**Лекция 3. Требования к архитектуре информационной системы для обеспечения безопасности ее функционирования**

Архитектура программного обеспечения – это процесс преобразования таких характеристик программного обеспечения, как гибкость, масштабируемость, выполнимость, возможность повторного использования и безопасность, в структурированное решение, отвечающее техническим и бизнес-ожиданиям. Это определение заставляет нас задаться вопросом о характеристиках программного обеспечения, которые могут повлиять на проектирование архитектуры программного обеспечения.

Архитектура ПС определяет основные компоненты и подсистемы комплекса программ, как они соединяются между собой, и как могут быть достигнуты требуемые свойства. Основные компоненты ПС включают: операционные системы, базы данных, внутренние подсистемы ввода/вывода, коммуникационные подсистемы, прикладные программы, инструментальные средства программирования и диагностики и т.п. В крайнем случае, с использованием языков широкого применения, архитектура системы должна создаваться поставщиком специально для этого применения (или этого класса применений). С точки зрения безопасности создание архитектуры ПС является этапом, на котором разрабатывается стратегия основной безопасности комплекса программ и системы. Конструкция архитектуры программного средства должна устанавливаться поставщиком и/или разработчиком, при этом должно быть создано подробное описание конструкции. Описание должно быть основано на подразделении на компоненты/подсистемы, для каждой из которых предоставлена следующая информация:

* являются ли они новыми, существующими или собственными;
* проходили ли они ранее верификацию, и если да, условия проведения их верификации;
* относится ли каждая подсистема/компонент к обеспечению безопасности или нет;
* уровень соответствия ПС комплексу требований по безопасности для каждой подсистемы/компонента;
* определять все взаимодействия ПС и аппаратуры и подробно оценивать их значение;
* использовать для представления архитектуры систему обозначений, которая однозначно определяет или ограничивается однозначно определяемыми свойствами.

При создании нового программного продукта очень важно выбрать оптимальную архитектуру как можно раньше. Правильная архитектура поддерживает команду инженеров при создании продукта. Неправильный выбор архитектуры препятствует развитию и может привести к дорогостоящим переделкам в будущем.

Подход к разработке ориентированный на будущее архитектуры программного обеспечения помогает создавать более простые конструкции, быстрее внедрять новые функции и избегать серьезных изменений архитектуры по мере развития продуктов, он масштабируется вверх и вниз в зависимости от размера проекта (продукт, услуга или просто большая функция) и может быть как строгим, так и неформальным, в зависимости от инженерной культуры в организации.

Рассмотрим два способа, с помощью которых команда инженеров может разработать программную архитектуру для своего продукта: специальная или запланированная.

Специальная архитектура возникает, когда команда разработчиков просто продвигается вперед, а разработчики создают продукт в соответствии со своими собственными предположениями, мнениями и предпочтениями без особого плана. Это нормально для небольших проектов с коротким сроком службы и без планов дальнейшего развития, таких как веб-сайты маркетинговых кампаний. Плановая архитектура - наоборот. Это продукт процесса проектирования, которым руководила команда инженеров.

Проблема с первым случаем заключается в том, что он не гарантирует, что команда предложит архитектуру, оптимизированную для достижения реальных целей организации. Причина в том, что разработчики будут предлагать технологии и шаблоны проектирования на основе своего мнения, но у них не будет общего набора критериев для оценки доступных вариантов и выбора наиболее подходящих.

Напротив, последний случай побуждает команду сначала согласовать общий набор критериев, полученных из анализа целей организации и условий окружающей среды.

Кроме того, это снижает влияние субъективных факторов на результат и побуждает инженеров выходить за рамки своего текущего опыта и знаний в поисках оптимальных технических решений.

В результате команда увеличивает свои шансы на разработку архитектуры, оптимизированной для достижения целей организации, и снижает вероятность того, что им придется вносить дорогостоящие изменения в архитектуру в будущем.

