Программная инженерия

Проектирование программного обеспечения (Software Design)

Глава базируется на IEEE Guide to the Software Engineering Body of Knowledge - SWEBOK®, 2004. Содержит перевод описания области знаний SWEBOK® “Software Design”, с комментариями и замечаниями.

Официальный сайт “Основ программной инженерии” (по SWEBOK) - http://swebok.sorlik.ru

Программная инженерия

Проектирование программного обеспечения (Software Design)

Процесс определения архитектуры, компонентов, интерфейсов и других характеристик системы или ее компонентов называется проектированием. Результат процесса проектирования – дизайн. Рассматриваемое как процесс, проектирование есть инженерная деятельность в рамках жизненного цикла (в данном контексте – программного обеспечения), в которой надлежащим образом анализируются требования для создания описания внутренней структуры ПО, являющейся основой для конструирования программного обеспечения как такового. Программный дизайн (как результат деятельности по проектированию) должен описывать архитектуру программного обеспечения, то есть представлять декомпозицию программной системы в виде организованной структуры компонент и интерфейсов между компонентами. Важнейшей характеристикой готовности дизайна является тот уровень детализации компонентов, который позволяет заняться их конструированием. Термины дизайн и архитектура могут использоваться взаимозаменяемым образом, но чаще говорят о дизайне как о целостном взгляде на архитектуру системы.

Проектирование играет важную роль в процессах жизненного цикла создания программного обеспечения (Software Development Life Cycle), например, IEEE и ISO/IEC (ГОСТ Р ИСО.МЭК) 12207.

Проектирование программных систем можно рассматривать как деятельность, результат которой состоит из двух составных частей:

- Архитектурный или высокоуровневый дизайн (software architectural design, top-level design) – описание высокоуровневой структуры и организации компонентов системы;
- Детализированная архитектура (software detailed design) – описывающая каждый компонент в том объеме, который необходим для конструирования.

В результате консенсуса, принятого создателями SWEBOK, данная область знаний не описывает все сущности или понятия, имеющие в своем названии слово “дизайн” или “архитектура”. В 1999 году Том ДеМарко (Tom DeMarco) [DeMarco, 1999], один из известных специалистов в программной инженерии, предложил терминологическое разделение различных видов дизайна:

- *D-дизайн (D-design, decomposition design)* – декомпозиция структуры программного обеспечения в виде набора фрагментов или компонент;
- *FP-дизайн (FP-design, family pattern design)* – семейство архитектурных представлений, базирующихся на шаблонах;
- *I-дизайн (I-design, invention)* – создание высоко-уровневой концепции, видения того, что из себя будет представлять программная система; данный вид дизайна является результатом процесса анализа требований и их трансформации в подходы к реализации.

Если обсуждать данную область знаний в терминах ДеМарко, проектирование программного обеспечения в понимании программной инженерии подразумевает D- и FP-дизайн. I-дизайн в большей степени относится к работе с программными требованиями.

Соответственно, данная область знаний тесно связана со следующими областями программной инженерии:

- Software Requirements
- Software Construction
- Software Maintenance
- Software Engineering Management
- Software Quality

Сама же область знаний по проектированию программного обеспечения представлена в виде 6 секций, структурированных по темам.

