## Тема 3.1. Введение в разработку программного обеспечения

[**3.1.1. Основные понятия**](#_3.1.1.Основные_понятия)

[**3.1.2. Проблемы разработки сложных программных систем**](#_3.1.2._Проблемы_разработки)

[**3.1.3. Понятие жизненного цикла ПО. Модели жизненного цикла**](#_3.1.3._Понятие_жизненного)

### 3.1.1.Основные понятия

Основой проектирования современного программного обеспечения является системный подход.

***Системный подход*** – это методология исследования объекта любой природы как системы.

***Система*** – это совокупность взаимосвязанных частей, работающих совместно для достижения некоторого результата. Определяющий признак системы – поведение системы в целом не сводимо к совокупности поведения частей системы.

***Программное обеспечение (ПО)*** – это система, включающая в себя: компьютерные программы; документацию; данные, необходимые для корректной работы программ.

***Спецификаций ПО*** – это законченное описание поведения [ПО](http://ru.wikipedia.org/wiki/%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%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), которое требуется разработать.

***Проектирование ПО*** – включает в себя: процесс создания спецификаций ПО на основе исходных требований к нему; процесс создания проекта программного обеспечения; дисциплина, изучающая методы проектирования. Проектирование подразумевает выработку свойств системы на основе анализа постановки задачи, а именно: моделей предметной области, [требований к ПО](http://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B5%D0%B1%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%BC%D1%83_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8E), а также опыта проектировщика.

Модель предметной области накладывает ограничения на логику предметной области и [структуры данных](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85).

В зависимости от класса, создаваемого ПО, процесс проектирования может обеспечиваться как «ручным» проектированием, так и различными средствами его автоматизации. В процессе проектирования ПО для выражения его характеристик используются различные нотации – [схемы](http://ru.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D0%BA-%D1%81%D1%85%D0%B5%D0%BC%D0%B0) алгоритмов, [ER-диаграммы](http://ru.wikipedia.org/wiki/ER-%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), [UML](http://ru.wikipedia.org/wiki/UML)-диаграммы, [DFD](http://ru.wikipedia.org/wiki/DFD)-диаграммы и др..

* Проектированию обычно подлежат:
* [Архитектура ПО](http://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F); Устройство компонентов ПО;
* [Пользовательские интерфейсы](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8F).

***Архитектура ПО*** – это [структура](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0) [программы](http://ru.wikipedia.org/wiki/%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%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), которая включает программные компоненты, видимые снаружи свойства этих компонентов, а также отношения между ними. Этот термин также относится к документированию архитектуры программного обеспечения. [Документирование](http://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%86%D0%B8%D1%8F_%D0%BD%D0%B0_%D0%BF%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%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) архитектуры ПО упрощает процесс коммуникации между [заинтересованными лицами](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%B9%D0%BA%D1%85%D0%BE%D0%BB%D0%B4%D0%B5%D1%80), позволяет зафиксировать принятые на ранних этапах проектирования решения и позволяет использовать [компоненты](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82) этих решений повторно в других проектах.

***Проект ПО*** – совокупность спецификаций ПО (включающих модели и проектную документацию), обеспечивающих создание ПО в конкретной программно-технической среде.

Программное обеспечение условно можно разбить на два класса: «малое» и «большое».

***«Малое»*** программное обеспечение имеет следующие характеристики:

* решает одну несложную, четко поставленную задачу;
* размер исходного кода не превышает нескольких сотен строк;
* скорость работы программного обеспечения и необходимые ему ресурсы не играют большой роли;
* ущерб от неправильной работы не имеет большого значения;
* модернизация программного обеспечения, дополнение его возможностей требуется редко;
* как правило, разрабатывается одним программистом или небольшой группой;
* подробная документация не требуется, ее может заменить исходный код, который доступен.

Примером «малого» ПО могут служить базовые алгоритмы, изучаемые в разделе 2.

***«Большое» программное обеспечение*** имеет 2-3 или более характеристик из следующего перечня:

* решает совокупность взаимосвязанных задач;
* использование приносит значимую выгоду;
* удобство его использования играет важную роль;
* обязательно наличие полной и понятной документации;
* низкая скорость работы приводит к потерям;
* сбои, неправильная работа, наносит ощутимый ущерб;
* программы в составе ПО во время работы взаимодействует с другими программами и программно-аппаратными комплексами;
* работает на разных платформах;
* требуется развитие, исправление ошибок, добавление новых возможностей;
* группа разработчиков состоит из более, чем 5 человек.

Кратко рассмотрим процесс проектирования «большого» ПО, поскольку создание «малого» обычно не вызывает больших трудностей.

Классификация программных проектов по размеру группы разработчиков и длительности проекта может быть следующая:

* ***небольшие проекты*** – проектная команда менее 10 человек, срок от 3 до 6 месяцев;
* ***средние проекты*** – проектная команда от 20 до 30 человек, протяженность проекта 1-2 года;
* ***крупномасштабные проекты*** – проектная команда от 100 до 300 человек, протяженность проекта 3-5 лет;
* ***гигантские проекты*** – армия разработчиков от 1000 до 2000 человек и более (включая консультантов и соисполнителей), протяженность проекта от 7 до 10 лет.