**Подход к проектированию архитектуры на основе анализа нефункциональных требований**

Подход к архитектурному проектированию на основе анализа нефункциональных требований состоит из трех этапов:

* Выявление и анализ нефункциональных требований к продукту.
* Выбор соответствующих атрибутов качества программного обеспечения для продукта и постановка их целей.
* Выбор технологий и шаблонов проектирования, которые соответствовали бы целевым показателям соответствующих атрибутов качества для удовлетворения нефункциональных требований.

Выявив нефункциональные требования к продукту или услуге, инженеры могут выбрать соответствующие атрибуты качества, установить цели и разработать архитектуру программного обеспечения для их достижения.

Другими словами, программный продукт должен иметь определенные атрибуты качества, чтобы соответствовать определенным нефункциональным требованиям.

**Шаг 1**. Выявление и анализ нефункциональных требований

Менеджеры по продукту и бизнес-аналитики редко могут предоставить инженерам исчерпывающий список нефункциональных требований. Некоторые нефункциональные требования связаны с чисто техническими аспектами продукта, и команда разработчиков продукта может не знать о них.

В результате команда инженеров должна руководить этим процессом самостоятельно.

Работая с продуктом, дизайна и, возможно, маркетинга, поддержки, аналитики, юриспруденции и других отделов, инженеры должны подробно изучить, что должен делать новый продукт, чтобы определить ожидания, предположения и требования к тому, как новый продукт должен выполнять свои функции.

Обратите внимание, что это не обязательно должна быть серия официальных встреч.

Иногда достаточно пары быстрых чатов с представителями этих ведомств, чтобы уточнить детали и перейти к следующему этапу.

Давайте посмотрим на пример, когда перед командой стоит задача разработать мобильное приложение для потоковой передачи видеоконтента. Это оставляет много места для интерпретации. После разговора с командой разработчиков и вопросов, например:

Сколько клиентов вы ожидаете через 6-12 месяцев после запуска?

Как вы думаете, сколько клиентов будут использовать это одновременно?

Как долго будут видео?

Насколько это было бы плохо, если бы продукт не работал в течение часа?

На какой рынок вы планируете вывести его в первые 12 месяцев?

это может быть расширено до чего-то вроде «через 12 месяцев после запуска продукт должен быть достаточно масштабируемым для потоковой передачи видеоконтента 0,5–1 миллиона одновременных пользователей 24/7 по всему миру».

Это побудило бы разработчиков рассматривать только те шаблоны проектирования и технологии, которые позволяют разрабатывать высокомасштабируемую, высокодоступную и отказоустойчивую архитектуру.

Кроме того, отдел маркетинга может добавить к этому, что продукт действительно должен быть локализованным, поскольку он будет доступен по всему миру.

Юридическая группа может потребовать, чтобы продукт соответствовал законам различных юрисдикций по всему миру в отношении того, какой видеоконтент может и не может быть доступен для различных возрастных групп.

В результате этого шага мы определили пять атрибутов качества, потенциально важных для архитектуры продукта: масштабируемость, доступность, отказоустойчивость, локализуемость и соответствие.

Как было сказано выше, на данном этапе нет необходимости уточнять все возможные аспекты будущего поведения продукта. Инженеры должны быть прагматичными: как только основные нефункциональные требования были определены и проанализированы и команда достигла точки убывающей отдачи, возможно, пора переходить к следующему этапу.

**Шаг 2.** Выбор соответствующих атрибутов качества

Имея список качественных атрибутов из предыдущего шага, разработчики могут выбрать, для каких из них следует оптимизировать архитектуру продукта, и определить свои цели.

Разработчики также могут захотеть узнать, для каких качественных характеристик не нужно оптимизировать архитектуру продукта. Тогда они будут знать, какими атрибутами нужно пожертвовать, чтобы удовлетворить важные требования. Кандидаты на такие нерелевантные атрибуты либо не имеют связанных нефункциональных требований, либо имеют те, которые легко удовлетворить.

В приведенном выше примере команда инженеров может решить оптимизировать для:

* масштабируемость, поэтому продукт может обрабатывать 0,5–1 млн одновременных пользователей,
* доступность, то есть время безотказной работы, скажем, 99,995%,
* локализуемость, поэтому его контент, поддержка и маркетинговые материалы могут быть доступны на 10 самых популярных (20, 30 и т. д.) языках в мире.

