые представлен Метод структурного системного анализа и проектирования SSADM (Structured Systems Analysis and Design Method), разработанный в UK Office of Government Commerce.

По аналогии с Computer-Aided design and Computer-Aided Manufacturing (CAD/CAM), использование этих инструментов было названо **Computer-Aided Software Engineering (CASE)**.

Методы СА и СП (в частности, SSADM) сопровождаются нотациями (диаграммами), облегчающими взаимодействие между пользователями и разработчиками. Это были:

- структурные схемы (Structure Charts) – для структурного проектирования;
- диаграммы потоков данных (Data Flow Diagrams, DFD) – для структурного анализа;
- модели данных (Data Model Diagrams).

Эти диаграммы использовались структурными методами в различных комбинациях. Среди них встречалось множество вариаций.

Примерно в 1990 появляется термин «инженерия разработки ПО» (Infor-mation Engineering, IE, James Martin), являющаяся логическим расширением структурных методов, появившихся в течение 1970х.

3. Метод структурного анализа и проектирования SADT

SADT (Structured Analysis and Design Technique) – это методология инженерии разработки ПО (software engineering) для описания систем в виде иерархии функций (функциональной структуры).

Структурный анализ возник в конце 60-х годов в ходе революции, вызванной структурным программированием. Метод SADT был предложен Дугласом Т. Россом как способ уменьшить количество дорогостоящих ошибок за счет структуризации на ранних этапах создания системы, улучшения контактов между пользователями и разработчиками и сглаживания перехода от анализа к проектированию. Дуглас Т. Росс часть своих PLEX-теорий относящихся к методологии и языку описания систем, назвал «Методология структурного анализа и проектирования» (SADT). Исходная работа над SADT началась в 1969 г.

Появление SADT на рынке произошло в 1975 г. после годичного оформления в виде продукта.

SADT широко распространение в настоящее время в европейской, дальневосточной и американской аэрокосмической промышленности (под названием IDEF0) позволяет эти цифры существенно увеличить. Таким образом, SADT выделяется среди современных методологий описания систем благодаря своему широкому применению. Почему SADT имеет такое широкое применение?

Во-первых, SADT является единственной методологией, легко отражающей такие системные характеристики, как управление, обратная связь и исполнители. Это объясняется тем, что SADT изначально возникла на базе проектирования систем более общего вида в отличие от других структурных методов, «выросших» из проектирования программного обеспечения.

Во-вторых, SADT в дополнение к существовавшим в то время концепциям и стандартам для создания систем имела развитые процедуры поддержки коллективной работы и обладала преимуществом, связанным с ее применением на ранних стадиях создания системы.

Кроме того, широкое использование SADT показало, что ее можно сочетать с другими структурными методами. Это достигается использованием графических SADT-описаний в качестве схем, связывающих воедино различные методы, примененные для описания определенных частей системы с различным уровнем детализации.

Основы SADT

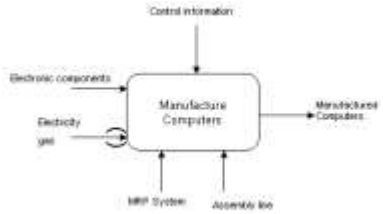
SADT использует два типа диаграмм: 1) модели деятельности (activity models); 2) модели данных (data models).

SADT использует стрелки для построения этих диаграмм и имеет следующее графическое представление:

- главный блок (box), где определено название процесса или действия;
- с левой стороны блока – входящие стрелки: входы действия;
- сверху – входящие стрелки: данные, необходимые для действия;
- внизу – входящие стрелки: средства, используемые для действия;
- справа – исходящие стрелки: выход действия.

SADT использует декомпозицию на основе подхода «сверху вниз». Каждый уровень декомпозиции содержит до 6 блоков.

SADT начинается с уровня (level) 0, затем может быть детализирован на более низкие уровни (1, 2, 3, ...). Например, на уровне 1, блок уровня 0 будет детализирован на несколько элементарных блоков и так далее …



На уровне 1 действие «Manufacture computers», может быть разбито (declined), например на 4 блока:

1) получить электронные компоненты («receive electronic components»);

2) сохранить электронные компоненты («store electronic components»);
3) доставить электронные компоненты на сборочную линию («bring electron-ic components to the assembly line»);
4) собрать компьютеры («Assemble computers»).