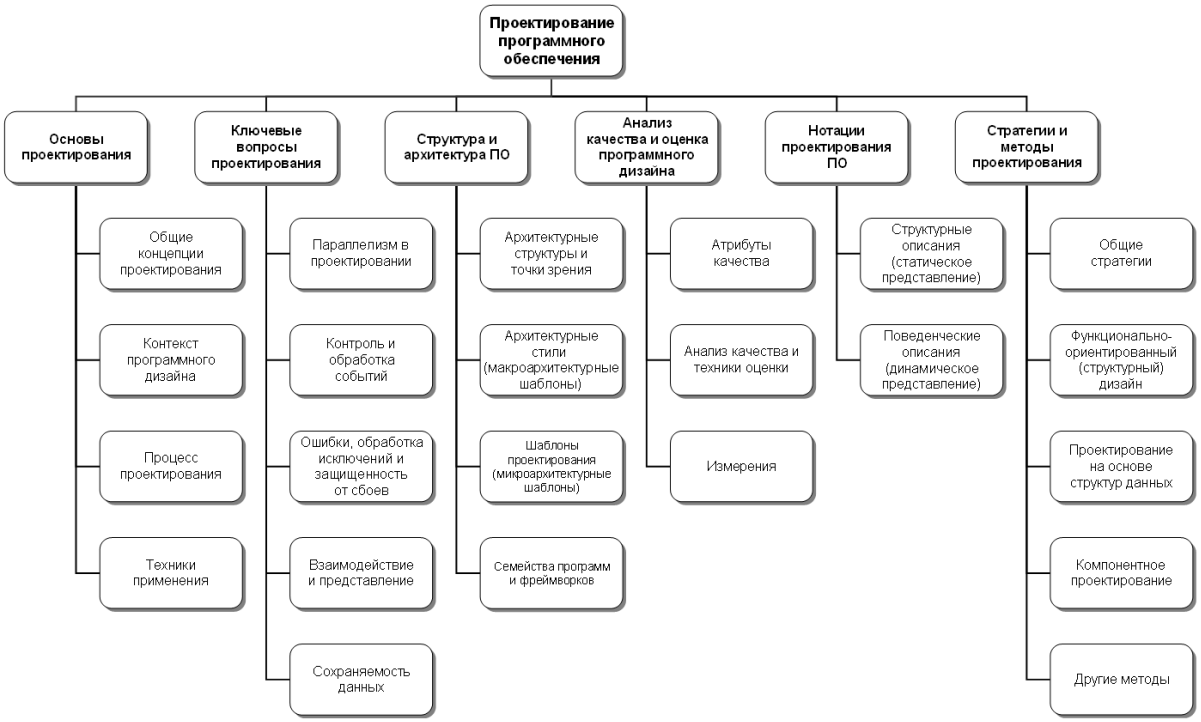

Рисунок 3. Область знаний “Проектирование программного обеспечения” [SWEBOK, 2004, с.3-2, рис. 1]

1. Основы проектирования (Software Design Fundamentals)

Эта секция вводит концепции, понятия и терминологию в качестве основы для понимания роли и содержания проектирования (как деятельности) и дизайна (архитектуры, как результата) программного обеспечения.

Темы данной секции:

1.1 Общие концепции проектирования (General Design Concepts)

К ним относятся: цель архитектуры, ее ограничения, возможные альтернативы, используемые представления и решения.

Например, архитектурный фреймворк – TOGAF [TOGAF, 2003], разработанный и развиваемый консорциумом The Open Group (www.opengroup.org), предлагает следующие <возможные> цели (goals):

- Улучшение и повышение продуктивности бизнес-процессов
- Уменьшение затрат
- Улучшение операционной бизнес-деятельности
- Повышение эффективности управления
- Уменьшение рисков
- Повышение эффективности ИТ-организации
- Повышение продуктивности работы пользователей
- Повышение интероперабельности (возможности и прозрачности взаимодействия)
- Уменьшение стоимости <поддержки> жизненного цикла
- Улучшение характеристик безопасности
- Повышение управляемости

1.2 Контекст проектирования (Context of Software Design)

Для понимания роли проектирования программного обеспечения важно понимать контекст, в котором осуществляется проектирование и используются его результаты. В качестве такого контекста выступает жизненный цикл программной инженерии, а проектирование напрямую связано с результатами анализа требований, конструированием программных систем и их тестированием. Стандарты жизненного цикла, например, IEEE и ISO/IEC (ГОСТ Р) 12207 уделяют специальное внимание вопросам проектирования и детализируют их, описывая контекст проектирования – от требований до тестов.

1.3 Процесс проектирования (Software Design Process)

Проектирование в основном рассматривается как двух-шаговый процесс:

1.3.1 Архитектурное проектирование – декомпозиция структуры (статической) и организации (динамической) компонент;

1.3.2 Детализация архитектуры – описывает специфическое поведение и характеристики отдельных компонент.

Выходом этого процесса является набор моделей и артефактов, содержащих результаты решений, принятых по способам реализации требований в программном коде.

1.4 Техники применения (Enabling Techniques)

Принципы проектирования, также называемые техниками применения, являются ключевыми идеями и концепциями, рассматриваемыми на фундаментальном уровне в различных методах и подходах к проектированию программного обеспечения.