Таким образом***, сложное*** или ***«большое» ПО*** (которое называют также ***программными системами***, ***программными комплексами***, ***программными продуктами***) отличается от «небольшого» не столько по размерам, сколько по наличию дополнительных факторов, связанных с его востребованностью и готовностью пользователей «платить» как за приобретение самого программного продукта, так и за его сопровождение и за специальное обучение работе с ним.

### 3.1.2. Проблемы разработки сложных программных систем

Обычно сложная программа обладает следующими свойствами:

* она решает одну или несколько связанных задач, зачастую сначала не имеющих четкой постановки, настолько важных для каких-либо лиц или организаций, что те приобретают значимые выгоды от ее использования;
* существенно, чтобы она была удобной в использовании (в частности, она должна включать достаточно полную и понятную пользователям документацию, а также набор документов для обучения работе с программой);
* ее низкая производительность на реальных данных приводит к значимым потерям для пользователей;
* ее неправильная работа наносит ощутимый ущерб пользователям и другим организациям и лицам, даже если сбои происходят не слишком часто;
* для выполнения своих задач она должна взаимодействовать с другими программами и программно-аппаратными системами, работать на разных платформах;
* пользователи, работающие с ней, приобретают дополнительные выгоды от того, что программа развивается, в нее вносятся новые функции и устраняются ошибки (необходимо наличие проектной документации, позволяющей развивать ее, возможно, вовсе не тем разработчикам, которые ее создавали, без больших затрат на обратную разработку);
* в ее разработку вовлечено значительное количество людей («большую» программу практически невозможно написать с первой попытки, с небольшими усилиями и в одиночку);
* намного больше количество ее возможных пользователей, и еще больше тех лиц, деятельность которых будет так или иначе затронута ее работой и результатами.

Примером «большой» программы может служить стандартная библиотека классов C# или VB и др.

Строго говоря, ни одно из указанных свойств не является обязательным для того, чтобы программу можно было считать «большой», но при наличии двух-трех из них достаточно уверенно можно утверждать, что она «большая».

На основании некоторых из перечисленных свойств можно сделать вывод, что «большая» программа или программная система чаще всего представляет собой не просто код или исполняемый файл, а включает еще и набор проектной и пользовательской документации.

Для разработки программных систем требуются особые методы и технологии. Изучением организационных, инженерных и технических аспектов создания ПО, включая методы разработки, занимается дисциплина, которая называется ***программная инженерия***. Большая часть трудностей при разработке программных систем связана с организацией экономически эффективной совместной работы многих людей, приводящей к практически полезному результату.

Важно отметить, что ***практически полезная*** сложная программная система не обязательно является ***«правильной»***.

Большинство опытных разработчиков ПО считают, что практически значимые программные системы всегда содержат ошибки. При переходе от «небольших» программ к «большим» понятие «правильной» программы становится практически бессмысленным. Этот связано как с практической невозможностью полного доказательства или проверки этого, так и с тем, что смысл существования программной системы – удовлетворение потребностей и запросов большого количества различных заинтересованных лиц. А эти потребности не только нечетко определены, различны для разных групп пользователей и иногда противоречивы, но и значительно изменяются с течением времени.

Поэтому и основные проблемы разработки сложных программных систем связаны с нахождением разумного компромисса между затратами на разработку и качеством ее результата. В затраты входят все виды используемых ресурсов, из которых наиболее важны затрачиваемое время, бюджет проекта и используемый персонал. Удовлетворение пользователей от работы с программой (а, следовательно, доходы от ее продаж и предоставления дополнительных услуг) и удовлетворение разработчиков от ее создания определяются качеством программы, которое включает в себя такие аспекты, как набор предоставляемых возможностей, надежность, удобство использования, гибкость, удобство внесения изменений и исправления ошибок.

Кратко рассмотрим различные подходы к решению проблем разработки, при создании сложных программ. Для изложения этих подходов вводится система понятий, относящихся к программным системам и процессам их проектирования. В их число входят такие понятия, как ***жизненный цикл ПО***, ***процесс разработки ПО***, ***требования к ПО***, ***архитектура ПО***, ***образцы проектирования*** и пр.

Кроме того, особое внимание уделяется одному из современных подходов к разработке сложного ПО, ***объектной разработке***, предлагающей строить такие системы последовательно из отдельных элементов – **объектов**.

На основании опыта конструирования больших программных систем «сообществом программистов» разработаны так называемые ***технологические процессы***, содержащие достаточно детальные описания разных аспектов их создания и эксплуатации. Эти описания дают ответы на вопросы о том, как должна вестись разработка, какие лица должны в ней участвовать и на каких этапах, какие виды деятельности и в какой последовательности должны выполняться, какие документы являются входными, а какие документы и другие части программной системы должны быть подготовлены в результате проектирования. При практической разработки больших систем, однако, стоит помнить, что все обще-методические рекомендации имеют границы применимости, и чем детальнее они определяют действия разработчиков, тем вероятнее, что что-то пойдет не так, как это предусматривается авторами методик. Кроме того, огромное количество вспомогательных по сути документов, оформление которых часто требуется подобными методиками, иногда затрудняет понимание основных целей проекта, принимаемых в его ходе решений и сути происходящего в нем.

