



## 1. Определение, виды и компоненты ЭИС.

**Ответ:** Экономическая информационная система (ЭИС) – это совокупность взаимосвязанных компонентов (организационных, технических, программных, информационных и др.), предназначенная для хранения, обработки и выдачи информации, необходимой для управления объектом <sup>1</sup> <sup>2</sup>. Выделяют виды ЭИС по масштабу (федеральные, корпоративные, отделов, АРМ и т.д.) <sup>3</sup> и по функциям (учет, анализ, контроль, планирование и др.) <sup>4</sup>. К основным компонентам ЭИС относятся: **база данных** (единое хранилище данных), **концептуальная схема** (описание структуры данных) и **информационный процессор** (вычислительная система + СУБД, выполняющие операции над данными) <sup>5</sup> <sup>6</sup>. Эти компоненты обеспечивают централизованное хранение, актуальность и совместное использование данных в системе.

## 2. Понятие и состав обеспечивающих подсистем ЭИС.

**Ответ:** Обеспекивающие подсистемы ЭИС – это средства и ресурсы, с помощью которых функционируют все функциональные подсистемы ЭИС <sup>7</sup>. Их состав включает: **информационное обеспечение** (система классификации и кодирования, базы данных и знания) <sup>8</sup>, **техническое обеспечение** (компьютеры, сети, оргтехника, обслуживающий персонал) <sup>8</sup>, **программное обеспечение** (системные и прикладные программы для работы ЭИС) <sup>9</sup>, **математическое обеспечение** (экономико-математические методы и модели, используемые при разработке и работе ПО) <sup>10</sup>, **лингвистическое обеспечение** (языковые средства для диалога пользователя с системой) <sup>11</sup>, **организационное обеспечение** (методики организации проектирования, внедрения и эксплуатации системы) <sup>12</sup>, **технологическое обеспечение** (технологические схемы обработки информации в системе) <sup>13</sup>, **правовое обеспечение** (нормативно-правовые документы, регламентирующие работу системы) <sup>14</sup>, **эргономическое обеспечение** (средства обеспечения удобства и безопасности труда пользователей системы) <sup>15</sup>, **кадровое обеспечение** (персонал ИТ-специалистов для сопровождения системы) <sup>16</sup>. Эти обеспечивающие подсистемы являются общими для всей ЭИС и поддерживают работу ее функциональных частей.

## 3. Классификация программного обеспечения.

**Ответ:** Программное обеспечение (ПО) по своему назначению делится на три основные категории <sup>17</sup>: **системное ПО** – программы, управляющие работой компьютера и обеспечивающие среду для приложений (операционные системы, драйверы, сервисные утилиты и пр.) <sup>17</sup>; **прикладное ПО** – программы, решающие прикладные задачи пользователей (офисные приложения, системы управления предприятием, графические редакторы и др.); **инструментальное ПО** (средства разработки) – программы для создания другого ПО (компиляторы, библиотеки, интегрированные среды разработки) <sup>17</sup>. Также ПО классифицируют по другим признакам: по сфере применения (например, образовательное, научное, игровое), по степени доступности исходного кода (с открытым или закрытым кодом), по лицензии распространения (свободное, условно-бесплатное, коммерческое) и т.д. В целом такая классификация упорядочивает все виды программ и облегчает их изучение и выбор.

## 4. Эволюция и основные характеристики технологий разработки ПО.

**Ответ:** Разработка программного обеспечения прошла путь от кодирования на машинном языке до современных высокоуровневых методов и средств <sup>18</sup>. В конце 1950-х появились первые **языки высокого уровня** (Фортран, Алгол и др.), что ускорило программирование и расширило круг

решаемых задач <sup>19</sup> <sup>20</sup>. В 1960-70-е годы оформился **структурный подход** к программированию: программы строились как совокупность процедур и модулей, что улучшало понимание и сопровождение кода <sup>21</sup>. Затем, по мере роста сложности задач, возникли новые парадигмы: **функциональное программирование** (Лисп) и **логическое программирование** (Пролог) для задач ИИ <sup>22</sup>. В 1980-е получила распространение **объектно-ориентированная технология**, базирующаяся на принципах абстракции, инкапсуляции и наследования, что упростило моделирование сложных систем как совокупности взаимодействующих объектов. Параллельно развивались **CASE-средства** – автоматизированные системы поддержки проектирования, позволявшие генерировать код из моделей (хотя полностью заменить программирование они не смогли) <sup>23</sup>. Также эволюция коснулась процессов: в 1970-х сформировалась каскадная модель разработки (последовательные фазы анализа, кодирования, тестирования) <sup>24</sup>, позже модифицированная в спиральную (итерационное повторение фаз) <sup>25</sup>. С 1990-х распространились **гибкие методологии (Agile)**, предполагающие быструю итеративную разработку и частые поставки. Таким образом, технологии разработки ПО развивались от простых и жёстких к все более **абстрактным, автоматизированным и гибким**, что позволило успешнее справляться с ростом сложности программных систем.

## 5. Классификация стандартов. Области программной инженерии, охватываемые международными стандартами. Основные стандарты программной инженерии и их содержание.

**Ответ:** Стандарты можно классифицировать по охвату и статусу: **корпоративные** (внутренние стандарты фирм), **отраслевые** (для определённой отрасли, например СНИП в строительстве), **государственные (ГОСТ)** и **международные (ISO/IEC, IEEE и др.)** <sup>26</sup> <sup>27</sup>. В сфере программной инженерии международные стандарты охватывают различные области: процессы жизненного цикла ПО, управление качеством, оценку зрелости процессов, управление проектами, требования к документации, метрики и др. К основным стандартам относятся: **ISO/IEC 12207** – стандарт процессов жизненного цикла ПО (определяет понятия программного продукта, жизненного цикла, описывает набор основных процессов разработки, сопровождения и т.д.) <sup>28</sup>; **ISO/IEC 15504 (SPICE)** – модель оценки и улучшения процессов разработки (расширяет ISO 12207, вводит 6 уровней зрелости процессов, описывает схему аттестации процессов) <sup>29</sup>; **SEI CMMI** – модель зрелости процессов (определяет 5 уровней зрелости организации разработки ПО) <sup>30</sup>; **ISO 9001/90003** – стандарты систем менеджмента качества и их применение к ПО (требования к организационным процессам, ориентированным на качество) <sup>31</sup> <sup>32</sup>; **ISO/IEC 25010** (ранее ISO 9126) – модель качества ПО (определяет перечень характеристик качества и метрик для их оценки); **PMBOK** – свод знаний по управлению проектами <sup>33</sup>; **SWEBOK** – свод знаний по программной инженерии (описывает 10 ключевых областей знаний) <sup>34</sup>. Международная стандартизация развивается по линиям интеграции: например, стандарты ISO/IEC 12207 и ISO/IEC 15288 (системная инженерия) сейчас согласованы между собой и с соответствующими IEEE-стандартами <sup>35</sup> <sup>36</sup>. В целом стандарты программной инженерии содержат терминологию, процессы и требования, предназначенные для повышения качества и предсказуемости разработки ПО, и охватывают практически все аспекты жизненного цикла – от постановки требований до сопровождения и оценки процессов.

## 6. Определение стандарта и сертификации работы организации на соответствие стандарту.

**Ответ:** Стандарт – это нормативный документ, принятый признанным органом на основе консенсуса, который устанавливает правила, требования или характеристики к продуктам, процессам или услугам для многократного использования <sup>37</sup>. Проще говоря, стандарт – это эталон, которому добровольно (или обязательно) должны соответствовать определённые объекты.

**Сертификация** организации на соответствие стандарту – это процедура независимой оценки, в ходе которой уполномоченное сертификационное учреждение проверяет, что процессы или продукция организации удовлетворяют требованиям выбранного стандарта, и выдает **сертификат соответствия**. Например, сертификация по ISO 9001 подтверждает, что система менеджмента качества организации отвечает требованиям этого стандарта. Таким образом, стандарт задаёт **нормы**, а сертификация – это официальное подтверждение того, что организация соблюдает эти нормы в своей работе.

## 7. Международная практика разработки стандартов в области программной инженерии.

**Ответ:** Разработка международных стандартов по программной инженерии ведётся преимущественно организациями **ISO/IEC JTC 1/SC 7** (подкомитет по программной и системной инженерии) в сотрудничестве с **IEEE**. В международной практике стандарты разрабатываются в ходе открытого обсуждения экспертами из разных стран и принимаются на основе консенсуса. За последние десятилетия наблюдается тенденция к **унификации стандартов**: число разных организаций-разработчиков сократилось, и индустрия консолидировалась вокруг ISO/IEC и IEEE-стандартов <sup>38</sup> <sup>39</sup>. Например, Министерство обороны США вместо собственных военных спецификаций перешло на коммерческие стандарты ISO/IEC и IEEE <sup>38</sup>. Международные стандарты носят рекомендательный характер, однако в глобальной экономике на них ориентируются производители и заказчики ПО для обеспечения совместимости и качества. Процесс разработки включает создание рабочих групп, несколько стадий черновиков (WD, CD, DIS и пр.) и принятие путем голосования национальных органов по стандартизации. В итоге международные стандарты (ISO, IEC, IEEE) охватывают все основные процессы жизненного цикла ПО <sup>40</sup>, и практика их разработки основана на прозрачности, учёте мирового опыта и регулярном обновлении по мере развития технологий. Организации по всему миру могут добровольно **сертифицироваться** на соответствие ключевым международным стандартам (например, ISO 9001, CMMI), что способствует признанию их качества на мировом рынке.

## 8. Определение и содержание CASE-технологий разработки ПО.

**Ответ:** **CASE-технология** (Computer-Aided Software Engineering) – это совокупность методологий и инструментальных средств, предназначенных для автоматизации процессов разработки и проектирования программного обеспечения <sup>41</sup>. Иными словами, CASE – это подход, при котором используются специальные **CASE-средства** для моделирования системы и автоматического генерирования артефактов, что повышает качество и ускоряет разработку программ. Основная цель CASE-технологий – **максимально отделить проектирование от кодирования**, автоматизировать рутинные этапы и обеспечить единое хранилище проектных данных <sup>42</sup>. Содержательно CASE-технология включает: применяемую **методологию** (например, структурный анализ SADT или объектно-ориентированный подход UML), **нотации** моделирования (графические языки диаграмм – функциональные, информационные, ER-диаграммы, UML-диаграммы и т.д.), а также **инструментальные средства** – программные продукты, позволяющие строить модели и автоматически генерировать из них код или документацию <sup>43</sup> <sup>44</sup>. В рамках CASE используют модели процессов (DFD, IDEF0), модели данных (ER-диаграммы), спецификации и т.д. – все эти элементы хранятся в едином **репозитории проекта**, что обеспечивает согласованность и совместную работу разработчиков <sup>45</sup> <sup>46</sup>. К типичным возможностям CASE-средств относятся: анализ и проектирование (графические редакторы диаграмм), автоматическое **генерирование кода**, ведение **словаря данных**, документирование системы, **управление версиями** и коллективная работа. Таким образом, CASE-технология – это **интеграция методов разработки ПО с**

**программными инструментами**, что позволяет повысить производительность труда, качество проектной документации и скорость внесения изменений в программные системы.

#### **9. Перечислите элементы функциональных схем.**

**Ответ:** Функциональная схема (диаграмма) представляет модель системы в виде функций (операций) и связей между ними. Основными элементами любой функциональной диаграммы являются **функциональные блоки** (обозначающие выполняемые системой функции или процессы) и **связи (дуги)** между ними, показывающие обмен информацией или материальными потоками <sup>47</sup>. На схемах структурного анализа, например SADT/IDEFO, функции изображаются блоками, а взаимодействия – стрелками разных типов (входы, выходы, механизмы, управления) <sup>48</sup> <sup>49</sup>. В диаграммах потоков данных (DFD) к функциональным элементам также относятся **процессы** (функции), **потоки данных** (стрелки), **источники/приёмники (внешние сущности)** и **хранилища данных** <sup>50</sup>. Таким образом, к элементам функциональной схемы можно отнести: **блок (функцию)**, который выполняет определённое действие; **стрелку (связь)**, передающую данные или ресурсы между функциями; а в зависимости от нотации – также обозначения **внешних объектов** (источников сигналов, потребителей результатов) и **накопителей данных** (если информационные потоки нужно сохранить). Комбинируя эти элементы – функции и связи – функциональные схемы отображают, **что делает система и как взаимодействуют ее части**.