Что касается оставшихся двух атрибутов качества, команда может решить, что для них нет необходимости оптимизировать архитектуру:

* отказоустойчивость, может быть достигнута вместе с доступностью,
* соблюдение требований, поскольку ограничения контента в зависимости от географического местоположения не оказывают значительного влияния на технологии и шаблоны архитектуры, которые будут использоваться в продукте.

Разработчикам удобно иметь исчерпывающий список возможных атрибутов качества, чтобы проверить, есть ли у каждого атрибута какие-либо связанные нефункциональные требования к продукту. Наличие такого списка значительно упрощает шаги 1 и 2.

Список из 31 атрибута качества, можно сгруппировать по восьми характеристикам, которые определены в международном стандарте ISO / IEC 25010 в разделе, который представляет модель качества программного продукта:

* функциональная пригодность: функциональная полнота, функциональная правильность, функциональная пригодность;
* эффективность производительности: поведение во времени, использование ресурсов, емкость;
* надежность: зрелость, доступность, отказоустойчивость, восстанавливаемость;
* удобство использования: уместность, узнаваемость, обучаемость, работоспособность, защита от ошибок пользователя, эстетика пользовательского интерфейса, доступность;
* безопасность: конфиденциальность, целостность, неотрекаемость, подотчетность, подлинность;
* совместимость: сосуществование, взаимодействие;
* ремонтопригодность: модульность, возможность повторного использования, анализируемость, модифицируемость, тестируемость;
* портативность: адаптируемость, возможность установки, заменяемость.

Есть и другие атрибуты качества. Некоторым проектам, организациям или сферам деятельности могут потребоваться уникальные атрибуты качества.

Как и на предыдущем шаге, нет необходимости фиксировать все возможные атрибуты качества. Только важные имеют значение. От четырех до семи соответствующих атрибутов качества может быть достаточно для эффективного создания и оценки вариантов дизайна.

**Шаг 3.** Принятие архитектурных решений

Имея список соответствующих атрибутов качества и их целевых показателей, группа инженеров может приступить к созданию вариантов архитектуры. Давайте рассмотрим некоторые очевидные варианты архитектуры высокого уровня, которые могут быть предложены для трех атрибутов качества, указанных в приведенном выше примере.

Масштабируемость: продукт может использовать CDN для более быстрой доставки как минимум статического контента своим пользователям; видео можно хранить в облачном хранилище, возможно, в нескольких регионах по всему миру, чтобы ускорить потоковую передачу; серверы приложений также могут потребоваться развернуть в нескольких регионах; базы данных должны быть хорошо масштабируемыми и должны поддерживать репликацию в другие регионы, чтобы избежать больших задержек при вызове с серверов приложений, расположенных на противоположной стороне мира; часть функциональных возможностей серверной части может быть реализована с помощью облачных функций.

- Доступность: возможно, потребуется развернуть продукт в нескольких зонах доступности в каждом регионе; стратегии развертывания должны позволять развертывание без простоев; в каждом регионе может потребоваться копия всех данных, чтобы иметь возможность работать независимо.

- Возможность локализации: продукт должен поддерживать интеграцию с несколькими способами оплаты, популярными в разных странах; он должен быть интегрирован с системой управления переводами UI; каждый элемент контента должен поддерживать несколько локализованных версий; продукт должен надежно идентифицировать местоположение пользователя, чтобы знать, какой контент там доступен, а какой нет.

Определив доступные варианты, команда может наконец выбрать те из них, которые должны наиболее эффективно достичь целевых показателей соответствующих атрибутов качества.

Хотя это не гарантирует, что архитектура продукта не изменится в будущем, это все же снижает вероятность того, что изменения будут значительными, и дает инженерам достаточно информации для уверенного продвижения вперед.

Подводя итог, определив нефункциональные требования и соответствующие атрибуты качества, группа инженеров продукта может сгенерировать варианты архитектуры и выбрать наиболее эффективные из них на основе анализа, а не личных предпочтений и мнений.