Семантика стрелок для действий (activities):

- входы (Inputs) входят слева и представляют данные или предметы потребления (consumables), нужные действию (that are needed by the activity);
- выходы (Outputs) выходят справа и представляют данные или продукты, производимые действием (activity);
- управления (Controls) входят сверху и представляют команды, которые влияют на исполнение действия, но не потребляются. В последней редакции IDEF0 – условия, требуемые для получения корректного выхода. Данные или объекты, моделируемые как управления, могут быть трансформированы функцией, создающей выход;
- механизмы означают средства, компоненты или инструменты, используемые для выполнения действия представляют размещение (allocation) действий.

Семантика стрелок для данных (data):

- входы (Inputs) – это действия, которые генерируют эти данные (are activities that produce the data);
- выходы (Outputs) потребляют эти данные (consume the data);
- управления (Controls) влияют на внутреннее состояние этих данных (influence the internal state of the data).

Роли SADT-процесса:

- авторы (Authors) – разработчики SADT модели;
- комментаторы (Commenters) – рецензируют (review) работу авторов;
- читатели (Readers) – возможные (the eventual) пользователи SADT диаграмм;
- эксперты (Experts) – те, от кого авторы получают специальную информацию о требованиях и ограничениях;
- технический комитет (Technical committee) – технический персонал, ответственный за рецензирование (reviewing) SADT модели на каждом уровне;
- библиотекарь проекта (Project librarian) – ответственный за все документы проекта;
- менеджер проекта (Project manager) – имеет полную техническую ответственность за системный анализ и проектирование (has overall technical responsibility the system analysis and design);
- аналитик (Monitor) (Chief analyst) – эксперт в области SADT, помогающий и консультирующий персонал проекта по использованию SADT;
- инструктор (Instructor) – обучает авторов и комментаторов SADT.

Этапы моделирования. Разработка SADT модели представляет собой итеративный процесс и состоит из нижеследующих условных этапов.

1 Создание модели группой специалистов, относящихся к различным сферам деятельности предприятия. На этом этапе авторы опрашивают компетентных лиц, получая ответы на следующие вопросы:

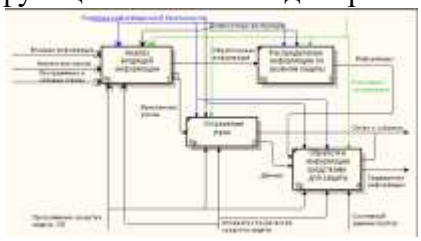
- что поступает в предметную область на «входе»;
- какие функции и в какой последовательности выполняются в рамках предметной области;
- кто является ответственным за выполнение каждой из функций;
- чем руководствуется исполнитель при выполнении каждой из функций;
- что является результатом работы объекта (на выходе)?

На основе полученных результатов опросов создается черновик модели (Model Draft).

2 Распространение черновика для рассмотрения, получения комментариев и согласования модели с читателями. При этом каждая из диаграмм черновика письменно критикуется и комментируется, а затем передается автору. Автор, в свою очередь, также письменно соглашается с критикой или отвергает ее с изложением логики принятия решения и вновь возвращает откорректированный черновик для дальнейшего рассмотрения. Этот цикл продолжается до тех пор, пока авторы и читатели не придут к единому мнению.

3 Официальное утверждение модели. Утверждение согласованной модели происходит руководителем рабочей группы в том случае, если у авторов модели и читателей отсутствуют разногласия по поводу ее адекватности. Окончательная модель представляет собой согласованное представление о системе с заданной точки зрения и для заданной цели.

Метод SADT получил дальнейшее развитие. На его основе в 1981 году разработана известная методология функционального моделирования **IDEF0**.



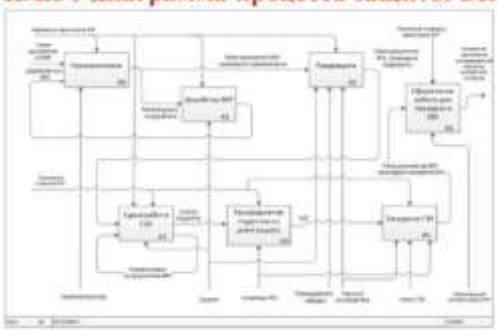
4. Методология функционального моделирования IDEF0

Основные понятия IDEF0

IDEF0 (Integration Definition for Function Modeling) – методология функционального моделирования для описания функций предприятия, предлагающая язык функционального моделирования для анализа, разработки, реинжиниринга и интеграции информационных систем бизнес процессов или анализа инженерии разработки ПО (or software engineering analysis).