#### **10. Определение и содержание технологического процесса проектирования ЭИС.**

**Ответ:** Технологический процесс проектирования ЭИС – это упорядоченная совокупность действий (операций) коллектива специалистов, направленная на разработку проекта информационной системы с заданными свойствами <sup>51</sup>. Он описывает, что нужно делать при проектировании, *в какой последовательности, какими средствами и кем* (распределение ролей) <sup>51</sup>. Содержательно технологический процесс охватывает весь жизненный цикл создания ЭИС: на каждой стадии – от предварительного обследования до внедрения – существуют свои методы и средства проектирования <sup>52</sup>. В типичном случае выделяют стадии: предварительное исследование и анализ требований, разработка концепции системы, техническое проектирование (детальное описание функций и подсистем) и рабочее проектирование (реализация и внедрение) – на каждой стадии применяется соответствующая технология и набор операций. Технологический процесс **регламентирует действия** (например, сбор требований, моделирование, кодирование, тестирование), их входы и выходы (документы, результаты), используемые методики (например, структурный анализ, объектное проектирование) и инструменты (CASE-средства, языки) <sup>53</sup> <sup>54</sup>. Таким образом, технологический процесс проектирования ЭИС – это формализованное описание как следует выполнять проектные работы, обеспечивая их полноту, последовательность и воспроизводимость результата.

#### **11. Классификация методов проектирования ЭИС.**

**Ответ:** Методы проектирования ЭИС можно классифицировать по нескольким признакам <sup>55</sup>. По степени автоматизации различают: *ручное проектирование* (без специальных инструментов, вся разработка ведётся “вручную”, например, кодирование на алгоритмических языках) и *автоматизированное проектирование* (с использованием CASE-средств, частичной генерацией или настройкой решений) <sup>56</sup>. По степени использования типовых решений: *оригинальное проектирование* – создание системы “с нуля”, индивидуальная разработка всех компонентов; *типовое проектирование* – настройка и сборка системы из готовых типовых модулей и решений (например, внедрение готового пакета с доработкой под требования) <sup>57</sup>. По степени адаптивности решений: выделяют методы *реконструкции* (адаптация имеющихся компонентов путем доработки

кода)<sup>58</sup>, параметризации (настройка системы изменением параметров, без программирования) <sup>59</sup> и реструктуризации модели (изменение модели предметной области с автоматической генерацией обновлённых решений) <sup>59</sup>. Сочетание этих признаков определяет стиль технологии проектирования. Например, каноническое (классическое) проектирование обычно ручное, оригинальное, с постепенной реконструкцией, а индустриальное проектирование – автоматизированное и/или типовое (параметризируемое) <sup>60</sup>. Каждый метод имеет область применения: оригинальные методы оправданы при уникальных требованиях, типовые – при наличии подходящих готовых компонентов; ручное проектирование применяется при небольших системах или отсутствии CASE-средств, автоматизированное – при больших проектах для повышения производительности. Таким образом, классификация методов проектирования ЭИС помогает выбрать подходящий подход в зависимости от наличия инструментов, повторного использования и требуемой гибкости.

## 12. Цели, задачи и содержание моделирования предметной области при проектировании ЭИС.

**Ответ:** Моделирование предметной области – это этап проектирования, на котором создаётся формальное описание реальной области деятельности (бизнес-процессов, объектов, правил), для которой создаётся информационная система. Цель такого моделирования – получить адекватную, понятную всем участникам проекта модель текущей деятельности, выявить проблемы и требования, которые должна решить будущая ИС <sup>61</sup>. Основные задачи: определить границы предметной области (что входит в сферу автоматизации, а что нет) <sup>62</sup>, собрать и проанализировать бизнес-правила и законы функционирования (семантические условия, например “каждый клиент имеет уникальный ID”) <sup>63</sup>, описать процессы и информацию – то есть какие функции выполняются, какие данные при этом используются и генерируются. Содержание моделирования включает построение набора моделей: функциональной модели (диаграммы, отражающие процессы и их взаимодействие – например, контекстная диаграмма и декомпозиции IDEF0/DFD) <sup>64</sup>, информационной модели (структура данных, ER-диаграммы, описывающие сущности и их атрибуты) и иногда динамической модели (например, сценарии или диаграммы состояний, если нужно описать поведение во времени). При моделировании аналитики проводят обследование объекта, интервью с экспертами, собирают документы – и на основе этого строят описание “как есть” (AS-IS). Полученная модель позволяет упорядочить и формализовать требования к будущей системе, увидеть процессы целиком <sup>65</sup>, выявить узкие места и области для улучшения. Критерием завершения моделирования предметной области служит достаточность детализации модели для понимания и постановки задач автоматизации – то есть модель должна отвечать на все вопросы, ради которых она создавалась <sup>66</sup> <sup>67</sup>. Таким образом, моделирование предметной области – это фундаментальный этап, обеспечивающий корректное понимание реальных нужд и правильную постановку задачи на разработку ЭИС.

## 13. Основные инструментальные средства структурного анализа и проектирования.

**Ответ:** В структурном анализе систем применяются несколько ключевых графических средств (нотаций) моделирования <sup>68</sup>, которые по сути являются инструментами аналитика:

- Диаграммы потоков данных (DFD) – отражают функции (процессы), которые должна выполнять система, и потоки данных между ними <sup>69</sup>. DFD позволяют проанализировать, как информация проходит через систему, в сочетании со словарём данных и спецификациями процессов <sup>70</sup>.
- Диаграммы “сущность-связь” (ERD) – отображают информационные сущности и отношения между ними <sup>71</sup>, служат для проектирования структуры базы данных системы.

- **Диаграммы переходов состояний (STD)** – моделируют поведение системы во времени, показывая различные состояния и события, вызывающие переходы <sup>72</sup>.

Перечисленные средства содержат графические символы для основных компонентов (процессы, потоки, хранилища, сущности) и текстовые описания для точного определения этих компонентов <sup>73</sup>. Например, DFD включает условные обозначения для **процессов, внешних сущностей, хранилищ данных и потоков данных** <sup>50</sup>. ER-диаграмма – символы для **сущностей** (прямоугольники), **связей** (ромбы/линии с пометками) и **атрибутов** (ovalы или списки). State-charts – круги/квадраты для **состояний** и стрелки для **переходов**.

Таким образом, под основными инструментальными средствами структурного анализа подразумеваются **типовые виды диаграмм**, поддерживаемые CASE-инструментами, – прежде всего DFD, ERD и STD. Они реализованы во многих программах (например, BPwin, RationalRose, Visio и др.), что позволяет аналитикам создавать модели, анализировать требования и проектировать систему на наглядном уровне <sup>31</sup> <sup>74</sup>. Использование этих средств способствует визуализации функций и данных, облегчает коммуникацию между заказчиком и разработчиками, а также служит основой для генерации технической документации.

#### 14. Базовые принципы методологии SADT.

**Ответ:** Методология **SADT** (Structured Analysis and Design Technique) основывается на ряде важных принципов <sup>75</sup> <sup>76</sup>:

- **Целевое моделирование:** перед началом необходимо чётко определить **цель модели** – набор вопросов, на которые модель должна ответить <sup>75</sup>. Модель SADT всегда создаётся под конкретную цель и рассматривается с выбранной точки зрения.
- **Единая точка зрения:** у модели должен быть только **один субъект и одна точка зрения** на систему <sup>77</sup> <sup>78</sup>. Это значит, что все диаграммы в SADT-модели описывают систему с одной договорённой позиции (например, с точки зрения руководства предприятия), что обеспечивает её целостность и непротиворечивость.
- **Декомпозиция:** сложная система отображается иерархией связанных диаграмм. **Каждая функция может быть разложена** (декомпозирована) на подфункции на следующем уровне, а все диаграммы вместе образуют серию, описывающую систему с увеличением детализации <sup>79</sup> <sup>48</sup>. Принцип “черного ящика”: на каждом уровне показывается только ограниченное число (3-6) блоков, детали раскрываются на нижележащих диаграммах.
- **Интерфейсность (ICOM):** у каждого функционального блока четко определены **интерфейсы**: *Input* (входы слева), *Output* (выходы справа), *Control* (управляющие воздействия сверху) и *Mechanism* (механизмы/ресурсы снизу) <sup>48</sup> <sup>49</sup>. Этот принцип обеспечивает строгую форму записи: любые воздействия на функцию классифицируются по типу и связываются стрелками со стороны, соответствующей их роли.
- **Совмещение графики и текста:** SADT использует параллельно **графический язык** (диаграммы) и **естественный язык** (текстовые описания) <sup>80</sup>. Каждая диаграмма сопровождается словесным описанием (глосарием или описанием блоков/стрелок), что гарантирует строгость и однозначность модели. Графическое представление делает модель наглядной, а текстовое – точной.
- **Итеративность и рецензирование:** процесс построения SADT-модели носит итеративный характер. Предусмотрено **итеративное уточнение** модели с участием экспертов предметной области (рецензирование) <sup>81</sup> <sup>82</sup> – это гарантирует корректность модели и согласие всех заинтересованных сторон.

Следуя этим принципам, SADT-модель обеспечивает **полное, точное и адекватное описание** системы под заданным углом <sup>83</sup>. Она представляет систему как совокупность взаимодействующих функций, структурированную по уровням детализации, с чётко обозначенными границами и интерфейсами. Такой подход позволяет аналитикам и пользователям прийти к общему пониманию того, что делает система и как она это делает, ещё до этапа кодирования.

## 15. Назначение, основные понятия и структурные элементы стандарта IDEF0.

**Ответ:** IDEF0 – это стандарт функционального моделирования, предназначенный для описания функций системы и потоков данных/материалов, которыми эти функции оперируют. Его **назначение** – представить структуру деятельности организации или системы в виде иерархии функций (работ) и показать, какие входы преобразуются в выходы под воздействием управлений и механизмов. IDEF0 широко используется для моделирования бизнес-процессов предприятия на этапе обследования (AS-IS) и проектирования улучшений (TO-BE). Основные понятия IDEF0 унаследованы от методологии SADT: **функциональный блок** (Activity, функция) – элемент диаграммы, обозначающий какую-либо работу или процесс; **стрелка** – обозначает интерфейс функции, причем различают четыре типа стрелок по стороне подключения к блоку <sup>48</sup> <sup>49</sup>: **вход** (Input, слева – то, что потребляется или преобразуется функцией), **выход** (Output, справа – результат выполнения функции), **управление** (Control, сверху – управляющие воздействия, регламентирующие выполнение, например требования, стандарты) и **механизм** (Mechanism, снизу – ресурсы, с помощью которых выполняется функция: люди, оборудование, ПО). Структурными элементами IDEF0-диаграммы являются именно эти блоки и стрелки (различных типов). Каждая IDEF0-модель включает **контекстную диаграмму** (A-0), показывающую саму систему как единый блок с внешними интерфейсами, и **дерево узлов** (Node Tree) – иерархию номеров диаграмм <sup>84</sup>. Детализация осуществляется на **декомпозиционных диаграммах**: каждый блок верхнего уровня может быть расшифрован на отдельной диаграмме нижнего уровня, где показаны его подфункции и интерфейсные стрелки <sup>85</sup> <sup>86</sup>. При этом связь между уровнями поддерживается через **номера узлов** (идентификаторы диаграмм) и ICOM-коды стрелок, чтобы явно отразить, какие стрелки унаследованы с родительской диаграммы. Таким образом, стандарт IDEF0 регламентирует **нотацию** (графические обозначения блоков и стрелок) и **правила моделирования** (порядок декомпозиции, ограничение 3-6 блоков на диаграмме, согласование интерфейсов), что позволяет создавать строго формализованные функциональные модели системы. Назначение IDEF0 – дать всем участникам проекта **наглядную функциональную схему** рассматриваемой системы, понятную как аналитикам, так и экспертам предметной области.