Это должно помочь им быстрее достичь целей организации и избежать значительных изменений архитектуры в будущем.

Обратите внимание, что этот подход не предназначен для создания модели Big Design Up Front или водопада. Цель здесь - быстро определить объективные критерии, которым архитектура должна соответствовать в обозримом будущем, а не ограничивать дизайн и заменять реальную разработку продукта чертежами диаграмм и написанием документов.

Подход также довольно хорошо масштабируется вверх и вниз, поэтому он может быть настолько строгим или неформальным, насколько этого требует конкретный продукт. Я использовал его при разработке новых продуктов, новых систем и даже просто больших наборов функций.

При разработке архитектуры и создании инфраструктуры корпоративной ИС неизбежно встает вопрос о ее защищенности от угроз. Решение вопроса состоит в подробном анализе таких взаимно пересекающихся видов работ, как реализация ИС и аттестация, аудит и обследование безопасности ИС.

Основой формального описания систем защиты традиционно считается модель системы защиты с полным перекрытием, в которой рассматривается взаимодействие области угроз, защищаемой области и системы защиты. Таким образом, модель может быть представлена в виде трех множеств: множество угроз безопасности, множество объектов (ресурсов) защищенной системы, множество механизмов безопасности. Элементы этих множеств находятся между собой в определенных отношениях, собственно и представляющих систему защиты. Для описания системы защиты обычно используется графовая модель. Но формальные подходы к решению задачи оценки защищенности из-за трудностей, связанных с формализацией, широкого практического распространения не получили. Значительно более действенным является использование неформальных классификационных подходов. Для этого применяют категорирование: нарушителей (по целям, квалификации и доступным вычислительным ресурсам); информации (по уровням критичности и конфиденциальности); средств защиты (по функциональности и гарантированности реализуемых возможностей); эффективности и рентабельности средств защиты и т.п.

Идеология открытых систем существенно отразилась на методологических аспектах и направлении развития сложных ИС. Она базируется на строгом соблюдении совокупности профилей, протоколов и стандартов де-факто и де-юре. Программные и аппаратные компоненты по этой идеологии должны отвечать важнейшим требованиям переносимости и возможности согласованной, совместной работы с другими удаленными компонентами. Это позволяет обеспечить совместимость компонент различных информационных систем, а также средств передачи данных.

При создании сложных, распределенных информационных систем, проектировании их архитектуры, инфраструктуры, выборе компонент и связей между ними следует учитывать помимо общих (открытость, масштабируемость, переносимость, мобильность, защита инвестиций и т.п.) ряд специфических концептуальных требований, направленных на обеспечение безопасности функционирования:

• архитектура системы должна быть достаточно гибкой, т.е. должна допускать относительно простое, без коренных структурных изменений, развитие инфраструктуры и изменение конфигурации используемых средств, наращивание функций и ресурсов ИС в соответствии с расширением сфер и задач ее применения;

• должны быть обеспечены безопасность функционирования системы при различных видах угроз и надежная защита данных от ошибок проектирования, разрушения или потери информации, а также авторизация пользователей, управление рабочей загрузкой, резервированием данных и вычислительных ресурсов, максимально быстрым восстановлением функционирования ИС;

• следует обеспечить комфортный, максимально упрощенный доступ пользователей к сервисам и результатам функционирования ИС на основе современных графических средств, мнемосхем и наглядных пользовательских интерфейсов;

• систему должна сопровождать актуализированная, комплектная документация, обеспечивающая квалифицированную эксплуатацию и возможность развития ИС.

Подчеркнем, что системы безопасности, какими бы мощными они ни были, сами по себе не могут гарантировать надежность программно-технического уровня защиты. Только проверенная архитектура способна сделать эффективным объединение сервисов, обеспечить управляемость информационной системы, ее способность развиваться и противостоять новым угрозам при сохранении таких свойств. как высокая производительность, простота и удобство использования.

