2. Современная система разработки программного обеспечения

Ничто так не ограничивает полет
фантазии программиста, как компилятор.
Программистский фольклор

В предыдущей главе мы познакомились с тем, каким образом компьютер можно привлечь к решению практических задач, возникающих в различных областях человеческой деятельности. При этом мы узнали, по какой схеме надлежит действовать, с чего начинать, когда и в каком качестве привлекать специалистов в предметной области, какие результаты от них требовать и как эти результаты использовать. Мы выяснили, что процесс применения вычислительной техники для решения реальных практических задач может быть до некоторой степени формализован.

У Вас не возникло никаких вопросов? Резонный вопрос – что мы понимаем под *реальной практической задачей*? И чем эта задача отличается от "нереальной" или "непрактической"? К примеру, мы встретили школьных друзей и после бурного наплыва ностальгических воспоминаний о любимой школе приняли решение реализовать программную систему для нахождения корней квадратных уравнений, взяв за основу мощный математический аппарат, изученный нами много лет назад. Будет ли это в точности то, что мы здесь и далее понимаем под реальной практической задачей?

Конечно, нет. По счастью, в России все еще есть довольно много людей, решающих квадратные уравнения если не в уме, то, по крайней мере, при помощи ручки и бумаги и без существенных временных затрат. А, следовательно, вряд ли наша программная система будет востребована. Никто не станет ее использовать ни в коммерческих, ни в исследовательских, ни в учебных целях. Почему это так? Вспомним, откуда появилась идея о написании программы? Вспомнили? Правильно! Основная предпосылка была: "давайте что-нибудь напишем".

Серьезные программы не пишут просто так. Обычно, до момента начала разработки имеется довольно точное представление, зачем нужна эта программа, и кто ей будет пользоваться. Таким образом, идея о написании программы не возникает сама по себе, ее порождает некоторая *задача*, для которой заранее очевидны выгоды от использования компьютера. Так, вряд ли Вы захотите вручную обсчитывать весь комплекс задач бухгалтерии крупного предприятия или вычислять орбиту, по которой должен двигаться спутник. Вот такие задачи можно смело отнести к *реальным* и уж безусловно к *практическим*. Итак, запомним: *сначала реальная задача, в которой требуется привлечение компьютера, потом программа*.

На последней стадии этого процесса в дело вступают современные системы разработки программного обеспечения. Рассмотрению их состава и функционального назначения посвящена данная глава.

О средствах разработки

Кто не хочет сделать – ищет оправдание,
кто хочет – ищет средство.
Народная мудрость

Взглянем под другим углом зрения на рассмотренную в предыдущей главе схему по использованию компьютеров в решении реальных задач. Основной вопрос: “Каких специалистов необходимо привлечь для успешного решения задач, возникающих на разных этапах деятельности?” Иначе говоря, кто будет анализировать предметную область, кто строить модель, кто создавать алгоритм и т.д.?

Заметим, что разработка программного обеспечения к настоящему моменту превратилась в обычный *технологический процесс*, на разных стадиях которого действуют подготовленные специалисты, применяющие в своей повседневной производственной практике различные технологии. Среди таких специализаций выделяются аналитики, маркетологи, менеджеры, кодировщики, тестировщики, специалисты по созданию документации и многие другие.

Отрасль, связанная с разработкой программного обеспечения, в которой работают авторы данной книги, а также, возможно, собираются работать ее читатели, очень молода. Действительно, ей всего лишь около 50 лет, но накопленный в ней объем знаний, технологических решений, методик и просто практических рекомендаций к действию воистину огромен! Рассмотреть весь процесс производства программного обеспечения в рамках одной книги невозможно, и мы не собираемся этого делать. В данной книге мы предлагаем нашим читателям познакомиться с тем, как пишется программный код, на каких принципах основан этот процесс, какие технологии применяются, в чем их содержательный смысл.

Вопросов, которые нам потребуется рассмотреть, достаточно много. С чего же начать? Остановитесь на секунду и подумайте, с чего бы начали Вы? Пока не знаете? Тогда давайте представим себе ситуацию: нас страшно заинтересовала задача нахождения среднего арифметического двух чисел. Наши действия: находим алгоритм ее решения, осознаем всю глубину заложенного в алгоритме математического аппарата, с торжествующим возгласом мчимся к компьютеру и… А что дальше? Начинаем писать программу? А как или, как говорят программисты, на чем (имея на самом деле ввиду на каком языке)? Оказывается, прежде нам понадобится решить для себя, *на каком языке программирования*[1] *мы будем писать программу*. Решение это в реальной ситуации зависит от многих факторов, в нашем же случае ответ содержится на обложке книги. В рамках данной книги изучается язык программирования Zonnon.