## 16. Назначение, основные понятия и структурные элементы методики диаграмм потоков данных (DFD).

**Ответ:** DFD (Data Flow Diagrams) – методика структурного анализа, предназначенная для моделирования информационных потоков и процессов системы. Ее **назначение** – показать, **как данные движутся** через систему, какие трансформации (процессы) над ними выполняются, и куда данные поступают или где хранятся. DFD особенно полезны на этапе определения требований, т.к. фокусируются на логике обработки информации. Основные понятия DFD включают четыре типа сущностей <sup>87</sup> <sup>88</sup>: **процесс** (он же функция или работа – изображается кругом или прямоугольником с разделённой частью) – преобразует входные данные в выходные; **поток данных** (стрелка) – показывает перемещение данных от одного элемента к другому; **хранилище данных** (две параллельные линии или открытый прямоугольник) – место долгосрочного хранения данных, к которому могут обращаться процессы; **внешняя сущность (источник/приёмник)** – объект вне системы (пользователь, внешняя система), который предоставляет данные системе или получает результаты. Структурные элементы DFD – это именно перечисленные символы: процессы, внешние

сущности, хранилища и потоки данных, – и связи между ними <sup>50</sup>. На диаграммах **нет жёсткого разделения сторон входа/выхода** для потоков, в отличие от IDEF0: стрелка может входить в процесс с любой стороны, важно только направление потока <sup>89</sup>. DFD-модель строится обычно **иерархически**: начинается с **контекстной диаграммы** (показывает систему одним процессом и все внешние источники/приёмники с потоками между ними), затем следует **де композиция** – разбиение главного процесса на более мелкие на диаграмме уровня 1, и т.д., пока процессы не станут элементарными <sup>90</sup> <sup>91</sup>. Связь между уровнями осуществляется через **нумерацию процессов** (например, процесс 2 на верхнем уровне расшифровывается диаграммой 2, где процессы 2.1, 2.2 и т.д.) и через сохранение консистентности потоков (входы/выходы верхнего процесса должны распределиться между под процессами). Таким образом, DFD предоставляет аналитикам средство **описания функциональных требований** – как каждый процесс преобразует входные данные в выходные и взаимодействует с другими процессами и хранилищами <sup>92</sup>. Строгость DFD ниже, чем у IDEF0 (нет категорий стрелок), зато они проще для понимания. В результате DFD часто используются для **анализа текущей системы и спецификации требований**, а затем могут дополняться другими моделями (IDEF0, IDEF3) для более подробного проектирования <sup>93</sup>.

## 17. Сравнительная характеристика IDEF0- и DFD-технологий структурного анализа и их отличие.

**Ответ:** IDEF0 и DFD – оба являются методами структурного функционального моделирования, но имеют разный фокус и нотацию. **IDEF0** описывает систему как совокупность взаимосвязанных **функций (работ)**, акцентируя что делает система, при каких входах и управлении, и с помощью каких ресурсов <sup>94</sup>. **DFD** рассматривает систему как совокупность **данных и процессов**, акцентируя как **данные перемещаются и трансформируются** <sup>94</sup> <sup>95</sup>. Главное отличие – в трактовке стрелок и правил их использования. В IDEF0 **стрелки строго типизированы** по стороне подключения (вход, выход, управление, механизм) и представляют скорее **потенциальные взаимосвязи** (требуемое присутствие того или иного объекта данных/ресурса) <sup>48</sup>. В DFD все **потоки данных унифицированы** (не разделяются на вход/управление – любая стрелка просто несёт данные) и представляют **реальное движение данных** между компонентами <sup>96</sup> <sup>89</sup>. **IDEF0-модель** требует обязательного отображения управляющих воздействий (контролей) для каждой функции – это одна из ключевых отличительных черт: наличие управления сверху считается обязательным “по хорошему тону” (каждый блок должен выполняться на основе каких-то правил) <sup>97</sup>. В DFD же управление явно не выделяется – потоки могут нести как основные данные, так и управляющую информацию, нет категоризации стрелок. **Хранилища данных**: в IDEF0 нет отдельного символа “база данных” – если нужно отразить обращение к хранилищу, его можно показать либо как механизм (например, ресурс “БД”) либо вообще вне модели. В DFD присутствуют специальный элемент “накопитель данных” (Storage), отражающий базы данных или файлы внутри модели <sup>50</sup>. **Декомпозиция**: и IDEF0, и DFD поддерживают иерархическое разложение. Однако в IDEF0 при декомпозиции требуется жесткое соответствие интерфейсов – стрелки родительского блока должны “стыковаться” со стрелками дочерней диаграммы; для этого используются ICOM-коды и/или механизмы туннелей (чтобы отразить появление или исчезновение стрелок) <sup>98</sup>. В DFD подход более гибкий: можно добавлять или устранять потоки на нижних диаграммах без формальных обозначений – считается, что более детальные процессы могут вводить новые данные или отбрасывать их. **Диаграммы-представления**: IDEF0 обычно сопровождается деревом узлов (структура модели) и словарём описаний функций и стрелок. DFD также предполагает словарь данных и мини-спецификации процессов для детализации каждой “элементарной” функции. По применению: **IDEF0** лучше подходит для **высокоуровневого функционального моделирования** предприятия, когда важно участие ресурсов и условий (например, регламенты, подразделения – их можно отразить как механизмы/управления).

**DFD** удобен для анализа потоков информации, при разработке требований к программной системе – он более интуитивен для программистов, фокусируется на данных и алгоритмах. Как правило, IDEF0-модель более строгая и статичная (не показывает последовательности), а DFD – более **динамичный**: поток данных косвенно указывает порядок, хотя формально время не задано (можно понять, что сначала данные идут из источника в процесс, потом в хранилище и т.д.). В итоге, обе методики часто используются совместно: IDEF0 – чтобы зафиксировать функции и границы системы, DFD – чтобы проработать логику обработки данных внутри этих функций <sup>99</sup>. Основные отличия резюмируя: **IDEF0** требует категории стрелок (Input/Control/Output/Mechanism) и ограничивает число блоков на диаграмме, DFD этого не имеет; IDEF0 не вводит явного образа “данного хранилища”, DFD имеет специальный символ; IDEF0 отображает взаимосвязи более абстрактно (условные зависимости), DFD – более конкретно (фактическое движение данных).

## 18. Понятие, использование и критерии окончания декомпозиции при моделировании процессов.

**Ответ:** Декомпозиция – это разделение модели процесса на более мелкие составляющие (подпроцессы) с целью упростить понимание и разработку. В контексте моделирования бизнес-процессов декомпозиция позволяет представить сложный процесс в виде иерархии уровней, постепенно детализируя его. **Понятие:** декомпозиция предполагает, что любой процесс можно разложить на ряд более элементарных действий, которые совместно выполняют исходную функцию. Например, высокоуровневый процесс “Управление заказами” можно декомпостировать на “Прием заказа”, “Обработка оплаты”, “Отгрузка товара” и т.д. **Использование:** при моделировании процессов декомпозиция применяется итеративно: сначала строят модель верхнего уровня (контекст), затем каждый крупный блок раскрывают диаграммой следующего уровня, и так продолжают, пока не достигнут требуемой детализации <sup>90</sup> <sup>100</sup>. Инструментально это реализовано во многих нотациях – например, в IDEF0 декомпозиция блоков, в DFD – детализация процессов уровня 0 до уровня 1, 2 и т.д. Декомпозиция прекращается, когда дальнейшее дробление нецелесообразно. **Критерии окончания декомпозиции:** Рекомендуется заканчивать моделирование, когда детализация модели достаточна для достижения ее цели <sup>66</sup>. Иными словами, нужно остановиться, когда дальнейшее углубление не даст новой полезной информации и не требуется для понимания или разработки. Формально в литературе выделяют такие ситуации, при которых **дальнейшую декомпозицию прекращают** <sup>101</sup> <sup>102</sup>: (1) текущий процесс уже описан **достаточно подробно**, и его описание отвечает на все вопросы, поставленные целью моделирования; (2) дальнейшая детализация потребует **смены уровня абстракции** (то есть выйдет за рамки принятой точки зрения) или (3) **смены точки зрения** (например, углубляясь дальше, нужно рассматривать процесс с позиции другого подразделения – значит, текущая модель достигла предела в рамках данной точки зрения); (4) если выявлено, что разбиваемый блок **тривиален** (прост до очевидности) – его нет смысла декомпозирировать; (5) если блок повторяет содержание другого блока (или сильно похож) – лучше не дублировать его декомпозицию. Еще один практический критерий – **сравнение с типовым образом**: когда несколько нижних блоков уже достаточно просты, автор модели может решить, что аналогичные блоки тоже нет нужды детализировать, а описать текстом <sup>67</sup> <sup>103</sup>. Например, если один из блоков расписан до конкретных инструкций (и это удовлетворяет цели модели), другие блоки аналогичного уровня сложности можно тоже считать элементарными. В итоге, завершение декомпозиции – **сбалансированное решение** аналитика: модель должна быть максимально простой, но при этом раскрывать все существенные аспекты процесса. Прекратить декомпозицию следует там, где **каждый элементарный процесс понятен исполнителям и может быть реализован без дополнительного дробления**.

## 19. Назначение, основные понятия и структурные элементы стандарта IDEF3.

**Ответ:** IDEF3 – стандарт описания процессов (Process Flow Description Capture Method), предназначенный для документирования последовательности действий и логики сценариев (workflow) в организации. Его **назначение** – зафиксировать **как именно протекает процесс**, от начала до конца, включая ветвления и синхронизации, в форме, понятной экспертам предметной области <sup>104</sup>. В отличие от IDEF0, который описывает **что делается**, IDEF3 моделирует **как делается** – то есть **причинно-следственные связи между событиями и ситуациями** в процессе <sup>105</sup>. Основные понятия IDEF3 включают: **UOB (Unit of Behavior)** – единица работы, элементарное действие или шаг процесса (графически изображается прямоугольником с описанием) <sup>106</sup>; **соединители (links)** – связи между UOB, показывающие последовательность или зависимости. Есть два типа связей: **связь предшествования (precedence link)** – направленная стрелка от одного UOB к другому, указывающая, что один шаг следует за другим <sup>107</sup>; и **связь отношения (relational link)** – пунктирная линия, отражающая другую зависимость или примечание (например, связь между шагом и объектом ссылки) <sup>108</sup>. **Объект ссылки (Referent)** – специальный элемент (показан иконкой "документа" или окружностью), используемый для добавления комментариев, аннотаций или ссылки на другой процесс/диаграмму <sup>109</sup>. Структурные элементы диаграммы IDEF3, таким образом: **блоки-UOB, стрелки-связи (разных видов) и объекты ссылок**. Кроме того, ключевая особенность IDEF3 – наличие **перекрёстков (junctions)** для моделирования ветвлений и слияний потоков. Перекрёстки бывают нескольких видов: AND, OR, XOR – как на входе, так и на выходе <sup>110</sup> <sup>111</sup>. Они обозначаются небольшими значками на линии связи:

- AND (точка или перечеркнутый круг) – означает, что должны выполниться **все** входящие шаги (для слияния) или будут запущены **все** исходящие шаги (для разветвления). Есть **асинхронный AND** (все предшествующие завершены, после чего все последующие могут начаться) и **синхронный AND** (все предшествующие завершаются одновременно и запускают одновременный старт всех последующих) <sup>112</sup>.
- OR – логическое ИЛИ (на значке надпись "1..n"): **один** или **несколько** шагов должны выполниться. Также различают обычный OR (асинхронный: выполняются один или несколько, порядок неважен) и синхронный OR (несколько ветвей выполняются параллельно, но из не полного набора) <sup>113</sup>.
- XOR (исключающее ИЛИ, значок "X") – **только один** из последующих шагов выполняется (при разветвлении) или только один из потоков поступает (при слиянии) <sup>114</sup>.

Перекрёстки позволяют задавать сложную логику маршрутов процесса – альтернативы и параллелизм. IDEF3-диаграммы могут использоваться как самостоятельно, так и в дополнение к IDEF0/DFD для уточнения динамики процессов <sup>115</sup>. В модели IDEF3 обычно выделяют **процессные диаграммы** (Process Flow Description) и при необходимости **диаграммы состояний объектов** (Object State Description) – вторые показывают, как меняются состояния важного объекта через последовательность процессов. Но основные структурные элементы в обоих случаях сходны: блоки-события и связи-переходы, плюс логические узлы. Резюме: IDEF3 предназначен для **описания потоков работ (workflow)**, основой которого являются **единицы работы (шаги процесса)** и **отношения предшествования между ними**. Эта нотация широко применяется для **анализа и оптимизации бизнес-процессов**, когда важно понять последовательность операций и варианты их выполнения <sup>116</sup> <sup>117</sup>.