С практической точки зрения обеспечения безопасности наиболее важными являются следующие принципы построения архитектуры ИС:

• проектирование ИС на принципах открытых систем, следование признанным стандартам, использование апробированных решений, иерархическая организация ИС с небольшим числом сущностей на каждом уровне — все это способствует прозрачности и хорошей управляемости ИС;

• непрерывность защиты в пространстве и времени, невозможность преодолеть защитные средства, исключение спонтанного или вызванного перехода в небезопасное состояние — при любых обстоятельствах, в том числе нештатных, защитное средство либо полностью выполняет свои функции, либо полностью блокирует доступ в систему или ее часть;

• усиление самого слабого звена, минимизация привилегий доступа, разделение функций обслуживающих сервисов и обязанностей персонала. Предполагается такое распределение ролей и ответственности, чтобы один человек не мог нарушить критически важный для организации процесс или создать брешь в защите по неведению или заказу злоумышленников. Применительно к программно-техническому уровню принцип минимизации привилегий предписывает выделять пользователям и администраторам только те права доступа, которые необходимы им для выполнения служебных обязанностей. Это позволяет уменьшить ущерб от случайных или умышленных некорректных действий пользователей и администраторов;

• эшелонирование обороны, разнообразие защитных средств, простота и управляемость информационной системы и системой ее безопасности. Принцип эшелонирования обороны предписывает не полагаться на один защитный рубеж, каким бы надежным он ни казался. За средствами физической защиты должны следовать программно-технические средства, за идентификацией и аутентификацией — управление доступом, протоколирование и аудит. Эшелонированная оборона способна не только не пропустить злоумышленника, но и в некоторых случаях идентифицировать его благодаря протоколированию и аудиту. Принцип разнообразия защитных средств предполагает создание различных по своему характеру оборонительных рубежей, чтобы от потенциального злоумышленника требовалось овладение разнообразными и, по возможности, несовместимыми между собой навыками.

• очень важен общий принцип простоты и управляемости ИС в целом и защитных средств в особенности. Только в простой и управляемой системе можно проверить согласованность конфигурации различных компонентов и осуществлять централизованное администрирование. В этой связи важно отметить интегрирующую роль Web-сервиса, скрывающего разнообразие обслуживаемых объектов и предоставляющего единый, наглядный интерфейс. Соответственно, если объекты некоторого вида (например, таблицы базы данных) доступны через Интернет, необходимо заблокировать прямой доступ к ним, поскольку в противном случае система будет уязвимой, сложной и плохо управляемой.

Анализ безопасности ИС при отсутствии злоумышленных факторов базируется на модели взаимодействия основных компонент ИС. В качестве объектов уязвимости рассматриваются:

- динамический вычислительный процесс обработки данных, автоматизированной подготовки решений и выработки управляющих воздействий;

- объектный код программ, исполняемых вычислительными средствами в процессе функционирования ИС;

-  данные и информация, накопленная в базах данных;

-  информация, выдаваемая потребителям и на исполнительные механизмы.

Полное устранение перечисленных угроз принципиально невозможно. Задача состоит в выявлении факторов, от которых они зависят, в создании методов и средств уменьшения их влияния на безопасность ИС, а также в эффективном распределении ресурсов для обеспечения защиты, равнопрочной по отношению ко всем негативным воздействиям.



Рис. 3. Модель анализа безопасности ИС при отсутствии злоумышленных угроз

**Инструментальные средства поддержки разработки систем ПО**

Выбор инструментальных средств разработки зависит от характера деятельности по разработке и от архитектуры ПС. Для требуемого уровня безопасности должен быть выбран соответствующий комплект технологических средств, включая языки, компиляторы, инструментальные средства управления конфигурацией, и при необходимости, автоматические испытательные средства. Нужно рассмотреть наличие подходящих инструментальных средств (не обязательно таких, какие применялись при первоначальной разработке системы) для обеспечения необходимых действий на протяжении всего срока службы системы безопасности.

Инструментальные средства разработки ПО– это совокупность аппаратно-программных средств, позволяющих осуществить написание и отладку программ с большой степенью достоверности их работоспособности.