Модель IDEF0 – это графическое описание (информационной) системы или предметной области (subject), которое разрабатывается с определенной целью с выбранной точки зрения. Модель IDEF0 представляет собой набор из одной или более (иерархически связанных) IDEF0-диаграмм, которые описывают функции системы или предметной области (subject area) с помощью графики, текста и глоссария.

IDEF0-диаграмма процесса защиты ВКР



Применение IDEF0

IDEF0 используется для создания функциональной модели, то есть результатом применения методологии IDEF0 к системе есть функциональная модель IDEF0. Функциональная модель – это структурное представление функций, деятельности или процессов в пределах моделируемой системы или предметной области.

Методология IDEF0 может быть использована для моделирования широкого спектра как автоматизированных, так и неавтоматизированных систем.

Для проектируемых систем IDEF0 может быть использована сначала для определения требований и функций, и затем для реализации, удовлетворяющей этим требованиям и исполняющей эти функции.

Для существующих систем IDEF0 может быть использована для анализа функций, выполняемых системой, а также для учета механизмов, с помощью которых эти функции выполняются.

Цели стандарта IDEF0

Основные цели (objectives) стандарта:

1) задокументировать и разъяснить технику моделирования IDEF0 и правила ее использования;
2) обеспечить средства для полного и единообразного (consistently) моделирования функций системы или предметной области, а также данных и объектов, которые связывают эти функции;
3) обеспечить язык моделирования, который независим от CASE методов или средств, но может быть использован при помощи этих методов и средств;
4) обеспечить язык моделирования, который имеет следующие характеристики:

- *общий* (generic) – для анализа как (информационных) систем, так и предметных областей;
- *строгий и точный* (rigorous and precise) – для создания корректных, пригодных к использованию моделей;

- *краткий* (concise) – для облегчения понимания, коммуникации, согласия между заинтересованными лицами и проверки (to facilitate understanding, communication, consensus and validation);
- *абстрактный* (conceptual) – для представления функциональных требований, независимых от физических или организационных реализаций;
- *гибкий* – для поддержки различных фаз жизненного цикла проекта.

Строгость и точность (Rigor and Precision). Правила IDEF0 требуют достаточной *строгости и точности* для удовлетворения нужд аналитика без чрезмерных ограничений (to satisfy needs without overly constraining the analyst).

IDEF0 правила включают следующее:

- **управление детализацией** (control of the details communicated at each level) – от трех до шести функциональных блоков на каждом уровне декомпозиции;
- **связанный контекст** (Bounded Context) – не должно быть недостающих или лишних, выходящих за установленные рамки деталей;
- **связанность интерфейса диаграмм** (Diagram Interface Connectivity) – наличие номеров узлов, функциональных блоков, С-номеров (C-numbers) и подробных ссылочных выражений (Detail Reference Expression);
- **связанность структуры данных** (Data Structure Connectivity) – коды ICOM (Input, Control, Output и Mechanism) и использование круглых скобок (ICOM codes and the use of parentheses);
- **уникальные метки и заголовки** (Unique Labels and Titles) – отсутствие повторяющихся названий в метках и заголовках;
- **синтаксические правила для графики** (Syntax Rules for Graphics) – функциональные блоки и стрелки;
- **ограничения на разветвления стрелок данных** (Data Arrow Branch Constraint) – метки для ограничений потоков данных на разветвлениях;
- **разделение данных на Вход и Управление** (Input versus Control Separation) – правило для определения роли данных);
- **маркировка стрелок данных** Data Arrow Label Requirements (minimum labeling rules);
- наличие **Управления** (Minimum Control of Function) – все функции должны иметь минимум одно Управление;
- **цель и точка зрения** (Purpose and Viewpoint) – все модели имеют формулировку цели и точки зрения.

Ограничения сложности. Обычно IDEF0-модели несут в себе сложную и концентрированную информацию, и для того, чтобы ограничить их перегруженность и сделать удобочитаемыми, в стандарте приняты соответствующие ограничения сложности, которые носят рекомендательный характер. При том, что, что на диаграмме рекомендуется представлять **от трех до шести** функциональных блоков, количество подходящих к одному функциональному блоку (выходящих из одного функционального блока) интерфейсных дуг предполагается **не более четырех**.