1.4.1 Абстракция (Abstraction)

В контексте проектирования программных систем существует два механизма абстракции – параметризация и специфицирование (может интерпретироваться как детализация). При этом, абстракция через специфицирование бывает трех видов: процедурная абстракция (динамическая, то есть в отношении поведения), абстракция данных (статическая, то есть в отношении информации) и абстракция контроля (то есть управления системой и обрабатываемой ею информацией).

Обычно под абстракций, как результатом процесса абстракции, понимают модель, упрощающую поставленную проблему до рамок, значимых для заданного контекста.

1.4.2 Связанность и соединение (Coupling and Cohesion)

Связанность (Coupling) – определяет силу связи (часто, взаимного влияния) между модулями.
Соединение (Cohesion) – определяет как тот или иной элемент обеспечивает связь внутри модуля, внутреннюю связь.

Значение оригинальных терминов очень близко и, в зависимости от контекста, “связанность” и “соединение” могут рассматриваться как степень самодостаточности или ее отсутствия (coupling) и функциональная зависимость (cohesion) , соответственно.

Хочется особенно подчеркнуть значимость этих понятий, так как с развитием сервисно-ориентированной архитектуры (Service-Oriented Architecture, SOA), слабосвязанной по своей природе (то есть со слабым “сопряжением”, слабой “силой связи” между модулями), по сравнению, например, с OMG CORBA (Common Object Request Broker Architecture), все чаще приходится сравнивать различные подходы и решения, определяемые способом и степенью связанности различных модулей, компонент и самих программных систем.

1.4.3 Декомпозиция и разбиение на модули (Decomposition and Modularization)

Декомпозиция и разбиение на модули сложных программных систем производится с целью получения более мелких и относительно независимых программных компонентов, каждый из которых несет различную функциональность (логически связанные группы функциональности).

1.4.4 Инкапсуляция/сокрытие информации (Encapsulation/information hiding)

Данная концепция предполагает группировку и упаковку (с точки зрения подготовки к развертыванию и эксплуатации) элементов и внутренних деталей абстракции (то есть модели) в отношении реализации с тем, чтобы эти детали (как малозначимые для использования компонента или по другим причинам) были недоступны пользователям элементов (компонент). При этом, в качестве “пользователя” одного компонента может выступать другой компонент. Более того, при использовании объектно-ориентированного подхода, наследники компонентов могут не иметь доступа ко внутренним деталям реализации компонента, который является их предком (зависит от объектно-ориентированной модели конкретного языка программирования или платформы).

1.4.5 Разделение интерфейса и реализации (Separation of interface and implementation)

Данная техника предполагает определение компонента через специфицирование интерфейса, известного (описанного) и доступного клиентам (или другим компонентам), от непосредственных деталей реализации.

1.4.6 Достаточность, полнота и простота (Sufficiency, completeness and primitiviness)

Этот подход подразумевает, что создаваемые программные компоненты обладают всеми необходимыми характеристиками, определенными абстракцией (моделью), но не более того. То есть не включают функциональность, отсутствующую в модели.

Данный принцип особенно ярко выделен и явно представлен в виде рекомендуемых практик (best practices) методологий гибкого моделирования и экстремального программирования, где “все, что надо, но ни граммом больше” лежит в основе самой концепции “прагматичного” подхода (и на стадии

моделирования, и в отношении реализации в коде). В оригинале этот принцип звучит как YAGNI – “You Aren’ t Going to Need It”, то есть “не делай этого, пока не понадобится”.

2. Ключевые вопросы проектирования (Key Issues in Software Design)

В какой-то мере, данную секцию стоило перевести как ключевые проблемы. Как проводить декомпозицию? Как организовать и объединить компоненты в единую систему? Как обеспечить необходимую производительность? Наконец, как обеспечить приемлемое качество системы? Все это – фундаментальные вопросы и проблемы проектирования, вне зависимости от используемых при проектировании подходов.

2.1 Параллелизм (Concurrency)

Эта тема охватывает вопросы, подходы и методы организации процессов, задач и потоков для обеспечения эффективности, атомарности, синхронизации и распределения (по времени) обработки информации.

2.2 Контроль и обработка событий (Control and Handling of Events)

В самом названии данной темы заложен комплекс обсуждаемых вопросов. В частности, данная тема касается и неявных методов обработки событий, часто реализуемых в виде функции обратного вызова (call-back), как одной из фундаментальных концепций обработки событий.