Хорошо, язык “общения” с компьютером мы выбрали[2]. Догадались, какой вопрос следующий? А где писать текст программы? Быть может великолепный текстовый редактор Microsoft Word, в котором Вы привыкли создавать такие красивые открытки для своих друзей и такие интересные рефераты по разным областям знания (как говорится: “Блажен, кто верует”) подходит для написания текста программы? Может быть и так, но профессиональные программисты почему-то программы в нем не пишут. Для этого используются… Впрочем, немного подождите.

Еще один вопрос: “А как превратить текст программы в нечто, что можно выполнить?” Здесь ответ уже далеко не тривиален, но, немного терпения, и мы преодолеем и это препятствие.

---

[1] расшифровка этого важного понятия будет приведена чуть ниже.

[2] на самом деле необходимость такого выбора должна вызывать у вдумчивого читателя вопрос: “А зачем?”. Разве не достаточно было создать один единственный язык для “общения” человека с компьютером? Ответ на этот вопрос мы немного отложим.

Перечень вопросов можно продолжить и, прочитав книгу до конца, Вы обнаружите их еще немало, однако пока нужно остановиться. Сказанного должно быть достаточно, чтобы подвести Вас к важной мысли: *для поддержки деятельности программиста необходимы специальные средства*[1].

Эта мысль на самом деле важна. Будет ли токарь работать без станка? Будут ли на лесопилке пилить дерево лобзиком или старинной двуручной пилой? Будут ли для переноса бетонных свай на стройке использовать десятки тысяч человек, как в Древнем Египте? Ответы очевидны. В любой отрасли существуют свои средства, упрощающие труд специалиста. Есть такие средства и в разработке программного обеспечения. Средства эти от года к году совершенствуются, причем, процесс их развития, как и всего остального в программной индустрии, необычайно стремителен.

В данной главе мы рассмотрим, какие именно средства помогают программистам в решении их профессиональных задач. При этом основное внимание будет уделено не конкретным образцам, а их классам (так, мы рассмотрим, что такое компилятор вообще, а не конкретный компилятор языка Zonnon). Готовы? Тогда за дело.

Основные средства разработки

Прежде всего, поговорим о тех средствах создания программ, без которых этот процесс вообще невозможен (точнее говоря практически невозможен) в настоящий момент времени.

Язык программирования высокого уровня

– Слушай, как тебе удалось выучить
английский язык всего за один день?
– Да ничего сложного, он очень похож на C++.
Программистский фольклор

Что значит написать программу? Написать программу – *реализовать* алгоритм, иначе говоря, представить его в виде понятных компьютеру указаний того, что необходимо делать. К сожалению, компьютеры не умеют понимать человека с полуслова, а это значит, что словесное описание алгоритма необходимо превратить в абсолютно точный набор инструкций, однозначно интерпретируемых машиной.

Представьте себе, что Ваш друг никогда не слышал про метод половинного деления. Представьте теперь, что Вам жизненно необходимо довести этот метод до его сведения. Какие средства для этого имеются в Вашем распоряжении?

Ну конечно, прежде всего, это великий и могучий русский язык. Вы просто объясняете другу суть метода, пользуясь некой математической терминологией и обиходными словами русского языка. Надеемся, что Вы хорошо изучили русский язык. Надеемся, что Вы в состоянии не только изъясняться на бытовом уровне, но и грамотно писать связный текст, рассказывать о чем-то и т.д. и т.п. Люди, считающие, что они освоили Microsoft Visual

---

[1] средства в данном случае имеются в виду программные, понятно что без компьютера как такового все остальное вообще бессмысленно.

Studio .Net, C++, C#, Object Pascal, Visual Java, Microsoft SQL Server, Windows, Linux и прочие умные слова, но, в то же время, не способные изъясняться на родном языке так, чтобы их понимали хотя бы коллеги, вряд ли смогут найти хорошую работу. А что, если друг англичанин (в наше время развития Интернет-технологий этот вариант более не представляется экзотическим)? Надеемся, что Вы активно изучаете также и английский язык, в противном случае Вас ожидают большие трудности в изучении технической документации, чтении электронной справки и т.д. К сожалению, большая часть информации, необходимой программистам, присутствует именно на английском языке. Если Вы более или менее владеете языком, то для Вас не составит особого труда рассказать по-английски, как работает метод.

Итак, основное средство передачи информации от одного человека к другому – некоторый понятный обоим язык общения.