Помимо методических рекомендаций, при конструировании больших систем часто используются прагматические принципы работы со сложными системами вообще. Они играют значительную роль в выработке качественных технических решений в достаточно широком контексте. Эти принципы позволяют распределять работы между участвующими в проектах людьми с меньшими затратами на обеспечение их взаимодействия и акцентировать внимание каждого из участников на наиболее существенных для его части работы характеристиках системы. К таким принципам относятся использование ***абстракции* и *уточнения*, *модульная разработка*** и ***повторное использование*.**

***Абстракция,*** как известно, является универсальным подходом к рассмотрению сложных систем. Интеллект одного человека достаточно ограничен, и не в силах иметь дело сразу со всеми элементами и свойствами систем большой сложности. Известно, что человеку крайне тяжело держать в голове одновременно десяток-полтора различных мыслей, а в современных системах число различных существенных аспектов доходит до сотен. Для того, чтобы все-таки как-то работать с такими системами, разработчики ПО пользуются возможностью ***абстрагироваться*** от всего, что несущественно для достижения поставленной в данной момент частной цели и не влияет на те аспекты рассматриваемой предметной области, которые для этой цели важны.

Чтобы перейти от абстрактного представления к более конкретному представлению, используется обратный процесс последовательного ***уточнения.*** Рассмотрев систему в каждом аспекте в отдельности, разработчики ПО пытаются объединить результаты анализа, добавляя аспекты по одному и обращая при этом внимание на возможные взаимные влияния и возникающие связи между элементами, выявленными при анализе отдельных аспектов.

***Абстракция*** и ***уточнение*** используются, прежде всего, для получения работоспособных решений, гарантирующих нужные свойства результирующей системы.

***Рассмотрим пример абстракции и уточнения****.* Систему идентификаторов пользователей Интернет-магазина можно представить как множество целых чисел, забыв о том, что эти числа – идентификаторы пользователей, и о том, что все это связано с Интернет- магазином. Затем описанную модель системы идентификаторов пользователей Интернет- магазина можно уточнить, определив конкретную реализацию множества чисел, например, на основе сбалансированных деревьев.

***Рассмотрим еще один пример****.* Рассматривая задачу передачи данных по сети, можно временно абстрагироваться от большинства проблем организации связи и заниматься только одним аспектом – организацией надежной передачи данных в нужной последовательности. При этом можно предполагать, что мы как-то умеем передавать данные между двумя компьютерами в сети, хотя, быть может, и с потерями и с нарушением порядка их прибытия по сравнению с порядком отправки. Установленные ограничения выделяют достаточно узкий набор задач. Любое их решение представляет собой некоторый *протокол передачи данных транспортного уровня*, т.е. нацеленный именно на надежную упорядоченную доставку данных. Выбирая такой протокол, из уже существующих протоколов, например, TCP, или разрабатывая новый, мы производим уточнение исходной общей задачи передачи данных.

Другой способ уточнения – перейти к рассмотрению протоколов, обеспечивающих исходные условия для нашей первой абстракции, т.е. возможность вообще что-то передавать по сети. При этом возникают протоколы нижележащих уровней–*сетевого* (отвечают за организацию связи между не соединенными непосредственно компьютерами при наличии между ними цепи машин, соединенных напрямую), *канального* (такие протоколы отвечают за определение формата передаваемых данных и надежность передачи отдельных элементов информации между двумя физически соединенными компьютерами) и *физического* (отвечают за определение физического носителя передаваемого сигнала и правильность интерпретации таких сигналов обеими машинами, в частности, за конкретный способ передачи битов с помощью электрических сигналов или радиоволн).

***Модульность*** – принцип организации больших систем в виде наборов подсистем, модулей или компонентов. Этот принцип предписывает организовывать сложную систему в виде набора более простых систем – модулей, взаимодействующих друг с другом через четко определенные интерфейсы. При этом каждая задача, решаемая всей системой, разбивается на более простые, решаемые отдельными модулями подзадачи, решение которых, будучи скомбинировано определенным образом, дает в итоге решение исходной задачи. После этого можно отдельно рассматривать каждую подзадачу и модуль, ответственный за ее решение, и отдельно – вопросы интеграции полученного набора модулей в целостную систему, способную решать исходные задачи.

Выделение четких интерфейсов для взаимодействия упрощает интеграцию, позволяя проводить ее на основе явно очерченных возможностей этих интерфейсов, без обращения к многочисленным внутренним элементам модулей, что привело бы к росту сложности.