Любая развитая технологическая система должна поддерживать все основные этапы создания проектируемого программного комплекса. Для достижения этой цели в общей структуре типовой технологической системы поддержки разработки (рис. 4) обычно выделяют базу данных проекта; подсистему автоматизации проектирования и программирования; подсистемы отладки, документирования и сопровождения, а также подсистему управления ходом выполнения проекта.

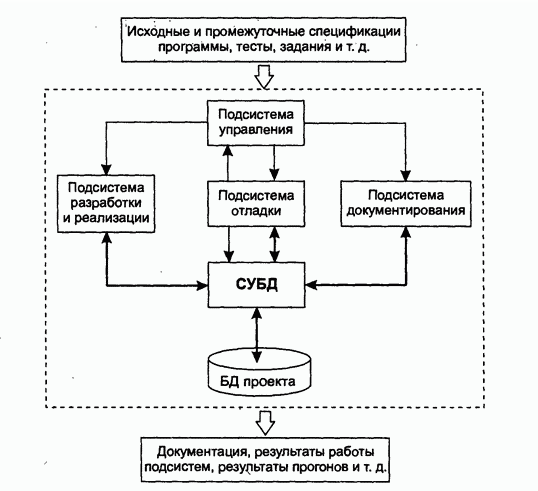


Рис. 4. Общая структура типовой технологической системы поддержки разработки

Развитые библиотечные системы поддержки разработки используются в настоящее время во всем мире во всех сколько-нибудь серьезных программных проектах. Но в подавляющем большинстве случаев такие системы достигли уровня удобства работы с ними квалифицированных программистов. Нас же, прежде всего, интересуют системы и проекты, в которых имеются тенденции к эксплицитному представлению технологических знаний, даже если они и не базируются на идеях и методах ИИ.

Один из таких проектов — Gandalf — ориентирован на автоматизированную генерацию систем разработки программного обеспечения. Исследования, выполняемые в рамках проекта Gandalf, касаются трех аспектов поддержки проектирования ПО: управление проектом, контроль версий и инкрементное программирование, а также интеграция их в единую среду. Управление в Gandalf-среде базируется на предположении, что разрабатываемый проект должен трактоваться как множество абстрактных типов данных, над которыми могут выполняться лишь определенные операции. Средством, реализующим данную концепцию, явилась система SDC (Software Development Control), представляющая собой набор программ, первоначально реализованных на языке Shell в системе UNIX, а позднее переведенная на язык С.

Исследования в области контроля версий были начаты еще Л. Коопридером на базе проекта FAFOS [Habermann et al., 1976], где изначально анализировались возможности создания семейства операционных систем. Была разработана нотация для описания взаимодействия между подсистемами, для описания различных версий подсистем (исходного и объектного кода, документации и т. п.) и для описания действующих на этапе разработки механизмов (компиляция, редактирование связей и т. п.). Затем был создан специальный язык Intercol как средство описания взаимосвязи и версий модулей в системе. И наконец, в систему были встроены знания о том, как конструировать систему из частей, не заставляя заниматься этим пользователя. В развитие этих работ была создана система SUCE, в рамках которой отслеживались различия между реализациями (версиями, которые действительно дают код для ряда спецификаций) и композициями (версиями, определяющими новые подсистемы как группы существующих подсистем). В системе LOIPE (Language-Oriented Incremental Programming Environment) инкрементная компиляция выполняется на уровне отдельной процедуры. Достоинством такого подхода является то, что при коррекции процедуры на уровне локальных объектов или типов перекомпилируется только она. Если же меняется спецификация, то перекомпилируются и все зависящие от нее процедуры. Пользовательский интерфейс с LOIPE-системой базируется на подсистеме синтакси-чески-ориентированного редактирования ALOE (A Language-Oriented Editor). Целью разработки этой подсистемы было исследование возможности создания и использования синтаксически-ориентированных редакторов в качестве базиса для сред программирования.

Анализ литературы последних лет по технологии программирования показывает, что новой ветвью в технологии промышленной разработки и реализации сложных и значительных по объему систем программного обеспечения является CASE-технология (Computer Aided Software Engineering) [1990].