А как объяснить что-либо машине? Увы, пока что для компьютера и русский, и английский, и суахили – одинаковая тарабарщина[1]. Поэтому, для того, чтобы что-то объяснить компьютеру, люди создали специальные языки. Классификация этих языков и история их возникновения частично затронуты во введении и подробно описаны в литературе [1]. Мы же здесь лишь отметим, что процесс написания программ за последние 50 лет прошел путь от программирования в инструкциях процессора (программирование в машинных кодах) через программирование на низкоуровневых языках (ассемблер) до программирования на *языках высокого уровня*. Вот на них мы и остановимся чуть подробнее.

Что такое язык программирования высокого уровня? Чем он отличается от естественного языка?

С формальной точки зрения *Язык программирования = Синтаксис + Семантика*.

Обратимся к литературе и посмотрим, как расшифровываются понятия *синтаксиса* и *семантики* для естественного языка:

- *Синтаксис* – раздел грамматики, изучающий внутреннюю структуру и общие свойства предложения [3].
- *Семантика* – раздел языкознания, изучающий значения единиц языка [3].

Для языков программирования справедливы следующие определения:

- *Синтаксис* – набор правил построения фраз алгоритмического языка, позволяющий определить осмысленные предложения в этом языке [2].
- *Семантика* – система правил истолкования отдельных языковых конструкций. Семантика определяет смысловое значение предложений алгоритмического языка [2].

Заметим, что определения достаточно похожи по своему смыслу. Действительно, язык программирования – искусственный (формальный) язык, предназначенный для записи алгоритмов [2]. Язык программирования задается своим *описанием* и реализуется в виде специальной программы: *компилятора* или *интерпретатора* [2].

Таким образом, если обычные (естественные) языки предназначены для общения людей между собой, то языки программирования – для общения программиста с компьютером.

Что же означает словосочетание “высокого уровня”? Чем языки программирования высокого уровня отличаются от языков “низкого уровня”? Быть может, кто-то всерьез считает, что языки “высокого уровня” – хорошие, а “низкого уровня” – плохие, неразвитые… Разумеется, это

[1] На заре развития вычислительной техники бытовало мнение, что через несколько лет компьютеры научатся понимать человеческий язык. Современные промышленные гиганты, такие как Microsoft, вкладывают большие средства в исследования в области распознавания речи и голосового управления, однако, до полноценного диалога или, даже, до полноценного голосового управления еще очень далеко.

заблуждение. Говоря об уровнях, мы ведем речь прежде всего о степени приближенности языка к машине. Уровень в данном случае – *уровень машинного восприятия*. Так, языки низкого уровня (ассемблер) по возможности приближены к машине, что делает соответствующие программы особенно эффективными с точки зрения их быстродействия. Однако, существенная проблема использования таких языков заключается в том, что программист – это прежде всего человек, и его способы восприятия информации весьма далеки от машинных, что чрезвычайно затрудняет написание программ на ассемблере. В настоящий момент на ассемблере реализуются сравнительно небольшие участки программного кода, связанные преимущественно с программированием аппаратуры (например, драйверы устройств). Подавляющее большинство программ пишется на том или ином языке программирования высокого уровня. Такие программы существенно ближе к восприятию человека, наделенного некоторыми профессиональными навыками – программиста. Следующая таблица иллюстрирует некоторые достоинства и недостатки языков программирования низкого и высокого уровня.

Свойство	**Программа на языке программирования низкого уровня**	**Программа на языке программирования высокого уровня**
Легкость создания	–	+
Понятность текста при беглом просмотре	–	+
Удобство отладки (поиска и исправления ошибок)	–	+
Удобство модификации	–	+
Удобство коллективной разработки	–	+
Быстродействие	+	–

К настоящему времени создано большое число разных языков программирования высокого уровня, однако реально используются лишь некоторые из них. К числу активно применяемых языков относятся C и C++, Pascal и Object Pascal, Fortran, Java, Basic (Visual Basic). Разрабатываются и используются новые языки, такие как C# и Zonnon. В данной книге рассматривается язык программирования высокого уровня *Zonnon*.

Транслятор (Интерпретатор, Компилятор)

Мыслим мы машинным кодом,
Это высший пилотаж.
Всех поздравит с Новым Годом
Самый новый вирус наш.
Песенка хакеров, "Компьютера"

Под *транслятором* (*translator*) обычно понимают специальную программу, которая переводит текст программы в последовательность машинных команд. Напомним еще раз: текст программы понятен человеку, набор команд понятен компьютеру.