***Рассмотрим пример модульности****.* Примером разбиения на модули может служить структура пакетов и классов библиотеки .Net. Классы, связанные с основными сущностями языка программирования и виртуальной машины, собраны в одну библиотеку классов. Вспомогательные широко применяемые в различных приложениях классы, такие, как коллекции, представления даты и пр., собраны в другую библиотеку классов. Классы, используемые для реализации потокового ввода-вывода – в третью библиотеку классов, и т.д.

Интерфейсом класса служат его общедоступные методы, а интерфейсом пакета – его общедоступные классы.

***Рассмотрим еще один пример****.* Другой пример модульности – принятый способ организации протоколов передачи данных. Мы уже видели, что удобно выделять несколько уровней протоколов, чтобы на каждом решать свои задачи. При этом надо определить, как информация передается от машины к машине при помощи всего этого многоуровневого механизма. Обычное решение таково: для каждого уровня определяется способ передачи информации **с** или **на** верхний уровень – предоставляемые данным уровнем службы. Точно так же определяется, в каких службах нижнего уровня нуждается верхний, т.е. как передать данные на нижний уровень и получить их оттуда. После этого каждый протокол на данном уровне может быть сформулирован в терминах обращений к нижнему уровню и должен реализовать операции-службы, необходимые верхнему. Это позволяет заменять протокол-модуль на одном уровне без внесения изменений в другие.

Хорошее разбиение системы на модули – непростая задача. При ее выполнении привлекаются следующие дополнительные принципы.

* **Выделение интерфейсов и сокрытие информации.**  
  Модули должны взаимодействовать друг с другом через четко определенные *интерфейсы* и скрывать друг от друга внутреннюю информацию – внутренние данные, детали реализации интерфейсных операций.  
  При этом интерфейс модуля обычно значительно меньше, чем набор всех операций и данных в нем.
* **Адекватность, полнота, минимальность и простота интерфейсов.**Этот принцип объединяет ряд свойств, которыми должны обладать хорошо спроектированные интерфейсы**.**
  + ***Адекватность интерфейса*** означает, что интерфейс модуля дает возможность решать именно те задачи, которые нужны пользователям этого модуля. Например, добавление в интерфейс очереди метода, позволяющего получить любой ее элемент по его номеру в очереди, сделало бы этот интерфейс не вполне адекватным – он превратился бы почти в интерфейс списка, который используется для решения других задач. Очереди же используются там, где полная функциональность списка не нужна, а реализация очереди может быть сделана более эффективной.
  + ***Полнота интерфейса*** означает, что интерфейс позволяет решать все значимые задачи в рамках функциональности модуля. Например, отсутствие в интерфейсе очереди метода offer() сделало бы его бесполезным – никому не нужна очередь, из которой можно брать элементы, а класть в нее ничего нельзя.
  + ***Минимальность интерфейса*** означает, что предоставляемые интерфейсом операции решают различные по смыслу задачи и ни одну из них нельзя реализовать с помощью всех остальных (или же такая реализация довольно сложна и неэффективна). Большое значение минимальности интерфейса уделяют, если размер модулей оказывает сильное влияние на производительность программы. Например, при проектировании модулей операционной системы — чем меньше она занимает места в памяти, тем больше его останется для приложений, непосредственно необходимых пользователям.  
    При проектировании библиотек более высокого уровня имеет смысл не делать интерфейс минимальным, давая пользователям этих библиотек возможности для повышения производительности и понятности их программ. Например, часто бывает полезно реализовать действия «проверить, что элемент не принадлежит множеству, и, если нет, добавить его» в одном методе, не заставляя пользователей каждый раз сначала проверять принадлежность элемента множеству, а затем уже добавлять его.
  + ***Простота интерфейса*** означает, что интерфейсные операции достаточно элементарны и не представимы в виде композиций некоторых более простых операций на том же уровне абстракции, при том же понимании функциональности модуля.
* **Разделение ответственности.**

Основной принцип выделения модулей – создание отдельных модулей под каждую задачу, решаемую системой или необходимую в качестве составляющей для решения ее основных задач. Принцип разделения ответственности имеет несколько важных частных случаев.

* + **Разделение политик и алгоритмов.**

Этот принцип используется для отделения постоянных, неизменяемых алгоритмов обработки данных от изменяющихся их частей и для выделения этих частей, называемых *политиками,* в параметры общего алгоритма.  
Так, политика, определяющая формат строкового представления даты и времени, задается в виде форматной строки при создании объекта. Сам же алгоритм построения этого представления основывается на этой форматной строке и на самих времени и дате.

Другой пример. Стоимость товара для клиента может зависеть от привилегированности клиента, размера партии, которую он покупает и сезонных скидок. Все перечисленные элементы можно выделить в виде политик, являющихся, вместе с базовой ценой товара, входными данными для алгоритма вычисления итоговой стоимости.

* + **Разделение интерфейса и реализации.**

Этот принцип используется при отделении внешне видимой структуры модуля, описания задач, которые он решает, от способов решения этих задач.