## 20. Назначение и использование перекрёстков и объектов ссылок на диаграммах IDEF3.

**Ответ:** В IDEF3-диagramмах **перекрёстки (junctions)** служат для отображения ветвлений и слияний ветвей процесса – то есть определяют правила **разветвления/объединения потоков работ**. Назначение перекрёстков – задать логику управления последовательностью шагов: например,

"после шага A одновременно запускаются B и C" (AND-разветвление) или "после шага A выполняется либо B, либо C" (XOR-разветвление), либо "шаг D начинается только когда завершены и B, и C" (AND-слияние), и т.д. Их использование: перекрёсток помещается на связь (линк) в месте, где одна ветвь процесса расходится на несколько или несколько сходятся в одну, и снабжается обозначением типа – **AND, OR или XOR**, а также модификатором синхронности (если нужно указать одновременность) <sup>118</sup>

<sup>111</sup>. Например, в нотации IDEF3 кружок с буквой "A" может означать AND, "O" – OR, "X" – XOR; двойная линия над символом – синхронный (все одновременно). С помощью комбинации перекрёстков можно задать достаточно сложные логические маршруты. **Объекты ссылок (Referents)** предназначены для **структурирования и пояснения схемы**. Их назначение – связывать диаграммы между собой и добавлять пояснения, не перегружая основную схему. Существует несколько видов объектов ссылок: *Jump Referent* – ссылка-переход, указывающая на продолжение процесса на другой диаграмме (например, когда модель процесса разбита на несколько страниц – на одной ставят объект "переход на диаграмму X"); *Note (примечание)* – объект для текстового комментария, поясняющего деталь или делающего оговорку; *Object Referent* – может обозначать состояние или объект, используемый в нескольких местах. В IDEF3 объект ссылки изображается в виде особого значка (например, строка текста в скобках или иконка документа) <sup>109</sup>. **Использование объектов ссылок:** если процесс слишком сложный, его можно разбить на две диаграммы и поставить на конце первой объект-ссылку, указывающий, что далее – см. другую диаграмму. Или, если нужно отметить, что на шаг влияет некое внешнее условие, можно добавить примечание. Объекты ссылок помогают **избежать дублирования**: например, один подпроцесс может использоваться в разных местах – вместо рисования его дважды, рисуют один раз, а в остальном ставят объекты-ссылки на него. Таким образом, перекрёстки в IDEF3 обеспечивают выразительность логики процессов (параллелизм/альтернативность), а объекты ссылок – **гибкость и читабельность схемы**, позволяя разбивать и комментировать модель без потери связности. Оба элемента делают IDEF3 инструментом для описания реальных процессов, где почти всегда есть ветвления, синхронизации и необходимость пояснений.

## 21. Назначение инstrumentального средства AllFusion Process Modeler.

**Ответ:** AllFusion Process Modeler (BPwin) – это CASE-средство, предназначенное для визуального моделирования, анализа и документирования бизнес-процессов организации. Его основное назначение – помочь аналитикам создавать **модели процессов** в нотациях структурного анализа (IDEF0, DFD, IDEF3) и тем самым улучшать понимание и оптимизацию деятельности предприятия <sup>119</sup>. Программа предоставляет удобный графический интерфейс для построения диаграмм, генерации отчетов и хранения описаний в репозитории. С помощью AllFusion Process Modeler можно **графически представить бизнес-процессы** любой сложности, визуализируя последовательность работ, потоки информации и ответственность подразделений <sup>120</sup>. Это средство используется на проектах реорганизации процессов, внедрения корпоративных информационных систем, стандартизации деятельности. Оно позволяет четко задокументировать текущие процессы ("как есть") и спроектировать новые ("как должно быть"), выявить лишние операции, проверить соответствие стандартам (например, ISO 9000) <sup>121</sup> <sup>122</sup>. Встроенные функции анализа (например, оценка затрат и производительности процессов) и генерации отчетов облегчают принятие управлеченческих решений. Таким образом, AllFusion Process Modeler служит **"электронной доской"** для **бизнес-аналитика**, повышая эффективность моделирования процессов: вместо чертежей на бумаге – формальные диаграммы, которые легко править, обсуждать и использовать для дальнейшего проектирования информационных систем.

## **22. Какую методологию поддерживает AllFusion Process Modeler (AIIF PM).**

**Ответ:** AllFusion Process Modeler поддерживает методологии **структурного анализа и моделирования бизнес-процессов**, в частности построение моделей в нотациях **IDEF0, DFD и IDEF3**<sup>123</sup>. Эти нотации относятся к структурным (процессно-ориентированным) методологиям проектирования. Таким образом, инструмент позволяет применять методологию функционального моделирования SADT/IDEF0 (для описания функций и их интерфейсов), методологию потоков данных (DFD – для описания информационных процессов) и сценарный подход IDEF3 (для документирования последовательностей действий). Кроме того, AIIF PM может интегрироваться с ERwin для поддержки методологии информационного моделирования (IDEF1X для моделей данных). Но в фокусе AllFusion Process Modeler именно **процессные методологии** – он предоставляет шаблоны и правила, соответствующие стандартам этих методологий, и потому широко используется для поддержки проектов по описанию и совершенствованию бизнес-процессов.

## **23. Назначение элементов управления и возможности настройки системы AIIF PM.**

**Ответ:** В AllFusion Process Modeler под **элементами управления** понимаются основные элементы интерфейса – такие как пункты меню, кнопки на панелях инструментов, диалоговые окна настроек – с помощью которых пользователь управляет программой. Их назначение – обеспечивать быстрый доступ к функциям моделирования. Например, на **основной панели инструментов** есть кнопки “Создать модель” (File/New), “Открыть модель” (File/Open), “Сохранить” (Save), “Печать” (Print) – они соответствуют стандартным операциям работы с файлами модели<sup>124</sup> <sup>125</sup>. Имеются кнопки вызова специальных утилит: **Report Builder** (генератор отчетов по модели)<sup>126</sup>, **Model Explorer** (включение/выключение обозревателя модели)<sup>127</sup>, инструментов масштабирования (Zoom)<sup>128</sup>, проверки орфографии документации и т.д. Эти элементы управления упрощают навигацию и редактирование модели: так, Model Explorer показывает древовидную структуру модели (список диаграмм и объектов), позволяя быстро переключаться между разделами модели<sup>129</sup>.

**Настройка системы** AIIF PM возможна через меню *Tools/Options* и другие диалоги – пользователь может адаптировать среду под свои нужды. Возможности настройки включают: изменение **параметров отображения** (например, сетка на диаграмме, шрифты и цвета для блоков и стрелок), настройку **нотаций** (выбор обозначений Gane-Sarson или Yourdon-DeMarco для DFD, стиль стрелок в IDEF0 и пр.), конфигурацию **генератора отчетов** (какие разделы модели включать в отчет), интеграцию с другими инструментами (например, настройки взаимодействия с хранилищем Model Mart или с ERwin). Можно настраивать **панели инструментов** – добавлять или убирать кнопки, менять их расположение. Администраторы могут задать **шаблоны моделей** (стандартные словари, оформительские стандарты) для унификации работы нескольких пользователей. Таким образом, AllFusion Process Modeler предоставляет достаточно гибкую настройку, что позволяет организациям внедрять его в свою работу, учитывая собственные стандарты моделирования (например, собственные словари терминов, пользовательские атрибуты для объектов модели и т.п.). В целом элементы управления AIIF PM предназначены для удобства работы аналитика, а возможности настройки – для адаптации инструмента к конкретной методике и стандартам предприятия.

## **24. Для каких целей используются диалоги «Свойство активностей» и «Свойство стрелок».**

**Ответ:** В AllFusion Process Modeler (BPwin) **диалог “Свойства активности”** предназначен для просмотра и редактирования атрибутов выбранного функционального блока (активности) на диаграмме. Через этот диалог пользователь задаёт или изменяет **имя функции**, ее **описание** (текстовое пояснение, часто включающее цель или алгоритм работы), может указать **идентификатор** (например, номер по дереву узлов), ответственное подразделение или исполнителя (если модель

предусматривает такие поля), а также присоединять различные заметки, ссылки на документы, показатели затрат/времени и т.д. Проще говоря, диалог "Свойства активности" позволяет детализировать **что именно представляет собой данная функция**: ее роль в модели, ограничения, связанные документы. Именно туда аналитик заносит всю поясняющую информацию, которая не отображается на графике, но должна храниться в модели (например, описание логики работы процесса).

**Диалог "Свойства стрелки"** служит аналогичной цели для объекта типа "стрелка" (связь, поток). В этом окне задаются атрибуты *потока данных или материального объекта*, который стрелка представляет. Обычно это **имя стрелки** (наименование информационного потока, например "Заказ клиента"), его **описание** – содержимое или структура данных, единицы измерения, формат, источник и приемник. Если ведётся словарь стрелок, то именно в свойствах стрелки указывается ссылка на соответствующую запись словаря или вводится определение. Также можно отметить тип стрелки (например, в IDEF0 – Input/Output/Control/Mechanism – это определяется местом, но дополнительные свойства могут отмечаться).

Оба диалога используются для **наполнения модели семантикой**: графически на диаграмме отображены только названия блоков и стрелок, а полный смысл раскрывается в их свойствах. Практически, когда аналитик создаёт модель, он размещает блоки и соединяет их стрелками, после чего через указанные диалоговые окна **документирует каждый элемент** – что означает блок, что несёт стрелка. Эта информация затем может быть включена в отчёты (например, в пояснительную записку к модели). Таким образом, "Свойства активностей" и "Свойства стрелок" – ключевые инструменты для создания само-документированной модели: они обеспечивают, что **каждый процесс и каждый поток в модели имеют четкое определение**, понятное разработчикам и заинтересованным лицам.

## 25. Перечислите возможности работы со стрелками при создании IDEF0 модели. Назначение и использование туннелей в моделях.

**Ответ:** В IDEF0-моделях **стрелки** играют важнейшую роль, отображая интерфейсы функций. При создании модели доступны следующие возможности работы со стрелками:

- **Маршрутизация и привязка стрелок:** стрелки можно соединять с любыми сторонами блоков в соответствии с их смыслом (слева – вход, сверху – управление, справа – выход, снизу – механизм). Инструмент позволяет наглядно рисовать стрелки, изгибать их для обхода других объектов, объединять несколько стрелок в одну линию (сходящиеся потоки) или, наоборот, **разветвлять одну стрелку на несколько выходных** (если один результат используется в нескольких местах).
- **Разветвление и слияние стрелок:** по методологии SADT/IDEF0, стрелка может **ветвиться** (один и тот же выходной поток направляется в несколько блоков-приемников) или **сливаться** (несколько стрелок сходятся перед входом в один блок, обозначая, что совокупность нескольких потоков поступает как единый вход). Графически это реализуется либо с помощью узлов-соединителей, либо просто выводом нескольких линий из одного порта. Слияние стрелок позволяет отразить объединение данных: например, два разных источника информации сходятся и образуют единый вход процесса.
- **Наследование стрелок при декомпозиции:** когда функция A0 декомпозируется на дочернюю диаграмму A1..A3, входящие/выходящие стрелки верхнего уровня **унаследуются** дочерней диаграммой. Инструмент автоматически выводит их на граничные "parking lot" области диаграммы низкого уровня. Пользователь может редактировать привязку: какая конкретно подфункция получает вход, какой выдаёт выход и т.д. При этом для соответствия интерфейсов используются

**коды ICOM** – маркировка стрелок буквами I, O, C, M и номерами, чтобы указать их связь с родительской диаграммой (см. вопрос 27).

- **Изменение вида и стиля стрелок:** средство моделирования позволяет настроить отображение – например, стрелки могут быть сплошными или штриховыми, разной толщины, цвета – иногда этим пользуются для визуального различия типов информации (например, материальный поток vs информационный). Также можно включать или отключать отображение имен стрелок на диаграмме, в некоторых методологиях – помечать стрелки номерами (ID).