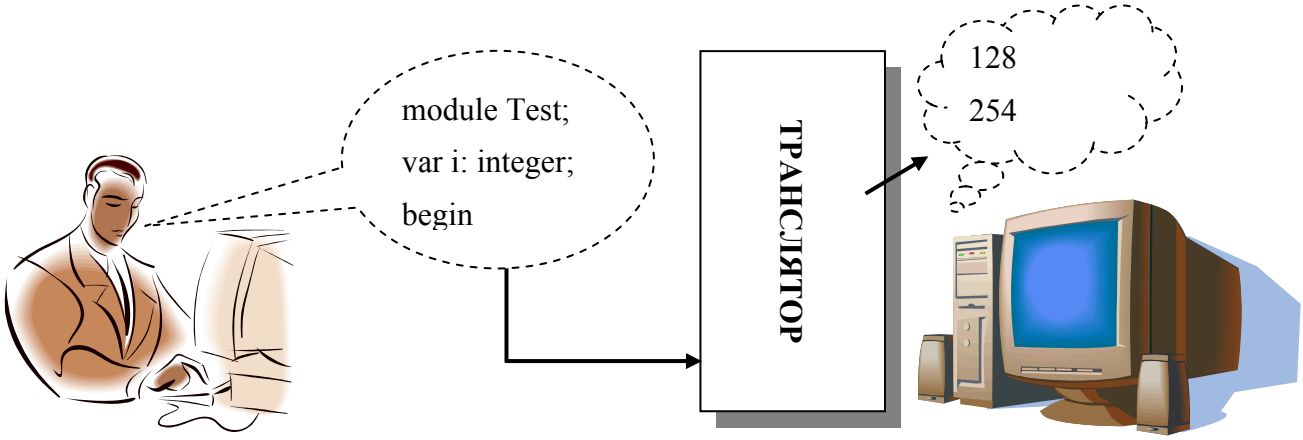

Рис 2.1. Роль транслятора в создании программ

Заметим, что трансляторы языков высокого уровня, таких как Zonnon, Pascal, C, Fortran и других, обычно называют *компиляторами* (*compiler*). Этим подчёркивается общепринятый для промышленных языков режим трансляции, при котором в начале осуществляется перевод программы в двоичное представление, а лишь затем программа передаётся на исполнение. Другой способ трансляции, называемый *интерпретация*, состоит в совмещении перевода и исполнения программы (в этом случае объектный модуль не сохраняется и его, соответственно, нельзя повторно использовать). Метод интерпретации используется при выполнении программ на языке Basic.

К числу основных достоинств компилируемых языков по сравнению с интерпретируемыми относятся:

- в компилируемых языках процесс построения (создания) исполняемого модуля выполняется один раз, а не при каждом запуске, что экономит время.
- в компилируемых языках обнаружение синтаксических ошибок происходит до запуска программы на выполнение, а не в его процессе.

Несмотря на очевидные недостатки интерпретируемых языков, они применяются в разных специфических задачах, а также в тех случаях, где простота программы важнее ее производительности (а программы на интерпретируемых языках почти всегда проще[1] своих аналогов на языках транслируемых).

Редактор связей (сборщик, динамический компоновщик, linker)

Я, ты, он, она – вместе целая страна.
Слова из песни

Уже в самом начале развития методов программирования стал применяться простой и эффективный приём выделения часто используемых алгоритмов в самостоятельные программы, получивших название стандартных *подпрограмм*. Примером могут служить подпрограммы вычисления элементарных функций (синус, косинус и др.), а также процедуры обмена с внешними устройствами компьютера. Однажды составленные и откомпилированные, они в дальнейшем могут применяться программистами в своих задачах путём подсоединения их к разработанному коду основного алгоритма. В более широком плане эта идея нашла своё выражение в технологии *модульного программирования*, которую мы будем рассматривать в нашей книге. В данный момент для нас важно обратить внимание на тот факт, что для

[1] простота здесь означает не то, что программа меньше “умеет”, а то, что она легче воспринимается (говорят “читается”) человеком.

обеспечения комплектации оттранслированной программы вспомогательными подпрограммами требуются специальные средства.

В систему программирования входит программа, называемая *редактор связей*, которая обеспечивает поиск вспомогательных подпрограмм в специальных библиотеках программ и их присоединение к основной программе пользователя. Результатом работы редактора связей является полностью готовый к исполнению двоичный код программы, называемый *загрузочным модулем*.

Загрузочный модуль может быть немедленно инициирован на исполнение, а может быть записан на диск и в дальнейшем многократно вызываться на исполнение с помощью специальной программы – загрузчика.

Отладчик (debugger)

А в конце обработать напильником.
Народная мудрость

В систему программирования входит также программа, облегчающая *отладку*, а точнее, *поиск ошибок*. При всём многообразии реализаций отладчиков их основные возможности заключаются в так называемой *трассировке* работы программы.