* **Слабая связность (coupling) модулей и сильное сродство (cohesion) функций в одном модуле.**

Оба эти принципа используются для выделения модулей в большой системе и тесно связаны с разделением ответственности между модулями. Первый требует, чтобы зависимостей между модулями было как можно меньше. Модуль, зависящий от большинства остальных модулей в системе, скорее всего, надо перепроектировать – это означает, что он решает слишком много задач. И наоборот, «сродство» функций, выполняемых одним модулем, должно быть как можно выше. Хотя на уровне кода причины этого «сродства» могут быть разными – работа с одними и теми же данными, зависимость от работы друг друга, необходимость синхронизации при параллельном выполнении и пр. – цена их разделения должна быть достаточно высокой. Наиболее существенно то, что эти функции решают тесно связанные друг с другом задачи.

* **Повторное использование.**

Этот принцип требует избегать повторений описаний одних и тех же знаний – в виде структур данных, действий, алгоритмов, одного и того же кода – в разных частях системы. Вместо этого в хорошо спроектированной системе выделяется один источник, одно место фиксации для каждого элемента знаний и организуется его повторное использование во всех местах, где нужно использовать этот элемент знаний. Такая организация позволяет при необходимости (например, при исправлении ошибки или расширении имеющихся возможностей) удобным образом модифицировать код и документы системы в соответствии с новым содержанием элементов знаний, поскольку каждый из них зафиксирован ровно в одном месте.

### 3.1.3. Понятие жизненного цикла ПО. Модели жизненного цикла.

Ранее говорилось о том, что сложную программную систему построить «простыми» методами невозможно. Ее разработка с неизбежностью будет тоже сложной деятельностью.

Разработка ПО имеет следующие специфические особенности:

* неформальный характер требований к ПО и формализованный основной объект разработки – программы;
* творческий характер разработки;
* дуализм ПО, которое, с одной стороны, является статическим объектом – совокупностью текстов, с другой стороны, – динамическим, поскольку при эксплуатации порождаются процессы обработки данных;
* при своем использовании (эксплуатации) ПО не расходуется и не изнашивается;
* «неощутимость», «воздушность» ПО, что подталкивает к безответственному переделыванию, поскольку легко стереть и переписать, чего не сделаешь при проектировании зданий и аппаратуры.

Привести такое предприятие к успеху возможно на основе общих принципов работы со сложными системами: организовав его в виде набора модулей, используя разные уровни абстракции, повторное использование отдельных элементов в разных местах так, чтобы изменения в таком элементе автоматически отражались всюду, где он используется.

Разработка ПО – разновидность человеческой деятельности. Выделить ее компоненты можно, определив набор задач, которые нужно решить для достижения конечной цели – построения достаточно качественной системы в рамках заданных сроков и ресурсов. Для решения каждой такой задачи организуется вспомогательная деятельность, к которой можно также применить декомпозицию на отдельные, более мелкие деятельности, и т.д. В итоге должно стать понятно, как решать каждую отдельную подзадачу и всю задачу целиком на основе имеющихся решений для подзадач.

В качестве примеров деятельностей, которые нужно проводить для построения программной системы, можно привести ***проектирование*** – выделение отдельных модулей и определение связей между ними с целью минимизации зависимостей между частями проекта и достижения лучшей его обозримости в целом, ***кодирование***– разработку кода отдельных модулей, разработку пользовательской документации, которая необходима для достаточно сложной системы. ***Инженерия ПО*** (softwareengineering) – совокупность инженерных методов и средств создания ПО. Фундаментальная идея программной инженерии: проектирование ПО является формальным процессом, который можно изучать и совершенствовать.

Освоение и правильное применение методов и средств программной инженерии позволяет повысить качество, обеспечить управляемость процесса проектирования.

Программная инженерия применяется для удовлетворения требований заказчика ПО. Основные цели программной инженерии:

* системы должны создаваться в короткие сроки и соответствовать требованиям заказчика на момент внедрения;
* качество ПО должно быть высоким;
* разработка ПО должна быть осуществлена в рамках выделенного бюджета;
* системы должны работать на оборудовании заказчика, а также взаимодействовать с имеющемся ПО;
* системы должны быть легко сопровождаемыми и масштабируемыми

Однако для корректного с точки зрения инженерии и экономики рассмотрения вопросов создания сложных систем необходимо, чтобы были затронуты и вопросы эксплуатации системы, внесения в нее изменений, а также самые первые действия в ходе ее создания – анализ потребностей пользователей и выработка решений, «изобретение» функций, удовлетворяющих эти потребности. Без этого невозможно, с одной стороны, учесть реальную эффективность системы в виде отношения полученных результатов ко всем сделанным затратам и, с другой стороны, правильно оценивать в ходе разработки степень соответствия системы реальным нуждам пользователей и заказчиков.

Все эти факторы приводят к необходимости рассмотрения всей совокупности деятельности, связанной с созданием и использованием **ПО**, начиная с возникновения идеи о новом продукте и заканчивая удалением его последней копии. Весь период существования ПО, связанный с подготовкой к его разработке, разработкой, использованием и модификациями, начиная с того момента, когда принимается решение разработать/приобрести/собрать, до того момента, когда полностью прекращается всякое ее использование, называют ***жизненным циклом ПО***.