**Туннели** в IDEF0 – это специальный механизм для обозначения стрелок, которые “прерываются” на границе диаграммы, не отображаясь на уровне выше или ниже. **Назначение туннелей** – упростить модель, скрыв некоторые интерфейсы от верхних уровней детализации, если они несущественны в контексте тех уровней. Например, если на детализированной диаграмме появляется вспомогательный выход, не значимый для общей картины системы, его можно “затуннелировать”, чтобы на родительской диаграмме не рисовать эту стрелку. **Использование:** туннель изображается двумя маленькими круглыми скобками у начала или конца стрелки на границе диаграммы <sup>130</sup>. Если скобки вокруг начала стрелки (там, где она выходит из границы дочерней диаграммы), значит стрелка **возникает в этой декомпозиции и не поднимается вверх** – на родительском уровне её нет <sup>130</sup>. Если скобки стоят в месте, где стрелка входит в границу (исчезает), – стрелка “уходит в туннель”, т.е. была на вышестоящей диаграмме, а на этом уровне далее не прослеживается. Проще говоря, туннелирование позволяет **скрыть входы/выходы**, которые локальны для данной детализации. Это важно для поддержания принципа “скрытия сложности”: верхний уровень схемы не перегружается второстепенными деталями. Например, на диаграмме высокого уровня у функции может не показываться какой-то внутренний сервисный выход, хотя на низком уровне он будет, – этот выход обозначается как туннельный. В IDEF0 методических рекомендациях отмечается, что злоупотреблять туннелями не стоит <sup>131</sup>, лучше использовать их минимально и по необходимости (вместо них часто рекомендуют применять разветвления/слияния стрелок, чтобы все важные связи были явными). Тем не менее, туннели – полезный инструмент для **упрощения схемы**: они дают гибкость в отображении модели на разных уровнях абстракции без противоречий. Резюмируя: **возможности работы со стрелками** – ветвление, объединение, наследование, описание (именование) и стилизация; а **туннели** позволяют “ввести стрелку из ниоткуда” на данном уровне или “спрятать” её на следующий уровень, не нарушая связность модели, но скрывая детали, которые не должны фигурировать вне локального контекста.

## 26. Назначение, использование и техника работы со словарём стрелок.

**Ответ:** Словарь стрелок – это справочник (таблица) всех потоков, используемых в модели (IDEF0/DFD), содержащий определения и описания этих стрелок. Его **назначение** – обеспечить единообразие и непротиворечивость терминологии: каждое имя стрелки (например, “Заявка на закупку”, “Отчёт о продажах”) описывается ровно один раз в словаре, и это описание автоматически связывается со всеми вхождениями данной стрелки на диаграммах. Таким образом, словарь стрелок служит **глоссарием данных и материальных потоков** системы, устраниет двусмысленности (все участники проекта видят, что именно подразумевается под тем или иным потоком) и уменьшает дублирование – если одна и та же сущность используется в разных процессах, она описывается единообразно <sup>132</sup> <sup>50</sup>.

**Использование словаря стрелок** в AllFusion Process Modeler происходит следующим образом: пользователь при создании или переименовании стрелки может занести ее в словарь, указав определение. Обычно система предлагает интерфейс (например, окно “Arrow Dictionary”), где

перечислены все стрелки модели. Аналитик заполняет для каждой **атрибуты**: текстовое определение, возможно, синонимы, тип данных, формат, единицы измерения, источник и назначение – любую информацию, которая важна для понимания сущности, переносимой стрелкой. Когда на диаграммах рисуются одинаковые стрелки, инструмент может ссылаться на одну запись словаря (так, если на разных диаграммах есть поток "Заказ клиента", словарь хранит описание "Заказ клиента – документ, содержащий..."). Техника работы включает: открытие словаря (обычно командами меню или кнопкой), добавление новой записи либо выбор существующей для назначение стрелке, редактирование записей (при изменении требований можно обновить описание в одном месте – оно обновится везде). Разработчики могут также импортировать/экспортировать словарь – например, выгрузить его в Word/Excel для согласования с бизнес-заказчиком.

Пример: стрелка "Отчёт о продажах" – в словаре можно описать "Отчёт о продажах – ежемесячный отчет, содержащий данные о проданных товарах, выручке и пр., готовится отделом продаж и передается в бухгалтерию". На диаграммах, где стрелка "Отчёт о продажах" присутствует, можно либо не дублировать описание (зная, что оно есть в словаре), либо настроить отображение всплывающей подсказки/примечания с этим текстом.

**Техника работы:** рекомендуется при моделировании сразу заносить новые стрелки в словарь, чтобы затем не возникло разнотолков. В конце моделирования проводится **вычитка словаря** – проверяют, нет ли дубликатов (например, "Заказ" и "Заказ клиента" – может оказаться одним и тем же, их нужно унифицировать). Также словарь стрелок часто включается в итоговую документацию по проекту, чтобы все заинтересованные лица могли понять, какие информационные потоки фигурируют в системе.

Таким образом, словарь стрелок – это **инструмент согласования терминологии и содержания потоков**, позволяющий **работать с моделью на уровне семантики**, а не только графики. С его помощью модель ЭИС становится самодостаточной: помимо схем, она содержит формальное описание всех данных, которыми оперирует система, что облегчает и разработку (можно использовать словарь для постановки задачи программистам), и дальнейшее сопровождение (новым сотрудникам будет понятна терминология).

## 27. Назначение и использование ICOM-кодов.

**Ответ:** ICOM-коды – это обозначения, используемые в IDEF0-диаграммах для идентификации интерфейсных стрелок при декомпозиции функций. Аббревиатура ICOM расшифровывается как Input, Control, Output, Mechanism. **Назначение ICOM-кодов** – гарантировать правильное сопряжение (стыковку) стрелок между родительской функцией и дочерней диаграммой ее декомпозиции <sup>133</sup>. Когда блок разложен на подблоки, важно отследить, какие входы, выходы и др. интерфейсы "наследуются" сверху, а какие появляются или исчезают. Для этого в нижней диаграмме около граничных стрелок проставляют код, указывающий их тип и номер. Например, если у родительского блока A1 было два входа, то на декомпозиционной диаграмме A1 эти же стрелки появятся как I1 и I2 (Input 1, Input 2). Аналогично, управляющие – C1, C2, выходы – O1, O2, механизмы – M1, M2. Эти индексы соответствуют порядку стрелок, изображённых на родительском контексте слева-направо (или сверху-вниз для управлений) <sup>48</sup> <sup>49</sup>. Если на дочерней диаграмме появляется новая стрелка, которой не было сверху (например, дополнительный выход), её помечают туннельными скобками, а кода ICOM у неё нет – это сигнал, что стрелка **не унаследована** с уровня выше <sup>130</sup>.

**Использование ICOM-кодов:** обычно в программном средстве они генерируются автоматически, когда вы размещаете граничные стрелки. Аналитик должен проверить и, при необходимости, исправить соответствие: например, если местоположение стрелок поменялось, нужно обновить нумерацию. ICOM-коды проставляются в круглых скобках рядом с названием стрелки на диаграмме декомпозиции. Так достигается "стыковка": просматривая диаграмму, любой может понять, откуда пришла данная стрелка. Кроме того, при составлении отчётов IDEF0 ICOM-коды используются для таблиц данных: обычно приводится раздел "Interfaces" для каждого блока, где перечислены I1 – ... – все входы с их описанием, C1 – все управление и т.д.

По сути, ICOM-коды – это **метки интерфейсов**. Они гарантируют, что модель непротиворечива: если на верхнем уровне у функции было три выхода, то у диаграммы-низ на выходе будут О1, О2, О3 – ровно три, ни больше ни меньше (исключение – туннели). Это одно из правил методологии SADT/IDEF0, обеспечивающее точное согласование между диаграммами <sup>86 134</sup>.

Пример: Родительский блок "Обработать заказ" имеет вход "Заказ клиента" (I1) и управление "Политика скидок" (C1), выходы "Счёт" (O1) и "Уведомление на склад" (O2), механизм "Менеджер по продажам" (M1). На диаграмме декомпозиции этого блока A0.1, на соответствующих граничных стрелках будут указаны (I1) Заказ клиента, (C1) Политика скидок, (O1) Счёт, (O2) Уведомление на склад, (M1) Менеджер по продажам – это указывает, что эти стрелки пришли сверху. Если на декомпозиции появится дополнительный выход "Статистика отказов", и решено, что он не важен наверху, его обозначат туннелированным (без О-кода, со скобками).

Таким образом, **ICOM-коды используются при декомпозиции для отслеживания интерфейсов**: они позволяют однозначно сопоставить стрелки разных уровней, облегчая проверку модели и чтение диаграмм.

## 28. Использование диаграмм дерева узлов и FEO и техника их создания.

**Ответ:** В IDEF0-моделировании помимо основных функциональных диаграмм используются вспомогательные: **диаграмма дерева узлов (Node Tree)** и **диаграммы FEO (For Exposition Only)**. **Диаграмма дерева узлов** отображает **иерархическую структуру модели** – то есть, как связаны между собой все диаграммы и функции. По сути, это "оглавление" модели в графическом виде: корневой узел – контекстная диаграмма A-0, от него отходят узлы A0, A1, ... – диаграммы верхнего уровня, далее ветви A1 разбиваются на A1.1, A1.2 и т.д. (если функция A1 декомпозирована), и так далее на все уровни <sup>84</sup>. Элементы дерева узлов – **блоки-функции** (представленные узлами дерева) и связи между ними, отражающие отношения "родитель–детали". **Использование:** диаграмма дерева узлов служит удобной навигационной схемой и проверкой полноты модели. Она позволяет быстро увидеть, какие узлы (функции) **расшифрованы далее**, а какие – нет, увидеть структуру номеров. Обычно дерево узлов строится автоматически CASE-средством на основе номеров узлов, либо может рисоваться вручную аналитиком для отчётности. Техника создания: начинают с узла, соответствующего верхней функции (например, "A0: Управление предприятием"), от него ветвят узлы для каждой функции на диаграмме A0 (A1, A2, A3, например "Маркетинг", "Продажи", "Закупки"), затем для каждого узла, у которого есть декомпозиция, рисуются дочерние узлы второго уровня (например, A2.1, A2.2 – подфункции функции "Продажи"), и т.д. Конечные узлы – те, что не разложены (элементарные функции). Диаграмма дерева узлов, таким образом, **показывает состав и структуру модели** в одном месте, ее часто включают в пояснительную записку, чтобы дать целостное представление о модели.

**Диаграммы FEO (For Exposition Only)** – это диаграммы, создаваемые **для пояснения отдельных аспектов модели и не входящие в основную иерархию декомпозиции**. Назначение FEO-диаграмм – предоставить дополнительный вид или альтернативную точку зрения на часть системы, которую неудобно или нежелательно помещать в строгую иерархию IDEF0. Используются они, когда надо показать какой-то фрагмент модели обособленно, без необходимости детальной проработки всех интерфейсов. Например, FEO-диаграммой можно изобразить будущую ("to-be") схему процесса рядом с текущей ("as-is"), или показать подробности, выходящие за рамки основной цели моделирования. Методически FEO-диаграммы помечаются особым образом и **не нумеруются** в общей последовательности узлов (им дают условные номера или буквы). **Техника создания:** в BPwin при добавлении диаграммы можно указать флаг "For exposition only" – тогда она не будет требовать полного соблюдения правил (например, можно не отображать все интерфейсы). Аналитик вручную размещает на ней нужные блоки и стрелки, но такая диаграмма **не участвует в структурной декомпозиции**. В отчёте FEO-диаграммы могут приводиться для иллюстрации альтернативных сценариев или обсуждения. Также FEO применяют, чтобы разобрать некий частный случай: скажем, есть сложный процесс с несколькими вариантами – один вариант можно вынести на FEO-диаграмму для пояснения, не загромождая основную.

Пример: Модель предприятия содержит IDEF0-диаграммы A0, A1...A4 для описания текущих функций. Аналитики предлагают новую функцию – ее можно изобразить на FEO-диаграмме "F1", не включая в основную иерархию (поскольку она еще не реализована). Эта диаграмма поможет обсудить предложение, не ломая согласованность основной модели.

Таким образом, **дерево узлов** используется для *структурирования и навигации* по модели (показывает, как разбиты функции по уровням), а **FEO-диаграммы** – для *дополнительной демонстрации* или анализа фрагментов системы вне жесткой структуры модели. Оба типа диаграмм улучшают экспрессивность методологии: дерево узлов дает обзор архитектуры модели, FEO – гибкость в изложении нюансов.