Основные понятия IDEF0

В основе методологии лежат четыре основных понятия:

- функциональный блок;
- интерфейсная дуга;
- декомпозиция;
- глоссарий.

Функциональный блок (Activity Box) представляет собой некоторую конкретную функцию в рамках рассматриваемой системы.

По требованиям стандарта название каждого функционального блока должно быть сформулировано **в глагольном наклонении** (например, «производить услуги»).

На диаграмме функциональный блок изображается прямоугольником. Каждая из четырех сторон функционального блока имеет свое определенное значение (роль), при этом:

- верхняя сторона имеет значение «Управление» (Control);
- левая сторона имеет значение «Вход» (Input);

- правая сторона имеет значение «Выход» (Output);
- нижняя сторона имеет значение «Механизм» (Mechanism).

Рис. Функциональный блок

Интерфейсная дуга/стрелка (Arrow) отображает элемент системы, который обрабатывается функциональным блоком или оказывает иное влияние на функцию, представленную данным функциональным блоком. Интерфейсные дуги часто называют потоками или стрелками.

С помощью интерфейсных дуг отображают различные объекты, в той или иной степени определяющие процессы, происходящие в системе. Такими объектами могут быть элементы реального мира (детали, вагоны, сотрудники и т.д.) или потоки данных и информации (документы, данные, инструкции и т.д.).

В зависимости от того, к какой из сторон функционального блока подходит данная интерфейсная дуга, она носит название «**входящей**», «**исходящей**» или «**управляющей**».

Необходимо отметить, что любой функциональный блок по требованиям стандарта должен иметь, по крайней мере, **одну управляющую интерфейсную дугу и одну исходящую**. Каждый процесс должен происходить по каким-то правилам (отображаемым управляющей дугой) и должен выдавать некоторый результат (выходящая дуга), иначе его рассмотрение не имеет никакого смысла.

Обязательное наличие *управляющих интерфейсных дуг* является одним из главных отличий стандарта IDEF0 от других методологий классов DFD (Data Flow Diagram) и WFD (Work Flow Diagram).

Механизмы показывают средства, с помощью которых осуществляется выполнение функций. Механизм может быть человеком, компьютером или любым другим устройством, которое помогает выполнять данную функцию (рис.).

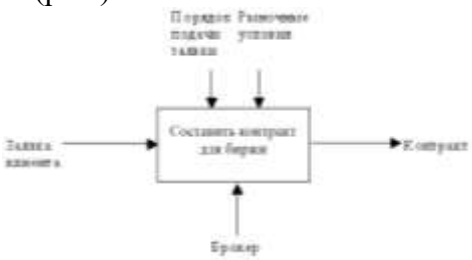


Рис. Пример механизма

По типу связываемых элементов интерфейсные дуги (стрелки) подразделяют на *граничные, стрелки вызова, внутренние стрелки*.

Граничные (border) **стрелки** на *контекстной диаграмме* применяются для описания взаимодействия системы с окружающим миром. Они могут начинаться у границы диаграммы и заканчиваться у функции, или наоборот. На обычной (не контекстной) диаграмме граничные стрелки представляют входы, управления, выходы или механизмы родительского блока диаграммы. Внесенные граничные стрелки как бы указывают на выход за пределы страницы на диаграмме декомпозиции нижнего уровня, и для указания их отличия от других стрелок диаграммы, стрелка таких стрелок изображается в квадратных скобках. Граничные стрелки обеспечивают правильное соединение диаграмм для получения согласованной модели состоящей из отдельных диаграмм. Т.о. эти стрелки являются интерфейсом между диаграммой и остальной частью модели, поэтому необходимо, чтобы все внешние стрелки диаграммы были согласованы со стрелками, образующими границу этой диаграммы. Такое согласование стрелок обеспечивает совмещение данной диаграммы со своей родительской диаграммой, т.е. это означает, что внешние стрелки согласованы по числу и наименованию (но не обязательно по расположению) со стрелками, касающимися декомпозированного блока родительской диаграммы. Источник или приемник граничных стрелок можно обнаружить, только изучая родительскую диаграмму. Все граничные стрелки на дочерней диаграмме (за исключением стрелок, помещенных в туннель) должны соответствовать стрелкам родительского блока.