Основным понятием программной инженерии является понятие ***жизненного цикла ПО*.**

***Жизненный цикл*** *(****ЖЦ****)* ***ПО*** *(softwarelifecycle) –* этот период времени, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации.

С точки зрения статической структуры ЖЦ является совокупностью процессов ЖЦ.

***Процесс ЖЦ*** – набор взаимосвязанных действий, преобразующих некоторые входные данные и ресурсы в выходные.

Каждый процесс характеризуется задачами, методами их решения, действующими лицами, результатами. Процессы ЖЦ протекают параллельно. Каждый процесс разделен на набор действий, каждое действие – на набор задач. Каждый процесс, действие или задача инициируется и выполняется по мере необходимости, причем не существует заранее определенных последовательностей выполнения:

* *основные* (приобретение, поставка, разработка, эксплуатация, сопровождение);
* *вспомогательные* (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, совместная оценка, аудит, разрешение проблем);
* *организационные* (управление, создание инфраструктуры, усовершенствование, обучение).

В ходе жизненного цикла **ПО** проходит через анализ предметной области, сбор требований, проектирование, кодирование, тестирование, сопровождение и др. ***виды деятельности***. Каждый вид представляет собой достаточно однородный набор действий, выполняемых для решения одной задачи или группы тесно связанных задач в рамках разработки и поддержки эксплуатации ПО.

При этом создаются и перерабатываются различного рода ***артефакты*** — создаваемые человеком информационные сущности, документы в достаточно общем смысле, участвующие в качестве входных данных и получающиеся в качестве результатов различных деятельностей. Примерами артефактов являются: модель предметной области, описание требований, техническое задание, архитектура системы, проектная документация на систему в целом и на ее компоненты, прототипы системы и компонентов, собственно, исходный код, пользовательская документация, документация администратора системы, руководство по развертыванию, база пользовательских запросов, план проекта, и пр.

На различных этапах в создание и эксплуатацию ПО вовлекаются люди, выполняющие различные ***роли***. Каждая роль может быть охарактеризована как абстрактная группа заинтересованных лиц, участвующих в деятельности по созданию и эксплуатации системы и решающих одни и те же задачи или имеющих одни и те же интересы по отношению к ней. Примерами ролей являются: бизнес-аналитик, инженер по требованиям, архитектор, проектировщик пользовательского интерфейса, программист-кодировщик, технический писатель, тестировщик, руководитель проекта по разработке, работник отдела продаж, конечный пользователь, администратор системы, инженер по поддержке и т.п.

Похоже, что общую структуру жизненного цикла любого ПО задать невозможно, поскольку она существенно зависит от целей, для которых это ПО разрабатывается или приобретается, и от решаемых им задач. Структура жизненного цикла будет существенно отличаться у программы для форматирования кода, которая сначала делалась программистом для себя, а впоследствии была признана перспективной в качестве продукта и переработана, и у комплексной системы автоматизации предприятия, которая с самого начала задумывалась как таковая. Тем не менее, часто определяют основные элементы структуры жизненного цикла в виде ***модели жизненного цикла ПО***.

***Модель жизненного цикла ПО*** выделяет конкретные наборы видов деятельности (обычно разбиваемых на еще более мелкие активности), артефактов, ролей и их взаимосвязи, а также дает рекомендации по организации процесса в целом. Эти рекомендации включают ответы на вопросы о том, какие артефакты являются входными данными у каких видов деятельности, а какие появляются в качестве результатов; в какие роли вовлечены в различные деятельности; как различные деятельности связаны друг с другом; каковы критерии качества полученных результатов; как оценить степень соответствия различных артефактов общим задачам проекта и когда можно переходить от одной деятельности к другой.

Жизненный цикл ПО является составной частью жизненного цикла программно-аппаратной системы, в которую это ПО входит. Поэтому часто различные его аспекты рассматриваются в связи с элементами жизненного цикла системы в целом.

Существует набор стандартов, определяющих различные элементы в структуре жизненных циклов ПО и программно-аппаратных систем. В качестве основных таких элементов выделяют ***технологические процессы*** — структурированные наборы деятельностей, решающих некоторую общую задачу или связанную совокупность задач, такие, как процесс сопровождения ПО, процесс обеспечения качества, процесс разработки документации и пр. Процессы могут определять разные этапы жизненного цикла и увязывать их с различными видами деятельностей, артефактами и ролями заинтересованных лиц.

Стоит отметить, что процессом (или технологическим процессом) называют и набор процессов, увязанных для совместного решения более крупной задачи, например, всю совокупность деятельностей, входящих в жизненный цикл ПО. Таким образом, процессы могут разбиваться на подпроцессы, решающие частные подзадачи той задачи, с которой работает общий процесс.