## 29. Возможные нотации в методиках, возможных при построении диаграмм в AllFusion PM.

**Ответ:** AllFusion Process Modeler поддерживает несколько **нотаций структурного моделирования**, позволяющих создавать разные виды диаграмм. В контексте методик, доступных в AllFusion PM, основными являются: **IDEF0**, **DFD** и **IDEF3**<sup>123</sup>. Это означает, что при построении диаграмм пользователь может выбрать нотацию IDEF0 (для функциональных моделей), нотацию DFD (диаграммы потоков данных, поддерживаются, как правило, оба популярных стандарта – Yourdon/DeMarco или Gane/Sarson), а также нотацию IDEF3 (для описания процессов-последовательностей). В версиях BPwin (AllFusion PM) 4.x есть возможность переключаться между этими методиками для разных диаграмм модели<sup>135</sup>. Кроме того, AllFusion PM предоставляет базовые графические элементы, которые могут применяться и вне строгих стандартов – например, блок-схемы (flowcharts) как упрощённая нотация – однако основной упор сделан на перечисленные стандарты семейства IDEF. Также инструмент интегрируется с ERwin, который поддерживает нотацию **IDEF1X** (для моделей "сущность-связь" баз данных). Таким образом, **возможные нотации** при работе с AllFusion Process Modeler – это стандартизованные нотации структурного анализа: IDEF0 (функции и интерфейсы), IDEF3 (workflow-процессы), DFD (потоки данных) – каждая со своими графическими обозначениями, шаблонами и правилами, заложенными в систему. Благодаря этому аналитик может в рамках одного проекта создавать и функциональные модели, и динамические диаграммы, и информационные потоки, выбирая подходящую нотацию для каждой задачи моделирования.

### **30. Нормативно-методическое обеспечение создания ПО и стандарты его жизненного цикла.**

**Ответ:** Нормативно-методическое обеспечение разработки программного обеспечения – это совокупность официальных стандартов, методических указаний и регламентов, определяющих процесс создания и сопровождения ПО. Оно включает как **международные стандарты**, так и **национальные (государственные)**. Среди ключевых стандартов жизненного цикла ПО: **ISO/IEC 12207 "Процессы жизненного цикла программных средств"** – устанавливает унифицированную модель жизненного цикла с набором процессов, действий и задач, рекомендуемых при разработке (от соглашений с заказчиком до вывода из эксплуатации) <sup>31</sup>. Национальным эквивалентом является, например, **ГОСТ Р ISO/IEC 12207** в России. Также международный **ISO/IEC 15288** (для систем в целом, включает ПО) часто применяется совместно, обеспечивая совместимость процессов системной и программной инженерии <sup>31</sup>. Для управления качеством жизненного цикла существует серия **ISO 9000** (в частности **ISO 90003** – применение ISO 9001 к ПО). Для методического обеспечения очень важны стандарты на документацию: например, старый советский **ГОСТ 19** и **ГОСТ 34** – комплекс стандартов на техническую документацию и стадии создания автоматизированных систем. **ГОСТ 34.601-90 "Стадии создания АС"** определяет типовую последовательность этапов работ при создании информационных систем <sup>136</sup>, включая: формирование требований, разработку концепции, техническое задание, эскизный и технический проекты, рабочую документацию, внедрение и сопровождение <sup>137</sup> <sup>138</sup>. **ГОСТ 34.602-89** регламентирует состав и оформление технического задания на систему <sup>139</sup>, **ГОСТ 34.603-92** – виды испытаний системы <sup>140</sup> и т.д. В США распространены модели зрелости процессов, например **CMMI**, которые тоже относятся к нормативно-методическим средствам: они определяют, каким требованиям должна удовлетворять организация-разработчик на каждом уровне зрелости процессов. В итоге нормативно-методическое обеспечение предоставляет разработчикам и заказчикам **единый язык и требования**: какие документы нужно выпустить, какие этапы пройти, как оценить качество продукта. Придерживание таких стандартов жизненного цикла (как ISO 12207 или ГОСТ 34) позволяет повысить предсказуемость и управляемость проектов, облегчает взаимодействие разных организаций и сертификацию ПО.

### **31. Перечислить этапы создания ИС и содержание основных процессов в соответствии с различными нормативами и стандартами.**

**Ответ:** Различные стандарты и модели ЖЦ выделяют схожие по смыслу этапы создания информационных систем, хотя могут называть и группировать их по-разному. **Классический каскадный подход** предусматривает этапы: 1) **Предпроектное обследование и анализ требований** – исследование текущей системы, сбор и формализация требований, обоснование необходимости разработки (на выходе – техническое задание или аналог) <sup>141</sup>; 2) **Проектирование системы** – разработка архитектуры и деталей: разбивка системы на модули, разработка базы данных, интерфейсов (на выходе – технический проект, спецификации) <sup>142</sup>; 3) **Реализация (кодирование)** – программирование модулей, разработка и наполнение БД, проведение модульных тестов (выход – исходный код, исполняемые модули, рабочая документация) <sup>143</sup>; 4) **Тестирование и внедрение** – интеграция системы, комплексное тестирование, опытная эксплуатация, обучение пользователей, ввод системы в промышленную эксплуатацию (выход – акт ввода, система функционирует у заказчика) <sup>144</sup>; 5) **Эксплуатация и сопровождение** – штатная работа системы, в ходе которой устраняются обнаруженные ошибки, вносятся изменения, возможна модернизация (фактически цикл разработки повторяется итеративно для новых версий) <sup>145</sup>. Стандарты **ГОСТ 34** дают похожие стадии, разбивая некоторые на подстадии: напр., 1) Формирование требований (обследование объекта, ТЭО), 2) Разработка концепции, 3) Техническое задание, 4) Эскизный проект, 5) Технический проект, 6) Рабочий проект, 7) Ввод в эксплуатацию (опытная, промышленная), 8) Сопровождение <sup>137</sup> <sup>138</sup>. Международный стандарт **ISO/IEC 12207** вместо последовательных "этапов"

описывает **процессы** жизненного цикла: *процесс разработки* (включает стадии от анализа требований к ПО до квалификационного тестирования), *процесс сопровождения*, *процесс эксплуатации*, а также вспомогательные процессы (конфигурационное управление, обеспечение качества, верификация/валидация) и организационные (управление проектом, обеспечение инфраструктуры). Тем не менее, содержание основных работ аналогично – сбор требований, проектирование, реализация, тестирование, установка, поддержка. **Spiral** и **Agile** подходы перераспределяют эти этапы по итерациям, но внутри итерации всё равно есть мини-этапы анализа, дизайна, кодинга и теста. Таким образом, несмотря на разные нормативы, **базовые этапы создания ИС** обычно охватывают: **анализ и планирование** → **проектирование** → **разработка и тестирование** → **внедрение** → **эксплуатацию и сопровождение**, а стандарты отличаются детализацией и возможной параллельностью выполнения этих процессов. Например, **спиральная модель** предусматривает повтор многократно цикла “планирование – анализ рисков – разработка – оценка” для каждой итерации, а **инкрементная** – последовательное добавление функциональности через повторение этапов разработки для каждого инкремента. Но суммарно все они в той или иной форме реализуют вышеперечисленные основные процессы жизненного цикла.

### **32. Определение технологической операции и технологической сети проектирования ЭИС.**

**Ответ:** В контексте технологии проектирования ЭИС под **технологической операцией** понимается отдельный, относительно самостоятельный шаг процесса проектирования, имеющий чётко определённые входы и выходы. Формально технологическая операция проектирования ЭИС – это минимальный фрагмент проектного процесса, представимый в виде пятёрки {Вход, Преобразователь, Выход, Ресурсы, Средства} <sup>146</sup>. Проще говоря, технологическая операция – это конкретное действие или задача, выполняемая проектировщиками (или инструментальными средствами) и приводящая к результату. Например, “сбор требований у пользователя” – технологическая операция, входом которой являются исходные данные от заказчика, выходом – сформулированные требования; преобразователь – метод интервью, ресурсы – аналитик и время, средства – анкеты, диктофон. Или “написание модуля кода” – вход: дизайн-модель, выход: исходный код модуля, преобразователь: процесс кодирования, ресурсы: программист, средства: IDE. Таким образом, технологическая операция характеризуется своей **функцией** (что делается), **входными продуктами** (что нужно на входе), **выходными продуктами** (что получается на выходе) и **затратами** (какие ресурсы и инструменты требуются для её выполнения) <sup>146</sup> <sup>147</sup>.

**Технологическая сеть проектирования** – это совокупность технологических операций, выстроенных в виде сети (графа), отражающей логическую и временную структуру процесса проектирования ЭИС. Иначе говоря, технологическая сеть – это модель процесса проектирования, где узлы – технологические операции, а связи – зависимости и порядок их выполнения. Например, сеть может показывать, что операция “Анализ требований” предшествует операции “Разработка технического задания”, которая разветвляется на параллельные операции “Проектирование БД” и “Проектирование модулей”, после чего следует операция “Сборка прототипа” и т.д. Каждая связь несёт на себе выход одного шага, который является входом для другого. Такая сеть часто изображается схематически, подобно сетевому графику (PERT/CPM диаграмма), либо описывается формальными средствами (как предлагал Хотяшов Е.Н., введя аппарат технологических сетей) <sup>148</sup> <sup>146</sup>. Цель технологической сети – формализовать процедуру проектирования, выявить, какие работы могут выполняться параллельно, какие последовательно, где узкие места. Если сравнить с привычным – это аналог **блока-схемы процесса разработки или календарного плана**, только с акцентом на входы/выходы операций.

Применение: с помощью технологической сети можно просчитать трудоёмкость проекта, спланировать ресурсы. Например, зная для каждой операции требуемые квалификации и трудозатраты (нормы на вход R – ресурсы <sup>149</sup>), и выстроив сеть, менеджер получает маршрутную карту проектирования. Кроме того, формально описав все операции (вплоть до математического описания их входов/выходов и алгоритмов), можно частично автоматизировать процесс – на этом строятся некоторые CASE-средства, предлагающие “мастера” выполнения тех или иных проектных операций.

Таким образом: **технологическая операция** – “атомарный” шаг процесса проектирования, а **технологическая сеть** – структура, объединяющая множество таких операций в единую процессную модель разработки ЭИС.

### **33. Принципиальные различия, достоинства, недостатки и границы применимости стилей проектирования ПО.**

**Ответ:** В программной инженерии под **стилями проектирования** обычно понимают разные подходы к разбиению системы на компоненты и организации процесса разработки. Рассмотрим три основных стиля: **структурно-функциональный (процедурный)**, **объектно-ориентированный** и **компонентно-ориентированный**. Их принципиальные различия:

- **Структурный (процедурный) стиль:** система проектируется как иерархия функций (процедур) и модулей, данные хранятся отдельно и передаются между процедурами. Главный принцип – *топ-даун декомпозиция*: большая задача последовательно делится на подзадачи. Достоинства: относительная простота – понятная, математически обоснованная структура, хорошо подходит для алгоритмических задач; модули получаются независимыми по функционалу, что облегчает отладку; требуется менее сложный аппарат (достаточно структурных языков, теории алгоритмов). Недостатки: с ростом размера системы процедурный стиль приводит к *разрастанию связей* – данные глобально доступны многим процедурам, что усложняет сопровождение; повторное использование кода затруднено (процедуры завязаны на конкретные глобальные структуры); модель предметной области отображается неявно (только через функции). Границы применимости: эффективен для сравнительно небольших систем с хорошо формализованными алгоритмами (например, вычислительные задачи, встроенные системы), где изменения редки. Для больших информационных систем с часто меняющимися требованиями структурный подход уже неудобен – внесение изменений в функциональной декомпозиции может затронуть много модулей.

- **Объектно-ориентированный (ОО) стиль:** система проектируется как совокупность взаимодействующих объектов, каждый из которых объединяет данные и методы их обработки (принципы ООР: инкапсуляция, наследование, полиморфизм). Принципиальное отличие – *центричность данных*: модель строится от реальных существований предметной области, которые представлены классами, а функции являются поведением этих существований. Достоинства: лучшая *модульность* и *повторное использование* – объектные классы можно переиспользовать в других проектах или расширять через наследование; *понижают связность* – взаимодействие ограничено явно определенными сообщениями, что облегчает сопровождение; код ближе к реальной модели, благодаря чему легче общаться с экспертами (можно обсуждать объекты, их свойства). Объектный подход хорошо поддерживается современными языками и инструментами, существует развитый аппарат дизайна (UML, паттерны). Недостатки: более *сложен для понимания новичками* – требует освоения концепций

ООА/ООП; может приводить к избыточности кода (много небольших методов, шаблонный код), несколько снижается производительность (из-за уровней абстракции, виртуальных вызовов – хотя сейчас это не критично). Также выделяют проблему “объектно-реляционного несоответствия” – при работе с базами данных нужно преодолевать разрыв между объектами в коде и таблицами в БД. Граница применимости: ОО-стиль предпочтителен для *больших и сложных систем*, которые будут развиваться, потому что облегчает масштабирование и изменение (добавление нового класса часто локализовано). Он применяется практически везде в индустрии – от приложений до игр. Однако для очень простых скриптов или одноразовых утилит полный ООП может быть избыточен (проще написать пару процедур). Также, если требования крайне нестабильны, объектную иерархию может часто ломать – тогда стоит применить гибкие техники (интерфейсы, композицию).