Первоначально CASE-технология появилась в проектах создания промышленных систем обработки данных. Это обстоятельство наложило свой отпечаток и на инструментальные средства CASE-технологии, где самое серьезное внимание уделялось, по крайней мере в ранних CASE-системах, поддержке проектирования информационных потоков. В настоящее время наблюдается отход от ориентации на системы обработки данных, и инструментальные средства CASE-технологии становятся все более универсальными.

Все средства поддержки CASE-технологии делятся на две большие группы: САSE-ToolKits и CASE-WorkBenches. Хороших русских эквивалентов этим терминам нет. Однако первые часто называют «инструментальными сундучками» (пакетами разработчика, технологическими пакетами), а вторые — «станками для производства программ (технологическими линиями).

CASE-ToolKit — коллекция интегрированных программных средств, обеспечивающих автоматическое ассистирование в решении задач одного типа в процессе создания программ.

Такие пакеты используют общее «хранилище» для всей технической и управляющей информации по проекту (репозиторий), снабжены общим интерфейсом с пользователем и унифицированным интерфейсом между отдельными инструментами пакета. Как правило, CASE-ToolKit концентрируются вокруг поддержки разработки одной фазы производства программ или на одном типе прикладных задач.

Все вышесказанное справедливо и по отношению к CASE-WorkBench. Но здесь, кроме того, обеспечивается автоматизированная поддержка анализа решаемых задач по производству программного обеспечения, которая базируется на общих предположениях о процессе и технологии такой деятельности; поддерживается автоматическая передача результатов работ от одного этапа к другому, начиная со стадии проектирования и кончая отчуждением созданного программного продукта и его сопровождением.

Таким образом, CASE-WorkBench является естественным «замыканием» технологии разработки, реализации и сопровождения программного обеспечения.

В настоящее время «типовая» система поддержки CASE-технологии имеет функциональные возможности, представленные на рис. 5.

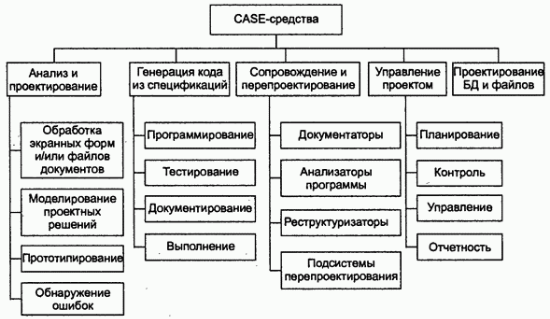


Рис. 5. Функциональные возможности типовой системы поддержки CASE-технологии

Как следует из этой Н-диаграммы, в CASE-среде должны поддерживаться все основные этапы разработки и сопровождения процессов создания программных систем. Однако уровень такой поддержки существенно различен. Так, например, если говорить об этапах анализа и проектирования, большинство инструментальных пакетов поддерживает экранные и отчетные формы, создание прототипов, обнаружение ошибок. Значительная часть этих средств предназначена для ПЭВМ. Многие поддерживают такие широко используемые методологии, как структурный анализ DeMarco или Gane/Sarson, структурное проектирование Yourdan/Jackson и некоторые другие. Существуют специализированные пакеты разработчиков для создания информационных систем, например AnaTool (Advanced Logical Software) для Macintosh; CA-Universe/Prototype (Computer Associates International) для ПЭВМ. Имеются CASE-среды и для поддержки разработки систем реального времени.

В среде разработчиков ПО существуют две оценки данного подхода: часть из них считает, что CASE-технология кардинально меняет процессы разработки и эксплуатации ПО, другие отрицают это и оставляют за инструментальными средствами CASE лишь функцию автоматизации рутинных работ. Однако анализ литературы показывает, что CASE средства все-таки «сдвигают» технологии разработки ПО с управления выполнением проектов в сторону метода прототипизации. И этот сдвиг, на наш взгляд, чрезвычайно важная тенденция в современной технологии программирования.