Стрелки вызова (Call) –- специальная стрелка, указывающая на другую модель работы. **Стрелка вызова** используется для указания того, что некоторая функция выполняется за пределами моделируемой системы, т.е. обозначают обращение из данной модели или из данной части модели к блоку, входящему в состав другой модели или другой части модели, обеспечивая их связь. **Наличие стрелок вызова** указывает на то, что разные модели или разные части одной и той же модели могут совместно использовать один и тот же элемент (блок). Стрелки вызова используются при слиянии и разделении моделей.

Внутренние стрелки – это стрелки IDEF0-диаграммы, характеризующие четыре основных отношения, концы которой связывают источник и потребителя, являющиеся блоками одной диаграммы. Внутренние стрелки не касаются границы диаграммы и не выходят за ее пределы, а начинаются у одного и кончаются у другого блока.

В IDEFO различают пять типов связей для внутренних стрелок по их **направленности**:

- **связь «выход-вход»** (output-input) является простейшей связью, поскольку она отражает прямые воздействия, которые интуитивно понятны и очень просты. Связь возникает тогда, когда выход одного блока становится входом для другого. Стрелка выхода вышестоящей работы (далее – просто выход) направляется на вход нижестоящей (блока с меньшим доминированием). Связь по «выход-вход» показывает доминирование вышестоящей работы, т.е. выход блока становится входом для блока с меньшим доминированием. Данные или объекты выхода вышестоящей работы не меняются в нижестоящей;
- **связь «выход-управление»** (output-control), является простейшей связью, поскольку она отражает прямые воздействия, которые интуитивно понятны и очень просты. При такой связи выход вышестоящей работы направляется непосредственно на управление нижестоящей, таким образом показывая доминирование вышестоящей функции. Данные или объекты выхода вышестоящей функции не меняются в нижестоящей;
- **обратная связь «выход-вход»** (output-input feedback) является более сложной, поскольку представляют **итерацию** или **рекурсию** (выходы из одной функции влияют на будущее выполнение других функций, что впоследствии влияет на исходную функцию). Такая связь, как правило, используется для описания циклов и часто называется связью по потоку данных. В такой связи выход нижестоящей работы направляется на вход вышестоящей (блока с большим доминированием). Обратные связи могут выступать в виде комментариев, замечаний, исправлений и т.д. (рис.).

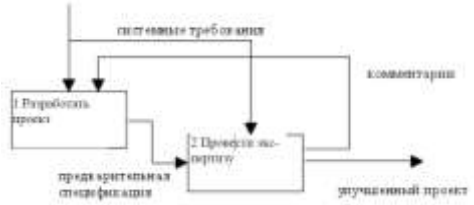


Рис. Пример обратной связи

- **обратная связь «выход-управление»** (output-control feedback) является более сложной, поскольку представляют итерацию или рекурсию (выходы из одной функции влияют на будущее выполнение других функций, что впоследствии влияет на исходную функцию). Обратная связь возникает, когда выход некоторой функции А воздействует на выход функции В, а выход функции В воздействует на другую активацию функции А. Обратная связь по управлению возникает тогда, когда выход некоторого блока (как результат деятельности некоторой функции) влияет на блок с большим доминированием, т.е. выход нижестоящей работы направляется на управление вышестоящей. Обратная связь по управлению часто свидетельствует об эффективности процесса;
- **связь «выход-механизм»** (output-mechanism), когда выход одной работы направляется на механизм другой. Эта взаимосвязь встречаются нечасто, и используется реже остальных. Она показывает, что одна работа подготавливает ресурсы, необходимые для проведения другой работы – связь отражает ситуацию, при которой выход одной функции становится средством достижения цели (подготавливает ресурсы), необходимые для проведения другой функции для другой. Обычно связи «выход-механизм» характерны при распределении источников ресурсов (например, требуемые

инструменты, обученный персонал, физическое пространство, оборудование, финансирование, материалы).

Одним из важных моментов при проектировании ИС с помощью методологии IDEF0 является точная согласованность типов внутренних связей по их **характеру** (табл. 1.).

Таблица 1. Типы и связей и их значимость

Тип связи	Относительная значимость
Случайная	0
Логическая	1
Временная	2
Процедурная	3
Коммуникационная	4
Последовательная	5
Функциональная	6

Краткие описания каждого из типов связей.

(0) Тип случайной связности: наименее желательный. Случайная связность возникает, когда конкретная связь между функциями мала или полностью отсутствует. Это относится к ситуации, когда имена данных на IDEF0-дугах в одной диаграмме имеют малую связь друг с другом. Крайний вариант этого случая показан на рис. 4.5.