- **Компонентно-ориентированный стиль:** система рассматривается как *набор независимых компонентов*, взаимодействующих через четко определенные интерфейсы. Компонент – более крупная единица, чем класс, обычно это цельный модуль/сервис, который можно разворачивать и заменять независимо (например, веб-сервис, библиотека, микросервис). Отличие от ОО: акцент на *разделение по функциональности и повторное использование готовых блоков*, часто без доступа к их исходному коду. Достоинства: **быстрота разработки** за счёт повторного использования – можно собрать систему из существующих компонентов (например, взять готовый модуль авторизации, платежей и интегрировать); **масштабируемость** – компоненты могут разворачиваться распределенно, заменяться без переписывания всей системы; компонентный подход хорошо совместим с DevOps (микросервисы). Недостатки: **сложность интеграции** – различные компоненты (особенно если от разных производителей) могут иметь несовместимые интерфейсы, потребовать “клейкого” кода; зависимость от сторонних поставщиков – если компонент не соответствует полностью требованиям, его адаптация может быть сложной; общая оптимизация затруднена (каждый компонент – чёрный ящик, сложно оптимизировать весь конгломерат). Кроме того, архитектура системы становится распределенной, надо учитывать надежность взаимодействий, сеть и т.п. Граница применимости: эффективна, когда существует богатый выбор готовых решений (например, корпоративные системы – CRM, ERP часто строятся компонентно, используя готовые модули). Компонентный стиль применяется, например, в архитектуре **SOA** (service-oriented) и **микросервисов**. Он хорош для крупных корпоративных или веб-систем, которые нужно быстро адаптировать, масштабировать, где разные команды могут разрабатывать разные сервисы. Не особо оправдан для небольших проектов – усилия на организацию инфраструктуры компонентов будут больше, чем выгода.

Подводя итог, **структурный vs объектный vs компонентный**: структурный проще, но плохо масштабируется; объектный – золотая середина для ПО средней и высокой сложности, повышает качество и гибкость; компонентный – ещё более высокий уровень абстракции, оптимален для распределенных систем и повторного использования, но требует зрелой инфраструктуры и управления. В реальных проектах элементы стилей могут сочетаться: например, внутри компонента используется ОО-подход, а на самом верхнем уровне система строится из компонентов. Выбор стиля определяется размером проекта, наличием готовых решений, требованиями к изменяемости – и является одним из ключевых решений архитектуры.

#### 34. Содержание этапов классического жизненного цикла, его достоинства и недостатки.

**Ответ:** Классический жизненный цикл разработки (каскадная модель, “водопад”) разделён на

последовательные этапы, каждый из которых должен быть завершён перед началом следующего <sup>150</sup> <sub>151</sub>. Основные этапы: (1) **Системное исследование и анализ требований**. На этой стадии собираются и документируются требования пользователя, формируется техническое задание. Заканчивается этап, как правило, утверждением ТЗ. (2) **Проектирование** (концептуальное и подробное). Здесь архитекторы и аналитики разрабатывают структуру системы: разбивают её на модули, разрабатывают базы данных, интерфейсы, алгоритмы. Результат – спецификации, схемы, макеты. (3) **Реализация (кодирование)**. Программисты пишут код согласно спецификациям, создают базы данных, возможно, отдельные модули сначала прототипируются. Этап завершается интеграцией модулей. (4) **Тестирование и верификация**. Сначала модульное тестирование, затем интеграционное, системное и приемочное тестирование. Проверяется соответствие системы требованиям из ТЗ. (5) **Внедрение (переход в эксплуатацию)**. Система устанавливается на рабочее место заказчика, проводится обучение персонала, опытная эксплуатация, затем принимается в промышленную эксплуатацию актом приемки. (6) **Эксплуатация и сопровождение**. На протяжении эксплуатации разработчики исправляют выявленные ошибки, выполняют небольшие доработки и поддерживают работоспособность системы до конца ее службы. Все этапы следуют друг за другом, обратного хода нет (только через официальные процедуры изменения требований).

**Достоинства классической модели:** она проста и **чётко структурирована** <sup>150</sup> – легко понять, что за чем идёт, ответственность на каждом этапе определена. Для заказчика и менеджеров такая модель даёт **определенность сроков и результатов**: на каждую стадию планируется комплект документации, который утверждается. Контроль качества встроен по фазам – нельзя прыгнуть к кодированию, не согласовав требования, что снижает риск забыть важные моменты. Подходит, когда требования ясны с самого начала и маловероятно их изменение: в таких условиях каскадная схема **минимизирует затраты на управление**, ведь нет сложных итераций и параллельной работы. Кроме того, всю систему можно просчитать заранее (оценка трудоёмкости по этапам, бюджет).

**Недостатки:** Главный минус – **негибкость к изменениям и длительная отдача**. Требования обычно уточняются в ходе работы, но каскадная модель плохо это учитывает: внести изменения “задним числом” дорого, так как нужно пересматривать уже выполненные и утвержденные документы и переделывать последующие фазы. Часто конечный продукт не вполне устраивает пользователя, так как к моменту его сдачи требования могли частично устареть <sup>152</sup>, или заказчик лишь увидев систему, понял, что хотел другого. Также **отсутствует ранняя видимость**: заказчик получит что-то работающее только в самом конце, а до того – документацию. Это ведёт к высоким рискам – если что-то пойдет не так, узнается об этом поздно. Тестирование сосредоточено ближе к концу, что повышает риск обнаружения фундаментальных проблем на поздних стадиях. Ещё недостаток – каскад плохо масштабируется для очень крупных проектов: длительность процесса и передачи между этапами приводят к потере контекста, а параллельно работать над разными фазами сложно (каждая ждёт предыдущую). В современных условиях, когда требования и технологии быстро меняются, каскадная модель считается устаревшей для многих типов ПО.

**Границы применимости:** классический жизненный цикл оправдан для **относительно небольших или хорошо определённых проектов**, например, разработка ПО по строго заданной спецификации (в оборонной отрасли раньше применялось). Сейчас его достоинства (упорядоченность, документированность) стараются сохранить в модифицированных моделях (V-модель, каскад с прототипированием), однако в чистом виде waterfall применяется редко – только там, где изменения действительно минимальны и цена ошибки очень высока (например, ПО для ядерных объектов, где обязательны жёсткие этапы и верификация).

### **35. Характеристика инкрементной модели жизненного цикла, его достоинства и недостатки.**

**Ответ:** Инкрементная модель жизненного цикла предполагает разработку системы не целиком сразу, а по частям (инкрементами), с постепенным наращиванием функциональности. Процесс выглядит как серия мини-водопадов: на первом инкременте реализуется базовый набор функций, затем на следующих – добавляются новые возможности, пока не будет получена полная система. Каждый инкремент проходит все необходимые этапы (анализ, дизайн, код, тест) <sup>153</sup>, но объем работы на каждом этапе меньше, чем в случае единственного цикла на всю систему. **Характеристика:** Уже после первого цикла заказчик получает рабочий продукт пусть с урезанным функционалом, но пригодный к использованию. Последующие версии выходят регулярно, добавляя новые функции. Важен момент приоритизации – требования распределяются по выпускам: наиболее критичные включаются в первый релиз, менее важные – в последующие. Таким образом, инкрементная модель – **итерационная**, но с укрупнёнными релизами, каждый из которых представляет ценность.

**Достоинства:** Снижается риск неудачи проекта, так как **результат поставляется поэтапно**, и по ходу можно скорректировать направление <sup>154</sup>. Заказчик получает ценность раньше – первый инкремент может решить часть проблем сразу (повышение удовлетворенности). **Тестирование и отладка упрощаются** – в каждом инкременте меньше объем, легче локализовать ошибки <sup>155</sup>. **Управление рисками улучшается** – можно пересмотреть сроки и требования в каждом цикле, учесть новые риски <sup>154</sup>. Кроме того, инкрементный подход более гибок к изменениям: требования к будущим инкрементам можно уточнять, опираясь на опыт использования предыдущих версий. Команда может параллельно работать: например, пока одна часть допиливается, другая проектируется к следующему релизу (если позволяет размер команды). Инкрементная модель хорошо подходит при недоступности сразу всех ресурсов – можно развернуть базовую систему, а затем финансировать улучшения по мере возможностей.

**Недостатки:** Планирование в инкрементной модели сложнее: нужно четко продумать **архитектуру с запасом** – она должна выдержать постепенное наращивание, иначе на поздних инкрементах можно столкнуться с тем, что фундаментальные решения мешают новым требованиям <sup>156</sup> <sup>157</sup>. Есть риск, что интеграция новых модулей будет сложнее предполагавшегося, если архитектура не все учитывала. Общая трудоёмкость иногда выше, чем при разовой разработке, из-за накладных расходов на каждый цикл (повторные тестирования, интеграции). **Требуется хорошая разбивка проекта** – неправильное определение содержания инкрементов может привести к тому, что первые версии слишком урезаны и бесполезны, либо наоборот попытка в первый релиз включить слишком много – нарушит сроки. Ещё недостаток: заказчик может начать воспринимать первую рабочую версию как уже итоговую и недофинансировать последующие (есть известный синдром, когда макет или упрощенная система удовлетворяет на 80%, и развитие тормозится). С технической стороны: при инкрементной разработке **некоторые ошибки архитектуры могут выявиться поздно**, и исправление их при наличии уже выпущенных инкрементов может быть болезненно (но это свойственно всем итерационным моделям).

**Границы применимости:** инкрементная модель хорошо работает, когда есть возможность **поэтапного ввода** системы и требования можно разделить на приоритетные и второстепенные. Например, разработка информационных систем, веб-сервисов – часто делается версия 1.0 с core-функциями, затем 1.1, 1.2 с расширениями. Она не очень подходит, когда система должна быть сдана сразу целиком в полном объеме (например, ПО для космического корабля – его нельзя “инкрементно” запускать). Также, если требования плохо поддаются приоритизации (все критично),

инкременты трудно выделить. В целом же инкрементная модель объединяет плюсы каскада (четкость фаз внутри каждого выпуска) с плюсом итеративности (адаптивность) и потому очень популярна, особенно как часть Agile-подходов (например, Scrum фактически реализует короткие инкременты-спринты).

### **36. Особенности спиральной модели жизненного цикла.**

**Ответ:** **Спиральная модель** (разработана Барри Боэмом, 1986) – итеративная модель ЖЦ, в которой процесс разработки представляется в виде спирали, проходящей через набор фаз с акцентом на **анализ рисков на каждом витке**. Особенности: каждый цикл спирали включает примерно следующие шаги <sup>158</sup>: 1) **Определение целей, альтернатив и ограничений** для данного витка (например, определить цели следующей итерации, варианты реализации, ресурсы); 2) **Анализ и оценка рисков** – ключевой этап, когда команда рассматривает, какие риски связаны с выбранными альтернативами (технологические, рыночные и т.д.) и планирует как их снять. Часто на этом шаге создаются **прототипы** для проработки и уменьшения рисков (например, прототип интерфейса для риска непонимания требований пользователя); 3) **Разработка и проверка** – собственно реализация запланированного функционала итерации: дизайн, кодирование, тестирование (объем работ зависит от того, на каком витке – ближе к внешним виткам объем больше); 4) **Оценка результатов и планирование следующего витка** – совместно с заказчиком оценивается достигнутый результат, принимается решение, идти ли на следующий виток, какие цели поставить далее. Таким образом, модель **не фиксирует конкретно этапы “Анализ – Проектирование – Реализация – Тест”**, хотя они присутствуют внутри шага разработки, но четко фиксирует **итерационный цикл с контролем рисков**. Изображается это как спираль, выходящая из центра: каждый новый круг все ближе подводит к готовому продукту, расширяя функциональность и уточняя цели.