Классификация по признакам

1. Рассмотрим основные классификации современных CASE-систем по следующим признакам: Поддерживаемые методологии проектирования: объектно-ориентированные, функционально (или структурно)-ориентированные и комплексно-ориентированные;
2. Поддерживаемые графические нотации построения диаграмм: с наиболее распространенными нотациями, с отдельными нотациями и с фиксированной нотацией;
3. Степень интегрированности: toolkit (неинтегрированные средства, которые охватывают большинство этапов разработки информационных систем), tools (отдельные локальные средства) и workbench (интегрированные средства, которые связаны репозиторием – общей базой проектных данных);
4. Тип и архитектура вычислительной техники: с ориентацией на глобальную вычислительную сеть (ГВС), на локальную вычислительную сеть (ЛВС), на ПЭВМ и смешанный тип;
5. Режим коллективной разработки проекта: с ориентацией на режим объединения подпроектов, режим реального времени разработки и без поддержки коллективной разработки;
6. Тип операционной системы: работающие под управлением UNIX, под управлением WINDOWS и под управлением разных операционных систем (OS/2, UNIX, WINDOWS и др.)

Классификация по типам

1. Средства проектирования и анализа, которые предназначены для анализа и построения моделей системы, которая проектируется, и моделей деятельности организации (предметной области). К ним относят System Architect, Power Designer, Paradigm Plus, Rational Rose, Oracle Designer, Silverrun, BPwin. Цель данных средств – определить системные требования и свойства, которыми должна обладать система, а также создать проект системы, которая будет удовлетворять этим требованиям и обладать соответствующими свойствами. Продуктом таких средств являются спецификации алгоритмов, спецификации компонентов системы и их интерфейсов, структур данных.
2. Средства проектирования БД, которые обеспечивают генерацию схем БД и моделирование данных (обычно на языке SQL) для наиболее распространенных СУБД. Средства проектирования баз данных входят в состав следующих CASE-средств: Power Designer, Paradigm Plus, Oracle Designer, Silverrun. Наиболее известное средство, которое ориентировано только на проектирование баз данных, – ERwin.
3. Средства управления требованиями, которые обеспечивают комплексную поддержку неоднородных требований к системе, которая создается. Примеры: DOORS – динамическая объектно-ориентированная система управления требованиями и RequisitePro.
4. Средства тестирования. Наиболее развитое сегодня – Rational Suite TestStudio – набор продуктов, которые предназначены для автоматического тестирования приложений.
5. Средства управления конфигурацией программного обеспечения – ClearCase, PVCS и др.
6. Средства документирования. Наиболее известное из них – SoDA (автоматизированное документирование программное обеспечение).
7. Средства управления проектом – Microsoft Project, Open Plan Professional и др.
8. Средства реверсного инжиниринга, которые предназначены для переноса существующей системы программного обеспечения в новую среду. Ими обеспечивается анализ схем баз данных и программных кодов и формируются на их основе различные модели и проектные спецификации.

Классификация по категориям

Классификация CASE-средств по категориям показывает широту охвата процессов разработки программного обеспечения, которые поддерживаются данным средством:

1. Вспомогательные программы (tools) – поддерживаются отдельные процессы разработки программного обеспечения (например, сравнение результатов тестов, компиляция программ, проверка непротиворечивости архитектуры системы и т.п.). Вспомогательная программа может быть универсальным функционально-законченным средством (например, текстовый процессор) или быть составляющей инструментальных средств.
2. Инструментальные средства (workbenches) – поддерживаются определенные процессы разработки программного обеспечения (к примеру, проектирование, создание спецификации и т.д.). Зачастую инструментальные средства представляют собой набор вспомогательных программ, интегрированных в меньшей или большей степени.
3. Рабочие среды разработчика (environments) – поддерживаются большинство или все процессы разработки программного обеспечения. Рабочие среды зачастую содержат несколько разных интегрированных инструментальных средств.

Кроме того, CASE-средства также классифицируют по применяемым объектно-ориентированным или структурным методам проектирования и анализа программного обеспечения.