В рамках специфических моделей жизненного цикла, которые предписывают правила организации разработки ПО в рамках данной отрасли или организации, определяются более конкретные процессы разработки. Отличаются они от стандартов, прежде всего, большей детальностью и четким описанием связей между отдельными видами деятельности, определением потоков данных (документов и артефактов) в ходе жизненного цикла. Таких моделей довольно много, ведь фактически каждый раз, когда некоторая организация определяет собственный процесс разработки, в качестве основы этого процесса разрабатывается некоторая модель жизненного цикла ПО. В рамках данной лекции мы рассмотрим лишь несколько моделей. К сожалению, очень тяжело выбрать критерии, по которым можно было бы дать хоть сколько-нибудь полезную классификацию известных моделей жизненного цикла. Такими моделями ЖЦ могут быть:

* каскадная (водопадная);
* эволюционная;
* основанная на формальных преобразованиях;
* итерационные (пошаговая и спиральная).

Формирование требований

Проектирование

Реализация

Тестирование

Ввод в действие

Эксплуатация и сопровождение

Снятие с эксплуатации

Принципы *каскадной модели*: фиксация требований к системе в начале проекта; переход со стадии на стадию только после полного завершения работ на текущей стадии; недопустимость возврата на пройденные стадии; жесткая привязка процессов ЖЦ к стадиям ЖЦ.

Стадия *формирования требований* включает процессы, приводящие к созданию документа, описывающего поведение ПО с точки зрения внешнего по отношению к нему наблюдателя с фиксацией требований относительно его качества.

*Проектирование* охватывает процессы: разработку архитектуры ПО, разработку структур программ в его составе и их детальную спецификацию.

*Реализация* или кодирование включает процессы создания текстов программ на языках программирования.

На этапе *тестирования* производится собственно тестирование, а также отладка и оценка качества ПО.

*Ввод в действие* – это развертывание ПО на целевой вычислительной системе, обучение пользователей и т.п.

*Эксплуатация ПО* – это использование ПО для решения практических задач на компьютере путем выполнения ее программ.

*Сопровождение ПО* – это процесс сбора информации о качестве ПО в эксплуатации, устранения обнаруженных в нем ошибок, его доработки и модификации, а также извещения пользователей о внесенных в него изменениях.

Достоинства: на каждой стадии формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности; выполняемые в логичной последовательности стадии работ облегчают планирование сроков завершения всех работ и соответствующих затрат. Недостатки: позднее обнаружение проблем; выход из календарного графика, запаздывание с получением результатов; высокий риск создания системы, не удовлетворяющей изменившимся потребностям пользователей; избыточность документации; неравномерная нагрузка членов группы, работающей над проектом в ходе ЖЦ.

На самом деле невозможно двигаться строго поступательно, необходимо возвращаться, чтобы исправлять ошибки, сделанные на ранних стадиях, устранять недоделки, учитывать меняющиеся в ходе проекта требования. В этом кроется причина недостатков водопадной модели.

Формирование требований

Проектирование

Реализация

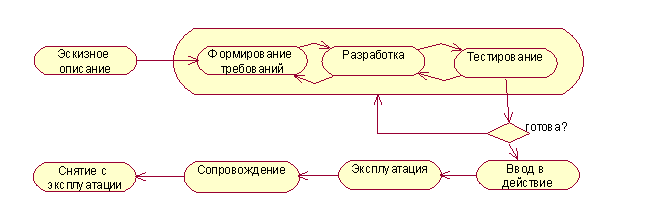
Тестирование

Ввод в действие

Эксплуатация и сопровождение

Снятие с эксплуатации

Особенности *эволюционной модели*: поэтапное уточнение требований к ПО с помощью прототипирования; параллельное осуществление анализа требований, разработки и верификации. Достоинства: полный учет требований заказчика, большее его участие в проекте; равномерная нагрузка на группу; раннее обнаружение проблем и их разрешение по мере возникновения. Недостатки: плохая документированность; запутанность создаваемого ПС и сложность внесения изменений; сложность планирования; необходимость специальных средств и технологий разработки ПС; годится лишь для небольших ПС (не более 50 Килострок).

**Схема эволюционной модели ЖЦ**

Формирование требований в виде формальных спецификаций

Формальное преобразование 1

. . .

Формальное преобразование N

Сборка и ввод в действие

Эксплуатация и сопровождение

Снятие с эксплуатации

Особенности *модели ЖЦ, основанной на формальных преобразованиях*: использование специальных нотаций для формального описания требований; кодирование и тестирование заменяется процессом предобразования формальной спецификации в исполняемую программу. Достоинства: формальные методы гарантируют соответствие ПО спецификациям требований к ПО, т. о. вопросы надежности и безопасности решаются сами собой. Недостатки: большие системы сложно описать формальными спецификациями; требуются специально подготовленные и высококвалифицированные разработчики; есть зависимость от средств разработки и нотации спецификаций.