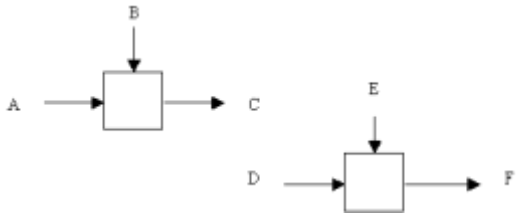


Рис. Случайная связность

(1) Тип логической связности. Логическое связывание происходит тогда, когда данные и функции собираются вместе вследствие того, что они попадают в общий класс или набор элементов, но необходимых функциональных отношений между ними не обнаруживается.

(2) Тип временной связности. Связанные по времени элементы возникают вследствие того, что они представляют функции, связанные во времени, когда данные используются одновременно или функции включаются параллельно, а не последовательно.

(3) Тип процедурной связности. Процедурно-связанные элементы появляются сгруппированными вместе вследствие того, что они выполняются в течение одной и той же части цикла или процесса.

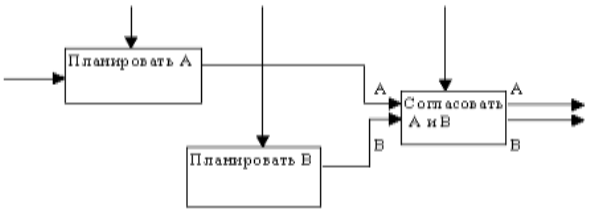


Рис. Процедурная связность

(4) Тип коммуникационной связности. Диаграммы демонстрируют коммуникационные связи, когда блоки группируются вследствие того, что они используют одни и те же входные данные и/или производят одни и те же выходные данные.

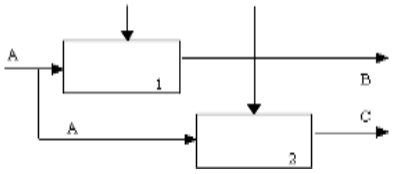


Рис. Коммуникационная связность

(5) Тип последовательной связности. На диаграммах, имеющих последовательные связи, выход одной функции служит входными данными для следующей функции. Связь между элементами на

диаграмме является более тесной, чем на рассмотренных выше уровнях связок, поскольку моделируются причинно-следственные зависимости.

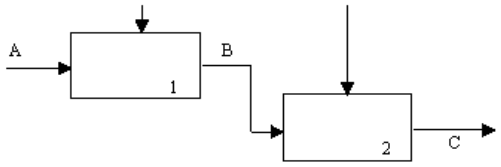


Рис. Последовательная связность

(6) Тип функциональной связности. Диаграмма отражает полную функциональную связность, при наличии полной зависимости одной функции от другой. Диаграмма, которая является чисто функциональной, не содержит чужеродных элементов, относящихся к последовательному или более слабому типу связности. Одним из способов определения функционально-связанных диаграмм является рассмотрение двух блоков, связанных через управляющие дуги.

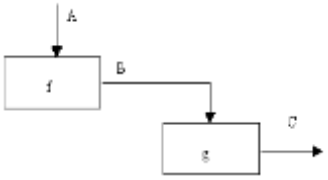


Рис.. Функциональная связность

В таблице представлены все типы связей, рассмотренные выше. Важно отметить, что **уровни 4 – 6** устанавливают типы связностей, которые разработчики считают важнейшими для получения диаграмм хорошего качества.

Таблица 2. Типы связностей

Значимость	Тип связности	Для функций	Для данных
0	Случайная	Случайная	Случайная
1	Логическая	Функции одного и того же множества или типа (например, «редактировать все входы»)	Данные одного и того же множества или типа
2	Временная	Функции одного и того же периода времени (например, «операции инициализации»)	Данные, используемые в каком-либо временном интервале
3	Процедурная	Функции, работающие в одной и той же фазе или итерации (например, «первый проход компилятора»)	Данные, используемые во время одной и той же фазы или итерации
4	Коммуникационнная	Функции, использующие одни и те же данные	Данные, на которые воздействует одна и та же деятельность
5	Последовательная	Функции, выполняющие последовательные преобразования одних и тех же данных	Данные, преобразуемые последовательными функциями
6	Функциональная	Функции, объединяемые для выполнения одной функции	Данные, связанные с одной функцией

Ветвление и слияние стрелок. IDEF является мощным языком описания систем во многом благодаря возможности декомпозиции стрелок. Стрелка в IDEF0 обычно изображают набор объектов (данных), поэтому они могут иметь множество начальных точек (источников) и конечных точек (назначений), поэтому стрелки могут разветвляться и соединяться разными сложными способами. Вся стрелка или ее часть может выходить из одного или нескольких блоков и заканчиваться в одном или нескольких блоках. Таким образом, одни и те же данные, порожденные одной функцией, могут использоваться или в одной, или в нескольких других функциях одновременно. С другой стороны, данные, порожденные в различных функциях, могут представлять собой одинаковые или однородные данные/объекты, которые в дальнейшем используются или обрабатываются в одном месте.