2.3 Распределение компонентов (Distribution of Components)

Распределение (и выполнение) по различным узлам обработки в терминах аппаратного обеспечения, сетевой инфраструктуры и т.п. Один из важнейших вопросов данной темы – использование связующего программного обеспечения (middleware*)

* часто middleware переводят как “промежуточное программное обеспечение”. Такой вариант перевода, к сожалению, рассматривает связующее ПО во второстепенной – “промежуточной” роли. Читатель, безусловно, может не согласиться с такой трактовкой, однако, многолетняя практика автора в обсуждении архитектурных вопросов с различными специалистами демонстрирует именно такой взгляд пользователей, не знакомых или не имеющих успешного опыта разработки и эксплуатации распределённых систем.

2.4 Обработка ошибок и исключительных ситуаций и обеспечение отказоустойчивости (Errors and Exception Handling and Fault Tolerance)

Вопрос темы, как ни странно, формулируется достаточно просто – как предотвратить сбои или, если сбой все же произошел, обеспечить дальнейшее функционирование системы.

2.5 Взаимодействие и представление (Interaction and Presentation)

Тема касается вопросов представления информации пользователям и взаимодействия пользователей с системой, с точки зрения реакции системы на действия пользователей. Речь в этой теме идет о реакции системы в ответ на действия пользователей и организации ее отклика с точки зрения внутренней организации взаимодействия, например, в рамках популярной концепции Model-View-Controller.

Ни в коем случае не надо путать данную тему с вопросами организации пользовательского интерфейса, являющимеся частью “Эргономики программного обеспечения” – Software Ergonomics (см. “Связанные дисциплины”).

2.6 Сохраняемость данных (Data Persistence)

Именно сохраняемость, а не сохранность, так как тема касается не доступа к базам данных, как такового, а также не гарантий сохранности информации. Суть вопроса – как должны обрабатываться “долгоживущие” данные.

3. Структура и архитектура программного обеспечения (Software Structure and Architecture)

В строгом значении архитектура программного обеспечения (software architecture) – описание подсистем и компонент программной системы, а также связей между ними. Архитектура пытается определить внутреннюю структуру получаемой системы, задавая способ, которым система организована или конструируется.

В середине 90-х, на волне распространения клиент-серверного подхода и начала его трансформации в “многозвенный клиент-сервер”, призванный обеспечить централизованное развертывание и управление общей (для клиентских приложений) бизнес-логикой, вопросы организации архитектуры программного обеспечения стали складываться в самостоятельную и достаточно обширную дисциплину. В результате, сформировалась точка зрения на архитектуру не только в приложении к конкретной программной системе, но и развился взгляд на архитектуру, как на приложение общих (generic) принципов организации программных компонент. В итоге, уже на сегодняшний день, на фоне такого развития понимания архитектуры, накоплен целый комплекс подходов и созданы (и продолжают создаваться и развиваться !) различные архитектурные “фреймворки”, то есть систематизированные комплексы методов, практик и инструментов, призванные в той или иной степени формализовать имеющийся в индустрии опыт (как положительный – например, design patterns, так и отрицательный – например, anti-patterns).

Примеры такой систематизации в форме фреймворков:

- TOGAF [TOGAF81, 2003] – The Open Group Architecture Framework (на момент написания данной главы доступен в версии 8.1, впервые опубликованной в декабре 2003 года)
- Модель Захмана – Zachman Framework [Zachman]
- Руководство по архитектуре электронного правительства E-Gov Enterprise Architecture Guidance [E-Gov, 2002]

3.1 Архитектурные структуры и точки зрения (Architectural Structures and Viewpoints)

Любая система может рассматриваться с разных точек зрения – например, поведенческой (динамической), структурной (статической), логической (удовлетворение функциональным требованиям), физической (распределенность), реализации (как детали архитектуры представляются в коде) и т.п. В результате, мы получаем различные архитектурные представления (view). Архитектурное представление может быть определено, как частные аспекты программной архитектуры, рассматривающие специфические свойства программной системы. В свою очередь, дизайн системы – комплекс архитектурных представлений, достаточный для реализации системы и удовлетворения требований, предъявляемых к системе.

SWEBOK не дает явного определения, что такое “архитектурная структура”. В то же время это понятие достаточно важно. Я хотел бы предложить его толкование как применение архитектурной точки зрения и представления к конкретной системе и описания тех деталей, которые необходимы для реализации системы, но отсутствуют (в силу достаточно общего взгляда) в используемом представлении. Таким образом, представление (view), концентрируясь на заданном подмножестве свойств является составной частью и/или результатом точки зрения, а архитектурная структура – дальнейшей детализацией в отношении проектируемой системы.