Трассировка – это отслеживание (ведение протокола) работы программы. В процессе трассировки программист может проследить порядок исполнения операторов, а также динамику изменения значений переменных программы.

В современных условиях отладка программ является не менее, а зачастую и более важным этапом разработки, чем собственно программирование (написание кода). Реальные задачи, пришедшие из разных областей человеческой деятельности, как правило, являются очень сложными. Объем программ, реализующих их решение весьма велик. Такой код обычно создается большим коллективом разработчиков, в связи с чем возникает много дополнительных проблем (в частности, необходимость поддерживать передачу информации между разными участниками проекта и согласовывать их деятельность). В конечном счете, сложность задач приводит к росту числа ошибок в программе. Известна старая шутка о том, что “любая программа содержит хотя бы одну ошибку”. Известно продолжение этой шутки: “если ошибок в программе не обнаружено, ищи ошибку в компиляторе”. В обоих положениях есть большая доля правды. На этапе разработки крайне редко удается полностью промоделировать реальную обстановку, в которой будет функционировать программа (представьте, например, как Вы создаете у себя тестовый полигон, размером с автомобильный завод), а также отследить все могущие возникнуть ситуации.

Все это, конечно, хорошо, но как же все-таки определить, есть в программе ошибка или нет? Кажется, что самый простой способ можно сформулировать так: “сейчас запустим и проверим!”. К сожалению, этот принцип применим лишь для очень ограниченного спектра задач. Конечно, если Вы создаете программную систему, которая должна нарисовать на экране тигра, Вы легко можете проверить ее работоспособность. Запускаете, смотрите – “да это же не тигр, это вовсе кролик!” Значит, вывод элементарен – не работает. Оставим на минуту вопрос о том, как теперь получить ответ на вопрос, почему не работает. Представим себе, что Вы пишете программу для расчета траектории движения спутника, который будет запущен в космос. Представили? “Сосчитали, попробуем запустить спутник. Ой, какой кошмар! Спутник улетел в неизвестном направлении…” Теперь поняли? Итак, необходимо уметь проверять работоспособность программы до внедрения ее в эксплуатацию. Если вдобавок к этому вернуться к вопросу о поиске причины возникновения обнаруженной ошибки, мы поймем, что диагностика и исправление ошибок в программах – важнейшая задача. В реальных программистских компаниях существует

целый штат сотрудников, которые занимаются этими проблемами (тестировщики, контролеры качества,…). Более того, теория в данной области не стоит на месте. Разработано несколько разных подходов к решению рассмотренных проблем. В нашей книге мы не будем подробно останавливаться на этих подходах, это предмет для серьезного разговора, заслуживающий отдельной книги.

Редактор кода

Мы начали рассмотрение системы программирования с её ключевых частей – транслятора и редактора связей, которые существовали с самого начала развития систем автоматизации программирования. А вот процесс составления программ долгое время оставался ручным. Программист записывал программу на специальном бланке, относил в отдел перфорации, где операторы с помощью специального оборудования наносили программу на перфокарты или перфоленты. С них программа загружалась в ЭВМ, запускалась, в ней обнаруживались ошибки, программист их исправлял, снова "набивал" перфокарты и так далее.

В настоящее время – время персональных компьютеров – этот рутинный процесс ушёл в прошлое. Современный программист, как правило, не пользуется бумагой для записи программ, а сразу заносит её текст в компьютер "из головы", пользуясь так называемыми *редакторами кода* (редакторами текстов) или *текстовыми процессорами*.

Редактор кода – это программная система, обеспечивающая первоначальную подготовку исходного текста программы и его исправление в процессе разработки. В отличие от универсальных текстовых процессоров (самым известным из них является уже упоминавшийся выше Microsoft Word) редакторы кода специализированы для работы именно с исходными текстами программ, поэтому они не имеют массы функций обычных редакторов (вроде работы с таблицами и рисунками), зато предоставляют другие функции не менее полезные. Существует довольно большое количество различных редакторов кода, список их возможностей также весьма обширен, начиная от простого набора текста, комбинирования отдельных фрагментов, поиска по образцу, выделения цветом различных элементов программы и заканчивая автоматическим форматированием в соответствии с устоявшимися правилами оформления кода для того или иного языка программирования (эти правила часто называют *стилем*).

Подготовленная с помощью редактора текстов программа запоминается в виде одного или нескольких файлов и в дальнейшем служит входной информацией для транслятора.

Дополнительные средства разработки

В предыдущем разделе были рассмотрены компоненты системы программирования, без которых невозможно обойтись. Здесь мы поговорим о некоторых других средствах, использование которые не является чем-то обязательным, но, в то же время, способно упростить процесс создания программ, сделать его более эффективным по разным критериям.