**Отличительные черты:** – **ориентация на риск**: перед каждым существенным решением проводится анализ рисков и по его результатам могут меняться подходы. – **Гибкость в наборах фаз**: спиральная модель – не жесткая последовательность, а рамка: на разных витках могут осуществляться разные активности (например, первый виток может вообще посвятить почти полностью выяснению требований и прототипированию, второй – архитектуре, третий – разработке версии 1, четвертый – расширению до версии 2). – **Непрерывная взаимодействие с заказчиком**: после каждого круга происходит оценка промежуточного результата и согласование планов (этот элемент потом очень воспринят Agile). – **Потенциально бесконечна**: модель подразумевает, что цикл повторяется, пока не будет достигнуто удовлетворение целям или пока риски/средства не остановят проект.

**Преимущества:** высокий уровень контроля рисков – перед вложением больших средств модель предусматривает их минимизацию, что особенно важно в крупномасштабных инновационных проектах. Гибкость – спираль допускает изменения требований: на новом витке можно скорректировать цель. Заказчик вовлекается на каждом цикле (понимает ход работ, видит прототипы). Недостатки: сложность управления – спираль требует квалифицированного управления рисками, что не всегда просто; трудность оценки бюджета и сроков с начала, т.к. не известно, сколько витков понадобится (планируется итеративно). Также модель может выродиться в затянувшееся прототипирование, если неправильно определять конечные цели.

**Пример применения:** разработка нового сложного ПО, где есть неизвестные факторы (новая технология, неясные требования) – сначала делают 0-й виток: изучение возможностей, прототип

концепции; затем 1-й: разработка базовой версии; 2-й: расширение функционала и т.д., анализируя риски (например, риск низкой производительности – делается прототип для измерения).

В итоге, **спиральная модель** объединяет систематичность каскада с итеративностью и проактивным управлением рисками. Она считается универсальной, но на практике напрямую встречается редко – чаще ее элементы входят в современные Agile-методы (которые тоже итеративны).

### **37. Отличие компонентно-ориентированной модели от спиральной и классической моделей ЖЦ информационной системы.**

**Ответ:** **Компонентно-ориентированная модель** жизненного цикла основывается на идее активного *повторного использования компонентов* (*reuse*) при создании ПО. Её иногда называют **CBDD (Component-Based Development)**. Главный отличительный момент: разработка ведётся не столько путём полного цикла “с нуля”, сколько путём **интеграции готовых программных компонентов** (модулей, служб) и разработки лишь недостающих частей. Это сказывается на этапах: большое внимание уделяется *анализу существующих решений* на рынке или внутри организации (этап выбора компонентов), а проектирование сводится к *определению архитектуры взаимодействия компонентов*. По сравнению с **классической моделью** (каскад), компонентно-ориентированный подход менее линеен: после стадии требований не обязательно делать полный собственный дизайн – сначала оцениваются готовые компоненты, возможно, корректируются требования под то, что доступно. Модель ЖЦ может выглядеть как: формирование требований -> поиск/выбор компонентов -> адаптация/разработка клающего кода -> интеграция и тестирование. То есть **этапы разработки и интеграции перемешаны** – часто компонентный стиль предполагает инкрементальную интеграцию: подключили компонент А, протестировали, затем компонент В и т.д.

**Отличия от спиральной модели:** спиральная – это про процесс с управлением рисками, а компонентная – про стратегию построения архитектуры из готовых блоков. Они совместимы: можно вести компонентно-ориентированную разработку итеративно. Но если сравнивать чисто:

- В спиральной модели основное – итерации с наращиванием функционала; в компонентной – основное *перераспределение труда*: много усилий уходит на интеграцию, меньше – на кодирование с нуля.
- Спираль не диктует, разрабатывать ли с нуля или брать готовое – там любой подход, лишь бы риски минимизировать (хотя использование готовых компонентов может быть одной из стратегий снижения рисков). В компонентном подходе это центральная идея: **максимум reuse**.
- По жизненному циклу: классический и спиральный модели предполагают полное тестирование всей системы после реализации. При компонентном подходе часто применяют концепцию “**сборка по готовности**”: как только подключён компонент, его сразу проверяют совместно с остальными (чтобы убедиться, чтостыковка прошла успешно).
- Документация: в каскаде и спирали упор на документацию требований и дизайна; в компонентном – дополнительно на спецификации интерфейсов компонентов, соглашения об их использовании, версии.

#### **Отличия от классической (водопадной) модели:**

- **Параллелизм и порядок:** классическая модель строго последовательна, компонентная же предполагает некоторый параллелизм работ: можно одновременно подбирать разные компоненты, работать над их адаптацией в разных командах (если архитектура разбита по компонентам).
- **Изменение ролей разработчиков:** при компонентном подходе роль “интегратора” становится

ключевой. Большую часть времени проектировщики тратят на оценку сторонних модулей, написание "адаптеров" между ними, на настройку конфигурации – это нехарактерно для классического подхода, где 80% – это собственно программирование функций по спецификации.

- **Стадия сопровождения:** при компонентной разработке сопровождение усложняется зависимостью от поставщиков компонентов (если сторонний компонент устареет или изменится лицензия, придется искать замену) – в классической модели всего ПО свое, сопровождение = поддержка собственного кода.

**Границы применимости компонентного подхода** ограничены доступностью качественных компонентов. Если система уникальна и ничего готового нет, то применить модель сложно – придется всё равно писать с нуля (т.е. каскад/спираль). Зато когда есть зрелые фреймворки, библиотеки, веб-сервисы – компонентный подход резко сокращает время разработки (например, веб-приложения сейчас часто строятся путем "склейки" множества библиотек/пакетов). Классическая и спиральная модели не предполагали по умолчанию такого обилия готового ПО (они из эпохи, когда почти всё писалось заново).

**Пример:** допустим, нужно реализовать интернет-магазин. По классической модели – полная разработка: анализ требований, дизайн, код (пишется своя система корзины, оплаты, каталога), тест. По спиральной – сделали прототип для снижения риска UX, потом первую версию – но тоже, скорее всего, писали сами, просто итерациями. По компонентному – берём готовую CMS или фреймворк, подключаем модуль оплаты от PayPal, модуль логистики от доставки, пишем лишь тонкий слой для специфических бизнес-правил. Итог: компонентная модель существенно **сокращает сроки**, но требует, чтобы требования были более-менее стандартными, соответствовали функционалу имеющихся компонентов (не всё можно гибко изменить).

В целом: **компонентно-ориентированная модель отличается акцентом на повторном использовании и интеграции**, тогда как классическая – на полном цикле разработки, а спиральная – на управлении рисками через итерации. Компонентный подход может быть встроен внутрь итеративных моделей: например, многие Agile-проекты по сути компонентно-ориентированные (активно используют библиотеки). Его плюс – быстрота и надежность (компоненты проверены), минус – зависимость и ограниченная гибкость (система частично диктуется возможностями компонентов). Классический подход противоположен – медленный, но полностью кастомный. Спиральный – средний: тоже кастомный, но позволяет корректировать курс. Таким образом, компонентная модель – **эволюция жизненного цикла**, возникшая благодаря насыщенности рынка готовых решений, и она дополняет, а не полностью заменяет прежние модели разработки.

---

- 1 2 3 4 Структура и компоненты эис, классификация  
<https://studfile.net/preview/9135822/page:2/>

5 6 Компоненты экономических информационных систем  
<https://studfile.net/preview/7125542/page:2/>

7 8 9 10 11 12 13 14 15 16 Обеспечивающие подсистемы ЭИС  
<https://necroms.narod.ru/o8.html>

17 Программное обеспечение - Российское общество Знание  
<https://znanierussia.ru/articles/>

%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5\_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5

18 19 20 21 22 23 24 25 1.2. Эволюция технологий программирования.  
<https://studfile.net/preview/2358842/page:2/>

26 27 28 29 30 33 34 136 139 140 Стандарты программной инженерии - презентация онлайн  
<https://ppt-online.org/691808>

31 32 35 36 38 39 40 74 Стандарты на процессы программной инженерии: сегодня и завтра |  
Открытые системы. СУБД | Издательство «Открытые системы»  
<https://www.osp.ru/os/2005/02/185312>

37 Что такое ГОСТ и ТУ  
<https://nccenter.ru/articles/poleznoe/gost-i-tu-ponyatiya-otlichiya-reglamentatsiya/>

41 42 CASE — Википедия  
<https://ru.wikipedia.org/wiki/CASE>

43 44 45 46 Технология разработки ПО: Методологические основы CASE технологий  
<https://technologiarpo.blogspot.com/p/case.html>

47 Состав функциональной модели sadt  
<https://studfile.net/preview/5615745/page:8/>

48 49 75 76 77 78 79 80 81 82 83 85 86 133 134 47. Основные принципы sadt моделирования.  
Структура sadt модели. Синтаксис sadt модели.  
<https://studfile.net/preview/8208544/page:22/>

50 68 69 70 71 72 73 90 91 100 132 Средства структурного анализа  
<https://studfile.net/preview/5615745/page:5/>

51 52 53 54 146 147 148 149 2.2. Технология проектирования эис  
<https://studfile.net/preview/3795470/page:10/>

55 56 57 58 59 60 141 142 143 144 145 Тема 1. Методологические основы проектирования эис  
<https://studfile.net/preview/4545730/>

61 62 63 64 Описание предметной области и определение цели проектирования  
<https://studfile.net/preview/7363576/page:6/>

65 Моделирование предметной области | ЛЕКЦИИ | Bd-Subd.Ru  
<http://bd-subd.ru/lekcii/modelirovaniye-predmetnoy-oblasti.htm>

66 67 101 102 103 17.2. Прекращение декомпозиции  
<https://studfile.net/preview/7831899/page:59/>

84 92 93 99 106 107 108 109 110 111 112 113 114 115 118 135 Idef3: назначение и основные элементы  
<https://studfile.net/preview/9497884/page:3/>

87 89 95 Использование DFD: как описать движение данных в бизнес ...  
<https://systems.education/data-flow-diagrams>

88 [PPT] Функциональное моделирование систем с использованием ...  
<https://moodle.enu.kz/mod/resource/view.php?id=28809>

94 Сравнение методологий DFD и IDEF0 - Bstudy  
[https://bstudy.net/822912/ekonomika/sravnenie\\_metodologiy\\_idef0](https://bstudy.net/822912/ekonomika/sravnenie_metodologiy_idef0)

96 6.13. Концепция DFD  
<http://www.kgau.ru/istiki/umk/mpb/ch06s13.html>

97 Основные методологии обследования организаций. Стандарт ...  
<https://www.cfin.ru/vernikov/idef/idef0.shtml>

98 124 125 126 127 128 129 130 131 Назначение элементов управления основной панели инструментов AllFusion Process Modeler  
<https://studfile.net/preview/7276093/page:2/>

104 116 Основы IDEF3 - Cfin.ru  
<https://www.cfin.ru/vernikov/idef/idef3.shtml>

105 IDEF3 - Википедия  
<https://ru.wikipedia.org/wiki/IDEF3>

117 Стандарт IDEF3 - Бизнес Консалтинг Групп  
<https://b-c-group.ru/bisness/idef3/>

119 Основы работы с AllFusion Process Modeler  
<https://pro-spo.ru/bisness-processing/2662--allfusion-process-modeler>

120 121 122 1.4. Разработка функциональной модели системы средствами AllFusion Process Modeler  
<https://studfile.net/preview/16404567/page:5/>

123 Функциональное моделирование с использованием продукта ...  
<https://mei06.narod.ru/sem9/theme8/9.htm>

137 ГОСТ 34.601-90 Информационная технология (ИТ). Комплекс ...  
<https://docs.cntd.ru/document/1200006921>

138 [PDF] ГОСТ 34. Разработка автоматизированной системы управления ...  
<https://mt-r.ru/upload/blok-skhemy/gost-34.pdf>

150 151 152 158 Эволюция методологий разработки / Хабр  
<https://habr.com/ru/articles/778502/>

153 Модели разработки ПО: минусы и плюсы | Лаборатория качества  
<https://qaschool.ru/blog/modeli-razrabotki-po-minusy-i-plyusy/>

154 Преимущества инкрементной модели жизненного цикла  
<https://studfile.net/preview/1444533/page:9/>

155 Инкрементная и спиральная модели - SCAND  
<https://scand.com/ru/company/blog/incremental-vs-spiral/>

<sup>156</sup> Инкрементная модель в SDLC: использование, преимущества и ...  
<https://www.guru99.com/ru/what-is-incremental-model-in-sdlc-advantages-disadvantages.html>

<sup>157</sup> Модели разработки и тестирования ПО: Инкрементная модель  
<https://bytextest.ru/2017/11/23/incremental-model/>