Модель Захмана [Zachman] является великолепным и, кстати, классическим источником комплекса архитектурных точек зрения и представлений, построенных в системе координат “вопрос-уровень детализации”. Каждое архитектурное представление является результатом ответа на вопрос (Как? Что? Где? и т.п.) в контексте необходимого уровня абстракции (содержание, то есть концепция: бизнес-модель, то есть функциональность и т.д.). Например, физическая модель данных (Physical Data Model) является ответом на вопрос “что?” в контексте технологической модели, а логическая

модель данных, отвечая на тот же вопрос, находится на один уровень абстракции выше – в контексте системной или логической модели.

3.2 Архитектурные стили (Architectural Styles)

В рассматриваемой редакции SWEBOK допущено несоответствие между структурой декомпозиции данной области знаний и описанием охватываемых ею тем. Если архитектурные стили присутствуют в декомпозиции, в самом описании области знаний темы 3.1 и 3.2 смешаны (по форматированию и структуре) в рамках темы “3.1”, (о чем автор сообщил ассоциированному редактору данной части SWEBOK).

Архитектурный стиль, по своей сути, шаблон проектирования макро-архитектуры - на уровне модулей, "крупноблочного" взгляда. Например, архитектура распределенной сервисно-ориентированной системы может строится в стиле обмена сообщениями через соответствующие очереди сообщений, может проектироваться на основе идеи взаимодействия между компонентами и приложениями через общую объектную шину, а может использовать концепцию брокера как единого узла пересылки запросов. В то же время, на более концептуальном уровне, мы можем говорить о выборе клиент-серверного стиля или распределенного стиля архитектуры системы. Таким образом, архитектурный стиль – набор ограничений, определяющих семейство архитектур, которые удовлетворяют этим ограничениям.

3.3 Шаблоны проектирования (Design Patterns)

Наиболее краткая формулировка того, что такое шаблон проектирования, может звучать так – “общее решение общей проблемы в заданном контексте”. Что это значит в реальной жизни? Если мы хотим организовать системы таким образом, чтобы существовал один и только один экземпляр заданного ее компонента в процессе работы с данной системой – мы можем использовать шаблон проектирования “Singleton”, описывающий такое общее поведение.

В то время, как архитектурный стиль определяет макро-архитектуру системы, шаблоны проектирования задают микро-архитектуру, то есть определяют частные аспекты деталей архитектуры.

Чаще всего говорят о следующих группах шаблонов проектирования:

- Шаблоны создания (Creational patterns) - builder, factory, prototype, singleton
- Структурные шаблоны (Structural patterns) - adapter, bridge, composite, decorator, façade, flyweight, proxy
- Шаблоны поведения (Behavioral patterns) - command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor

В SWEBOK данная тема, в силу упомянутого выше несоответствия между структурной декомпозицией и описанием области знаний “проектирование”, имеет номер 3.2 (следующая тема, в свою очередь, представлена в SWEBOK как 3.3).

3.4 Семейства программ и фреймворков (Families of Programs and Frameworks)

Один из возможных подходов к повторному использованию архитектурных решений и компонент заключается в формировании линий продуктов (product lines) на основе общего дизайна. В объектно-ориентированном программировании аналогичную смысловую нагрузку несут “фреймворки”, обеспечивающие решение одних и тех же задач – например, внутренней организации компонентов пользовательского интерфейса или общей логики работы распределенных систем.

4. Анализ качества и оценка программного дизайна (Software Design Quality Analysis and Evaluation)

4.1 Атрибуты качества (Quality Attributes)

Существует целый спектр различных атрибутов, помогающих оценить и добиться качественного дизайна. Эти атрибуты могут описывать многие характеристики системы и элементов дизайна как

такового – “тестируемость”, “переносимость”, “модифицируемость”, “производительность”, “безопасность” и т.п. Важно понимать, что обсуждаемые атрибуты касаются только дизайна (как результата), но не проектирования (как процесса). В принципе, все эти атрибуты можно разбить на несколько групп:

- применимые к run-time, то есть ко времени выполнения системы; например, среднее время отклика системы позволяющий оценить качество дизайна с точки зрения производительности;
- ориентированные на design-time, то есть позволяющие оценивать качество получаемого дизайна еще на этапе проектирования или, в общем случае, вплоть до тестирования, включительно; например, средняя нагруженность классов бизнес-методами (предположим бизнес-методов в каждом классе в среднем 30 – интересно, насколько легко можно поддерживать, модифицировать и развивать систему с такой внутренней структурой....);
- атрибуты качества архитектурного дизайна как такового, например, концептуальная целостность дизайна, непротиворечивость, полнота, завершенность; например, любой определенный бизнес-метод является вызываемым, то есть создан не просто потому что может понадобиться в будущем, а определен в соответствии с требованиями или необходим для реализации дизайна в выбранном архитектурном стиле.

Необходимо понимать, что существуют атрибуты, которые сложно измерить. Например, портируемость или безопасность. Не стоит путать атрибуты качества дизайна с атрибутами качества, фигурируемыми в ряду требований, предъявляемых к системе. Часть из них может отображаться друг на друга и нести эквивалентную смысловую нагрузку, некоторые могут быть связаны, большая часть атрибутов является специфичной именно для дизайна и не связана с требованиями. Например, если мы используем платформу J2EE (Java 2 Enterprise Edition) и ориентируемся на использование компонентой модели EJB (Enterprise JavaBeans), существуют признаки хорошего дизайна, специфичные для данной платформы и компонентной модели, но абсолютно никак не связанные с какими-либо требованиями к создаваемой на этой платформе программной системе. Если вернуться к измеряемым атрибутам качества, они описываются определенными метриками. Приведенный выше пример с количеством бизнес-методов на класс является метрикой, относящейся к теме 4.3 “Измерения”. Эта же метрика позволяет оценить атрибуты качества “модифицируемость” и “сложность” системы.

4.2 Анализ качества и техники оценки (Quality Analysis and Evaluation Techniques)

В индустрии распространены многие инструменты, техники и практики, помогающие добиться качественного дизайна:

- обзор дизайна (software design review); например, неформальный обзор архитектуры членами проектной команды;
- статический анализ (static analysis); например, трассировка с требованиями;
- симуляция и прототипирование (simulation and prototyping) – динамические техники проверки дизайна в целом или отдельных его атрибутов качества; например, для оценки производительности используемых архитектурных решений при симуляции нагрузки, близкой к прогнозируемым пиковым;

4.3 Измерения (Measures)

Также известные как метрики. Могут быть использованы для количественной оценки ожиданий в отношении различных аспектов конкретного дизайна, например, размера <проекта>, структуры (ее сложности) или качества (например, в контексте требований, предъявляемых к производительности). Чаще всего, все метрики разделяют по двум категориям:

- функционально-ориентированные
- объектно-ориентированные

5. Нотации проектирования (Software Design Notations)

Нотация есть соглашение о представлении. Часто под нотацией подразумевают визуальное (графическое) представление. Нотация может задаваться:

- стандартом; например, OMG UML – Unified Modeling Language, развиваемый консорциумом OMG (Object Management Group, http://www.omg.org);

- общепринятой практикой; например, в eXtreme Programming часто используются карточки функциональной ответственности и связей класса - Class Responsibility Collaborator или CRC Card (CRC по свое природе является текстовой, то есть невизуальной нотацией);
- внутренним методом проектной команды (“будем рисовать и обозначать так...”).

Определенные нотации используются на стадии концептуального проектирования, ряд нотаций ориентирован на создание детального дизайна, многие могут использоваться на обеих стадиях. Кроме того, нотации чаще всего используют в контексте (выбор нотации может быть обусловлен таким контекстом) применяемой методологии или подхода (см. 6 “Software Design Strategies and Methods” данной области знаний). Ниже мы будем рассматривать нотации, исходя из описания структурного (статического) или поведенческого (динамического) представления.

5.1 Структурные описания, статический взгляд (Structural Descriptions, static view)

Следующие нотации, в основном (но, не всегда), являются графическими, описывая и представляя структурные аспекты программного дизайна. Чаще всего они касаются основных компонент и связей между ними (статических связей, например, таких как отношения “один-ко-многим”).