Средства автоматизированной генерации кода

По щучьему веленью, по моему хотенью,
Ступайте-ка сани домой сами.
Емеля

Помните, как, сидя в школе на уроке, Вы мечтали о том, чтобы какой-нибудь джин прилетел и сделал за Вас такую нудную контрольную работу? Или написал сочинение? Или нарисовал изометрическую проекцию? Или сделал что-нибудь еще, предоставив Вам возможность просто посмотреть в окно, помечтать…

Вы будете удивлены, но подобные мысли неоднократно приходили в голову авторам этой книги. Значит ли это, что мы лодыри, лентяи и тунеядцы? Надеемся, что нет. Просто, периодически бывают случаи, когда очевидно, что именно нужно сделать, но эта работа является скучной, неинтересной, особенно грустно, если подобная работа до того уже выполнялась много раз. Что же делать?

К счастью, в разработке программ существует возможность частично избавить себя от рутины с помощью так называемых *средств автоматизированной генерации кода*, в некоторых случаях они умеют самостоятельно создавать программный код, выполняющий определенные стандартные действия. Примером таких средств может служить, например, Microsoft Visual Studio .NET, которая автоматически создает и заполняет полями класс “Форма” при создании нами нового окна и наполнении его различными компонентами. Если в предыдущем предложении Вы поняли ровно половину, не беда. Постепенно мы со всем разберемся, сейчас же важно усвоить следующие: в некоторых случаях современные системы программирования способны автоматически создавать фрагменты текста программы.

Профилировщик (profiler)

– Ну вот, ваш прибор работает. С вас тысяча.
– А почему так много?
– Десять за то, что я его починил, остальное за то, что я знаю, где чинить.
Приписывается Эдисону

Представьте себе, что Вы в сотрудничестве с другими программистами создали некую программную систему. После установки ее на реальном объекте (завод, магазин, склад) выяснилось, что при увеличении объемов обрабатываемых данных Ваша программа работает слишком медленно. Вопрос: как узнать, почему это происходит. Где в коде те “узкие места”, на которые приходится основное время выполнения? Как повысить быстродействие?

Для того, чтобы получить ответ на этот вопрос, существуют специальные программы, называемые *профилировщиками*. Одним из лучших профилировщиков на настоящий момент является Intel® VTune™ Performance Analyzer. При помощи этой и других подобных программ Вы можете серьезно повысить производительность Вашей программы, внеся в нее необходимые изменения. Процесс использования профилировщиков описан, например, в [4].

Средства документирования

Если ничего не получается,
прочтите, наконец, инструкцию.
Программистский фольклор

Осознаете ли Вы необходимость создания документации к разрабатываемой программной системе? Считаете ли Вы, что каждый программист должен уметь создавать документацию и делать это параллельно с разработкой программы? Если сейчас Вы на оба вопроса ответили “Нет”, отнесем это на счет Вашей неопытности. Однако, если Вы не измените своего мнения в будущем, найти работу по нашей специальности Вам вряд ли удастся.

Необходимость в создании документации неизбежно возникает в любом технологическом процессе. Процесс разработки программного обеспечения не является исключением. На всех этапах этого процесса создается масса документов различной направленности. Это документы управленческие (инструкции, приказы), юридические (лицензии), постановочные (техническое задание, требования к системе), проектные (проект программного комплекса), описательные, предназначенные для конечного пользователя (руководство пользователя), и, наконец, внутренняя документация, описывающая, как создавалась система, какова ее архитектура, какие модули в ней есть, что в них находится и т.д. и т.п.

На первых порах Вам может показаться, что такое обилие документации есть следствие неизбежной тяги человека к бюрократии. На самом деле это не так. Вы, разумеется, не верите и скептически качаете головой. Попробуем Вас разубедить! Для этого приведем пару примеров.