Для моделирования таких ситуаций в IDEF0 используются разветвляющиеся и соединяющиеся стрелки, которые отражают иерархию объектов, представленных этими стрелками. Разветвления стрелок и их слияние, тот самый синтаксис, который позволяет описывать декомпозицию содержимого стрелок и

уменьшает загруженность диаграмм графическими элементами (линиями). Однако стрелки описывают только ту иерархию, которая связывает отдельные функции системы, представленные на диаграммах.

На самом деле, из отдельной диаграммы редко можно понять полную иерархию стрелки. Обычно это требует чтения большей части модели, а иногда из-за выбранной точки зрения подробности отдельных стрелок не раскрываются совсем. Вот почему IDEF предусматривает дополнительное описание полной иерархии объектов системы посредством формирования **глоссария** для каждой диаграммы модели и объединения этих глоссариев в **словарь стрелок**.

Таким образом, словарь стрелок, важное дополнение модели, становится основным хранилищем полной иерархии объектов системы.

Чтобы стрелки и их сегменты правильно описывали связи между блоками-источниками и блоками-приемниками, используются ***метки*** для каждой ветви стрелок.

Для описания представления разветвлений и соединений стрелок разработаны специальные *соглашения* относительно представления, описания и правила маркирования разветвлений и соединений таких стрелок.

Ветвление стрелки – это разделение ее на два или большее число сегментов, изображенных в виде расходящихся линий, и означающее, что все содержимое стрелок или его часть может появиться в каждом ответвлении стрелки. Стрелка всегда помечается до разветвления, чтобы дать название всему набору. Кроме того, каждая ветвь стрелки может быть помечена или не помечена в соответствии со следующими *правилами*:

- если стрелка именована до разветвления, а после разветвления ни одна из ветвей не именована, то подразумевается, что каждая ветвь моделирует те же данные или объекты, что и ветвь до разветвления, т.е. все объекты, указанные в метке стрелки перед ветвлением принадлежат этим ветвям (каждому из сегментов);
- если стрелка именована до разветвления, а после разветвления только некоторые из ветвей именованы, это означает, что маркированные ветви соответствуют своему именованию (каждая метка ветви уточняет, что именно содержит ветвь), а неименованные ветви моделируют те же данные или объекты, что и ветвь до разветвления;
- недопустима ситуация, когда стрелка до разветвления не именована, а после разветвления не именована какая-либо из ветвей.

Иногда функция разделяет стрелку на ее **компоненты** в этом случае для получения дополнительных сведений о содержании компонент и взаимосвязях между ними важно изучить, что выполняет эта функция.

Слияние стрелок – это объединение двух или большего числа стрелок, изображенных в виде сходящихся вместе линий и означающее, что содержимое каждой ветви идет на формирование метки для стрелки, являющейся результатом слияния исходных.

Правила именования сливающихся стрелок полностью аналогичны правилам ветвления.

Соглашения по слиянию и разъединению стрелок:

- объединение стрелок возможно только в случае, если их источники не указаны на диаграмме или они представляют одни и те же данные. Такое вычерчивание позволяет абсолютно точно указать единый источник сходных данных;
- объединять стрелки с общим источником или с общим приемником можно только в том случае, если они представляют связанные данные. В этом случае общее название лучше описывает суть данных;
- стрелки связываются (сливаются), если они представляют сходные данные и их источник не указан на диаграмме.

После слияния *результирующая стрелка* всегда помечается для указания нового набора объектов, возникшего после объединения. Кроме того, *каждая ветвь* перед слиянием может помечаться или не помечаться в соответствии со следующими правилами:

- непомеченные ветви содержат все объекты, указанные в общей метке стрелки после слияния;
- если результирующая ветка помечена, то это означает, что ветви, помеченные перед слиянием, содержат все или некоторые объекты, перечисленные в общей метке после слияния Если, у

результирующей ветки метка отсутствует, это означает, что общая ветка передает все объекты, принадлежащие сливаемым веткам;

- ошибкой является наличие на диаграмме стрелки, которая после слияния не именована, а до слияния не именована какая-либо из ее ветвей.