- *Языки описания архитектуры (Architecture description language, ADL)*: текстовые языки, часто – формальные, используемые для описания программной архитектуры в терминах компонентов и коннекторов (специализированных компонентов, реализующих не функциональность, но обеспечивающих взаимосвязь функциональных компонентов между собой и с “внешним миром”);
- *Диаграммы классов и объектов (Class and object diagrams)*: используются для представления набора классов и <статических> связей между ними (например, наследования);
- *Диаграммы компонентов или компонентные диаграммы (Component diagrams)*: в определенной степени аналогичны диаграммам классов, однако, в силу специфики концепции или понятия компонента*, обычно, представляются в другой визуальной форме;

  * здесь необходимо отметить различие в понятиях класса (или объекта) и компонента: компонент рассматривается как физически реализуемый элемент программного обеспечения, несущий <в определенной степени> самодостаточную логику и реализуемый как конгломерат интерфейса и его реализации (часто, в виде комплекса классов);
- *Карточки <функциональной> ответственности и связей класса (Class responsibility collaborator card, CRC)*: используются для обозначения имени класса, его ответственности (то есть, что он должен делать) и других сущностей (классов, компонентов, актёров/ролей и т.п.), с которыми он связан; часто их называют карточками “класс-обязанность-кооперация”;
- *Диаграммы развёртывания (Deployment diagrams)*: используется для представления (физических) узлов, связей между ними и моделирования других физических аспектов системы;
- *Диаграммы сущность-связь (Entity-relationship diagram, ERD или ER)*: используется для представления концептуальной модели данных, сохраняемых в процессе работы информационной системы;
- *Языки описания/определения интерфейса (Interface Description Languages, IDL)*: языки, подобные языкам программирования, не включающие возможностей описания логики системы и предназначенные для определения интерфейсов программных компонентов (имён и типов экспортируемых или публикуемых операций);
- *Структурные диаграммы Джексона (Jackson structure diagrams)*: используются для описания структур данных в терминах последовательности, выбора и итераций (повторений);
- *Структурные схемы (Structure charts)*: описываю структуру вызовов в программах (какой модуль вызывает, кем и как вызываем).

5.2 Поведенческие описания, динамический взгляд (Behavioral Descriptions, dynamic view)

Следующие нотации и языки (часть из которых – графические, часть - текстовые) используются для описания динамического поведения программных систем и их компонентов. Многие из этих нотаций успешно используются для проектирования деталей дизайна, но не только для этого.

- *Диаграммы деятельности или операций (Activity diagrams)*: используются для описания потоков работ и управления;
- *Диаграммы сотрудничества (Collaboration diagrams)*: показывают динамическое взаимодействие, происходящее в группе объектов и уделяют особое внимание объектам, связям между ними и сообщениям, которыми обмениваются объекты посредством этих связей;
- *Диаграммы потоков данных (Data flow diagrams, DFD)*: описывают потоки данных внутри набора процессов (не в терминах процессов операционной среды, но в понимании обмена информацией в бизнес-контексте);
- *Таблицы и диаграммы <принятия> решений (Decision tables and diagrams)*: используются для представления сложных комбинаций условий и действий (операций);
- *Блок-схемы и структурированные блок-схемы (Flowcharts and structured flowcharts)*: применяются для представления потоков управления (контроля) и связанных операций;
- *Диаграммы последовательности (Sequence diagrams)*: используются для показа взаимодействий внутри группы объектов с акцентом на временной последовательности сообщений/вызовов;
- *Диаграммы перехода и карты состояний(State transition and statechart diagrams)*: применяются для описания потоков управления переходами между состояниями;
- *Формальные языки спецификации (Formal specification languages)*: текстовые языки, использующие основные понятия из математики (например, множества) для строгого и абстрактного определения интерфейсов и поведения программных компонентов, часто в терминах пред- и пост-условий;
- *Псевдокод и программные языки проектирования (Pseudocode and program design languages, PDL)*: языки, используемые для описания поведения процедур и методов, в основном на стадии детального проектирования; подобны структурным языкам программирования.

6. Стратегии и методы проектирования программного обеспечения (Software Design Startegies and Methods)

Существуют различные общие стратегии, помогающие в проведении работ по проектированию. В отличие от общих стратегий, методы проектирования более специфичны и, в основном, предлагают и предоставляют нотации (или наборы нотаций) для использования совместно с этими методами, а также процессы, которым необходимо следовать в рамках используемого метода.
Таким образом, методы в данном контексте это не просто некие “слабоформализованные” или просто частные практические подходы или техники. Методы здесь являются более общими понятиями, это - *методологии*, сконцентрированные на процессе (в частности, проектирования) и предполагающие следование определенным правилам и соглашениям, в том числе – по используемым выразительным средствам. Такие методы полезны как инструмент систематизации (иногда, формализации) и передачи знаний в виде общего фреймворка (то есть комплексного набора понятий, подходов, техник и инструментов) не только для отдельных специалистов, но для команд и проектных групп программных проектов.