Зададимся для начала одним, казалось бы, несложным вопросом: много ли программных продуктов Российского производства Вы знаете? Скорее всего, не очень. В чем же дело? Быть может, в России плохие программисты? Очевидно, это не так – наши студенты побеждают в международных олимпиадах по программированию, наши программисты находят применение своим силам во многих западных компаниях, более того, эти компании с большим удовольствием принимают их на работу. В чем же дело? Не претендуя на подробный анализ, укажем лишь одну причину (не самую главную, разумеется). Одним из факторов успешного коммерческого распространения чего-либо (в том числе и созданного программного обеспечения) является наличие хорошей инструкции. Хорошая инструкция – это не то, что напечатано 8-м шрифтом на папиросной бумаге. Это не то, что может понять только автор инструкции. Это не книга в 2000 страниц, глядя на которую будущему пользователю хочется побыстрее забыть про Ваш программный продукт. Хорошая инструкция – это то, при помощи чего можно быстро и самостоятельно научиться пользоваться приобретенной продукцией (в том числе и программным обеспечением). Как обстоит дело с написанием этих инструкций? Как правило, программисты любят писать программы, но не любят писать инструкции. Специально приглашаемые для этого люди неизбежно сталкиваются с проблемой: сначала кто-то должен обучить их самих пользоваться программой, досконально осветив все аспекты ее применения. Да и вообще, людей, способных грамотно создавать хорошие инструкции, крайне мало. В заключение первого примера отметим, что дополнительные трудности вызывает процесс перевода написанной инструкции (руководства пользователя) на разные иностранные языки (эффект “сломанного телефона” – один писал программу, другой – инструкцию, третий ее переводил).

Второй пример связан с созданием документации во время разработки программы (описания того, как устроена эта программа “изнутри”). Представьте себе, что у Вас есть коллектив программистов, который пишет программу, но не создает документацию. Представьте себе, что один из программистов уволился, а на его место был вынужденно принят на работу новый сотрудник. Как теперь он сможет разобраться, как устроено то, что ему надлежит доделывать и, возможно, в чем-то переделывать? Без наличия документации попытка разобраться в чужой программе во многом равносильна ее переписыванию с нуля.

Таким образом, создание документации – задача не менее важная, чем создание программного кода, и в ее решении нам помогают различные программные средства. Для примера сошлемся на одно из таких средств – “Система генерации проектной документации Rational SoDA” [5].

Понятие интегрированной среды разработки

В единстве наша сила!
Девиз любой команды

Интегрированная среда разработки (IDE, integrated development environment) – специальная программа, предоставляющая возможность удобной совместной работы с различными компонентами системы программирования.

В предыдущих разделах мы рассмотрели большое количество видов программ, входящих в систему программирования. Это и редакторы кода, и компиляторы, и сборщики, и отладчики, и многие другие. При первом же знакомстве со всеми этими программами становится понятно, что каждая из них может работать с разными начальными установками. Так, например, Вы можете настроить множество параметров для редактора кода: цвет фона, цвет шрифта, шрифт, размер символа табуляции и еще сотню разных характеристик. Для компилятора Вы можете указать, как Вы предпочитаете оптимизировать код: по скорости, по размеру, никак не оптимизировать, а также есть возможность управления многими другими параметрами. Аналогично обстоит дело практически со всеми составляющими системы программирования.

Теперь представьте себе, как Вы по очереди запускаете все эти программы с огромным количеством разных параметров в командной строке. Т.е. сначала Вы запускаете редактор кода, и пишите в нем программу. После этого Вы ее сохраняете, а затем закрываете редактор. Далее Вы запускаете компилятор, указав ему в командной строке файл с текстом программы и все необходимые настройки. Компилятор отработал и нашел 4 ошибки в строках 27, 31, 110 и 547. Вы снова запускаете текстовый редактор, открываете текст программы и в результате титанических усилий находите эти строки и ошибки в них. После этого Вы снова сохраняете программу, закрываете редактор и опять запускаете компилятор. В результате компилятор создает нечто (объектный код), на этот раз по счастью без синтаксических ошибок (это Вам повезло!). Теперь Вы запускаете сборщик, указывая ему в командной строке кучу параметров и тот самый объектный файл. Если Вам опять повезло и ошибок нет (это бывает отнюдь не всегда), то Вы, наконец-то, получаете исполняемый файл, запускаете его и… О ужас! Программа запустилась и “повисла”. Или не повисла, но вместо тигра нарисовала Вам слона. Или ничего не нарисовала, но сказала, что Ваше уравнение не имеет корней, хотя Ваши друзья математики с пеной у рта не далее чем вчера доказывали Вам, что решение точно есть! Что это значит? Это значит, что с семантикой что-то не то, иначе говоря, программа работает неправильно и ее необходимо отлаживать, искать ошибки. А как это делать? Ага, для этого у нас есть отладчик. Все закрываем, запускаем отладчик, передавая ему в командной строке кучу параметров, исходный файл (текст программы), исполняемый файл, отладчик запускается и… О ужас! Он говорит Вам, что Вы при компиляции и сборке не включили так называемую *отладочную информацию*, оптимизировав исполняемый файл по размеру, а значит, не сможете нормально его отлаживать. Что делать? Снова запускаем компилятор, сборщик, потом отладчик. И т.д. и т.п.