Особенности *итерационных моделей*:

* процесс разработки разбивается на последовательность шагов, выполняемых циклически;
* модель напоминает несколько последовательных «каскадов»;
* разные виды деятельности не привязаны намертво к определенным этапам разработки, а выполняются по мере необходимости, иногда повторяются, до тех пор, пока не будет получен нужный результат;
* с каждой пройденной итерацией ПО наращивается, в него интегрируются новые разработанные компоненты.

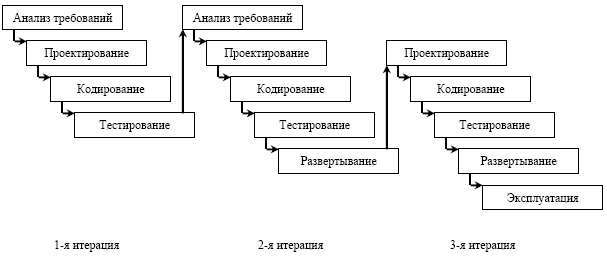


Рис. 3.1.3.-1 Схема пошаговой итерационной модели ЖЦ.

Особенности *итерационной* *спиральной модели*:

* общая структура действий на каждой итерации – планирование, определение задач, ограничений и вариантов решений, оценка предложенных решений и рисков, выполнение основных работ итерации и оценка их результатов;
* решение о начале новой итерации принимается на основе результатов предыдущей;
* досрочное прекращение проекта в случае обнаружения его нецелесообразности.

Достоинства итерационных моделей:

* полный учет требований заказчика, большее его участие в проекте;
* равномерная нагрузка на группу;
* раннее обнаружение проблем и их разрешение по мере возникновения, уменьшение рисков на каждой итерации.

Недостатки итерационных моделей: сложность планирования; плохая документированность создаваемого ПО.

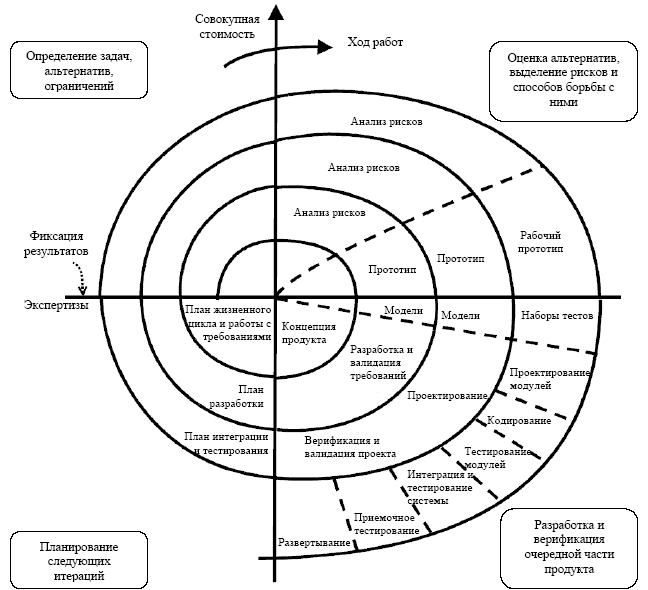


Рис. 3.1.3.-2 Схема спиральной модели ЖЦ

Проблемой современной программной инженерии являются «тяжелые» процессы. Характеристики *«тяжелого» процесса*:

1. необходимость документировать каждое действие разработчиков;
2. множество рабочих продуктов (в первую очередь - документов), создаваемых в бюрократической атмосфере;
3. отсутствие гибкости;
4. детерминированность (долгосрочное детальное планирование и предсказуемость всех видов деятельности, распределение человеческих ресурсов на длительный срок, охватывающий большую часть проекта.

Противоположностью «тяжелого» процесса является «*легковесный» процесс –* основа быстрой разработки ПО (agilesoftwaredevelopment). Быстрая разработка ориентируется на эффективную коммуникацию между разработчиками, высокую квалификацию разработчиков и другие факторы, позволяющие сократить расходы на «бюрократию». Принципы быстрой разработки:

1. Диалог «лицом к лицу» – самый эффективный способ обмена информацией.
2. Избыточная «тяжесть» технологии (дополнительные рабочие продукты, планы, диаграммы, документы) стоит дорого.
3. Более многочисленные команды требуют более «тяжелых» технологий.
4. Большая «тяжесть» процесса подходит для проектов с большей критичностью.
5. Возрастание обратной связи и коммуникации сокращает потребность в промежуточных продуктах.
6. Дисциплина, умение и понимание противостоят процессу, формальности и документированию.
7. Потеря эффективности в некритических видах деятельности вполне допустима.

Под критичностью понимаются масштабы последствий отказа разрабатываемого ПО. Уровни критичности:

* потеря удобства;
* потеря важных данных и/или рабочего времени;
* потеря невозместимых средств, дорогостоящего оборудования;
* потеря человеческой жизни.

Основные направления развития современной программной инженерии:

1. Управление требованиями
2. Управление конфигурацией и изменениями
3. Управление качеством ПО
4. Итерационная разработка ПО
5. Использование компонентной архитектуры (объектно-ориентированный подход)
6. Визуальное моделирование ПО