В методологии IDEF0 существует соглашения по размещению стрелок:

- вычерчивание стрелок, независимо от их назначения, выполняется только **по вертикали** и **горизонтали**, что позволяет проследить за направлением стрелок и определить блоки как точки сбора стрелок, которыми блоки и являются;
- следует обеспечить **максимальное расстояние** между блоками и поворотами стрелок, а также между блоками и пересечениями стрелок для облегчения чтения диаграммы. Одновременно уменьшается вероятность перепутать две разные стрелки;
- следует максимально увеличивать расстояние между входящими или выходящими стрелками на одной грани блока;
- следует максимально увеличить расстояние между поворотами и пересечениями стрелок;
- **наличие входных стрелок** у блока не является его обязательным атрибутом;
- если две стрелки проходят параллельно (начинаются из одной и той же грани одной функции и заканчиваются на одной и той же грани другой функции), то по возможности следует их объединить и назвать единым термином;
- **любой блок** обязательно должен иметь стрелки управления, наличие которых гарантирует работоспособность функции и обеспечивает наложение ограничений и включение/выключение функций системы;
- если данные служат и для управления, и для входа, то вычерчиваются только стрелки управления, что уменьшает сложность диаграммы и делает очевидным управляющий характер данных;
- расстояние между **параллельными стрелками** должно быть максимально возможным (с учетом габаритов диаграммы), что позволяет оставить больше места для меток и помогает зрительно определять количество стрелок и прослеживать их пути;
- расстояние между блоками и поворотами стрелок должно быть максимально возможным (с учетом габаритов диаграммы), что позволяет облегчить процесс чтения и уменьшить вероятность перепутать две разные стрелки;
- **циклические обратные связи** для одного и того же блока выполняются только для того, чтобы более четко подчеркнуть значение повторно используемого объекта. Обычно обратную связь изображают на диаграмме, декомпозирующей блок. Однако иногда требуется выделить повторно используемые объекты;
- **обратные связи «выход-управление»** вычерчиваются «вверх и над» («верхняя» петля), что позволяет указать ограничивающие обратные связи при минимальном числе линий и пересечений, а также собрать все стрелки управления в правой верхней части диаграммы;
- **обратные связи «выход-вход»** вычерчиваются «вниз и под» («нижняя» петля), что позволяет указать обратные потоки данных при минимальном числе линий и пересечений, а также собрать все входные стрелки в левой нижней части диаграммы;
- **обратные связи «выход-механизм»** должны быть показаны как «вниз и под»;
- **количество необязательных пересечений** стрелок при соединении большого числа блоков, число петель и поворотов каждой стрелки, должны быть минимальным, что позволяет значительно уменьшить сложность диаграммы. Это ручная и, в случае насыщенных диаграмм, творческая функция;
- иногда разрешается **выделять буферы** и повторно используемые объекты;
- **слияние стрелок** производится, если они имеют общий источник или приемник, или если они представляют связанные данные. В таком случае общее название лучше описывает суть данных;

- следует **минимизировать** число стрелок, касающихся каждой стороны блока, если, конечно, природа данных не слишком разнородна;
- если возможно, стрелки присоединяются к блокам в одной и той же ICOM-позиции. Тогда соединение стрелок конкретного типа с блоками будет согласованным, и чтение диаграммы упростится;
- при соединении большого числа блоков необходимо избегать необязательных пересечений стрелок. Следует минимизировать число петель и поворотов каждой стрелки;
- блоки (функции) являются **сопряженными через среду**, если они имеют связи с источником, генерирующим данные, без конкретного определения отношения отдельной части данных к какому-либо блоку;
- две или более функций являются **сопряженными через запись**, если они связаны с набором данных и не обязательно зависят от того, представлены ли все возможные интерфейсы как сопряжение через среду. Тип интерфейса, показанный на последнем рисунке, предпочтителен, поскольку определяются отношения конкретных элементов данных к каждому блоку;
- необходимо использовать (где это целесообразно) выразительные возможности **ветвящихся** стрелок;
- при наличии стрелок со сложной топологией целесообразно повторить метку для удобства ее идентификации.