Вам понравился процесс? Авторы книги застали момент, когда процесс именно так и выглядел. Поверьте, это очень неудобно. Для устранения неудобств и повышения эффективности процесса разработки создатели систем программирования стали строить их в виде так называемых *Интегрированных сред*. Термин “интегрированная” в названии среды означает, что она включает в себя в качестве элементов все необходимые инструменты для выполнения полного цикла работ над программой: написания, компиляции, построения исполняемого модуля, запуска, отладки. Кроме того, интегрированные среды позволяют выполнять следующие операции:

- визуально (в диалоге) производить быструю настройку параметров каждого из компонентов системы программирования;
- сохранять разные системы настроек и загружать их по мере необходимости;
- нажатием нескольких клавиш или выбором соответствующих пунктов меню осуществлять запуск одного или сразу нескольких компонентов системы программирования, автоматизируя процесс передачи им необходимых параметров.

Так, в любой интегрированной среде исполняемый модуль из исходного текста программы можно получить нажатием пары кнопок на клавиатуре[1]. Единственный минус таких сред является прямым следствием их главного плюса – собрав "под одной крышей" большой набор инструментов, интегрированная среда сама становится весьма сложной программой. Однако время, потраченное на ее изучение с лихвой окупается в дальнейшем. И, наконец, еще один положительный момент – устройство большинства сред одинаково в концептуальном плане, различия наблюдаются лишь в комбинациях клавиш для того или иного действия да в названиях пунктов меню. Таким образом, освоив первую в своей жизни интегрированную среду, на все остальные Вы потратите в разы меньше времени.

Визуальные среды

Лучше один раз увидеть…
Пословица

Одно из последних достижений человеческой мысли в области разработки программного обеспечения – визуальные среды программирования (самые известные – Borland® Delphi™ с базовым языком Object Pascal и многоязыковая среда Microsoft® Visual Studio.NET). Их появление связано с двумя важными факторами. Первый уже упоминался выше в этой главе – стремление человека максимально автоматизировать собственный труд. Компьютеризация человеческой цивилизации становится всеобщей, компьютеры, "родившись" в стенах научных учреждений, выбрались оттуда в промышленность, проникли в сферу обслуживания и, наконец, прочно завоевали себе место рядом с человеком в его собственном доме. Аппетит, как известно, приходит во время еды – вот и люди, получив в свое распоряжение такого помощника, требуют от него все больше и больше возможностей. Неизбежно растет число необходимых программ. Обеспечить такой объем потребностей можно только одним способом – стандартизировав производство. Визуальные среды – еще один шаг на этом пути. Они содержат в себе заготовки, из которых можно собирать работоспособный скелет программ, дополняя его впоследствии необходимой функциональностью.

Второй фактор связан с тем, что современный пользователь в большинстве своем не станет работать с программой, которая не удовлетворяет его "чувство прекрасного". Говоря серьезно, сейчас при создании программ их внешнему виду уделяется не меньшее значение, чем внутреннему содержанию. Однако здесь разработчика поджидает серьезная дилемма. С одной стороны, чем больше новая программа визуально отличается от других, тем лучше – ее легче запомнить, она, что называется, "бросается в глаза". А с другой, нужно быть очень осторожным – чуть переборщишь с "рюшечками", и программа станет слишком сложной для восприятия. А талантливых дизайнеров среди программистов ничуть не больше, чем среди остальных профессий. Как же быть остальным "простым смертным"? Визуальные среды и тут приходят на помощь. На самом деле нетрудно понять, что внешний вид программы можно собрать (да-да именно собрать как в конструкторе) из некоторого количества стандартных элементов. И если Вы не чувствуете в себе таланта архитектора, то этот путь для Вас. Кстати, появление визуальных сред в середине 90-х годов прошлого века привело, в числе прочих эффектов, к взрывообразному росту числа программ, написанных программистами-одиночками. Некоторые из этих программ стали (вполне заслуженно) всемирно известными и используются массой людей.

---

[1] Конечно, не все так безоблачно. Дело ограничится парой кнопок только, если в программе не оказалось ошибок и если предварительно были сделаны необходимые настройки среды. Впрочем, справедливости ради надо отметить, что настроек по умолчанию обычно бывает достаточно.

Источники информации

1. Гради Буч. “Объектно-ориентированный анализ и проектирование с примерами приложений на C++”
2. http://www.glossary.ru/
3. Большая советская энциклопедия
4. Касперски Крис, “Техника оптимизации программ. Эффективное использование памяти”, – СПб.: БХВ-Петербург, 2003
5. Александр Новичков. “Система генерации проектной документации Rational SoDA” – http://www.citforum.ru/programming/digest/soda.shtml