6.1 Общие стратегии (General Strategies)

Это обычно часто упоминаемые и общепринятые стратегии:

- “разделяй-и-властвуй” и пошаговое уточнение
- проектирование “сверху-вниз” и “снизу-вверх”
- абстракция данных и сокрытие информации
- итеративный и инкрементальный подход
- и другие...

6.2 Функционально-ориентированное или структурное проектирование (Function-Oriented – Structured Design)

Это один из классических методов проектирования, в котором декомпозиция сфокусирована на идентификации основных программных функций и, затем, детальной разработке и уточнении этих функций “сверху-вниз”. Структурное проектирование, обычно, используется после проведения

структурного анализа с применением диаграмм потоков данных и связанным описанием процессов. Исследователи предлагают различные стратегии и метафоры или подходы для трансформации DFD в программную архитектуру, представляемую в форме структурных схем. Например, сравнивая управление и поведение с получаемым эффектом.

6.3 Объектно-ориентированное проектирование (Object-Oriented Design)

Представляет собой множество методов проектирования, базирующихся на концепции объектов. Данная область активно эволюционирует с середины 80-х годов, основываясь на понятиях объекта (сущности), метода (действия) и атрибута (характеристики). Здесь главную роль играют полиморфизм и инкапсуляция, в то время, как в компонентно-ориентированном подходе большее значение придается мета-информации, например, с применением технологии отражения (reflection). Хотя корни объектно-ориентированного проектирования лежат в абстракции данных (к которым добавлены поведенческие характеристики), так называемый responsibility-driven design или проектирование на основе <функциональной> ответственности по SWEBOK* может рассматриваться как альтернатива объектно-ориентированному проектированию.

*С точки зрения автора книги, такое противопоставление – достаточно спорный вопрос, так как функциональная ответственность столь же близка принципам современного объектно-ориентированного проектирования, сколь и абстракция данных. Это вопрос эволюционирования взглядов и степени их консерватизма.

6.4 Проектирование на основе структур данных (Data-Structure-Centered Design)

В данном подходе фокус сконцентрирован в большей степени на структурах данных, которыми управляет система, чем на функциях системы. Инженеры по программному обеспечению часто вначале описывают структуры данных входов (inputs) и выходов (outputs), а, затем, разрабатывают структуру управления этими данными (или, например, их трансформации).

6.5 Компонентное проектирование (Component-Based Design)

Программные компоненты являются независимыми единицами, которые обладают однозначно-определенными (well-defined) интерфейсами и зависимостями (связями) и могут собираться и развертываться независимо друг от друга. Данный подход призван решить задачи использования, разработки и интеграции таких компонент с целью повышения повторного использования активов (как архитектурных, так и в форме кода).

Компонентно-ориентированное проектирование является одной из наиболее динамично развивающихся концепций проектирования и может рассматриваться как предвестник и основа *сервисно-ориентированного подхода (Service-Oriented Architecture, SOA)* в проектировании, не рассматриваемого, к сожалению, в SWEBOK, но все более активно использующегося в индустрии и смещающего акценты с аспектов организации связи интерфейс-реализация к обмену информацией на уровне интерфейс-интерфейс (то есть – межкомпонентному взаимодействию). По мнению автора книги, уже наступил тот момент, когда необходимо вводить отдельную тему, посвященную сервисно-ориентированному подходу в проектировании и сервисно-ориентированным архитектурам, как моделям. В частности, нотация UML 2.0 уже позволяет решать ряд вопросов, связанных с визуальным представлением соответствующих архитектурных решений, где *сервисы (службы)* могут рассматриваться как публикуемая функциональность одиночных компонентов и групп компонентов, объединенных в более “крупные” блоки, обеспечивающие предоставление соответствующей сервисной функциональности.

6.6 Другие методы (Other Methods)

Другие интересные, но менее распространенные подходы, в основном, представляют собой формальные и точные (строгие) методы, а также, методы трансформации.