Основные этапы развития программирования как науки.

Первый этап - «стихийное» программирование (период от момента появления первых вычислительных машин до середины 60-х годов XXв.) В этот период практически отсутствовали сформулированные технологии, и программирование фактически было искусством. Первые программы имели простейшую структуру. Они состояли из собственно программы на машинном языке и обрабатываемых ею данных (рис. 1.2). Сложность программ в машинных кодах ограничивалась способностью программиста одновременно мысленно отслеживать последовательность выполняемых операций и местонахождение данных при программировании.

Появление ассемблеров позволило вместо двоичных или 16-ричных кодов использовать символические имена данных и мнемоники кодов операций. В результате программы стали более «читаемыми».

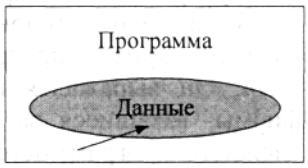


Рис. 1.2. Структура первых программ

Создание языков программирования высокого уровня, таких, как FORTRAN и ALGOL, существенно упростило программирование вычислений, снизив уровень детализации операций. Это, в свою очередь, позволило увеличить сложность программ.

Революционным было появление в языках средств, позволяющих оперировать подпрограммами.

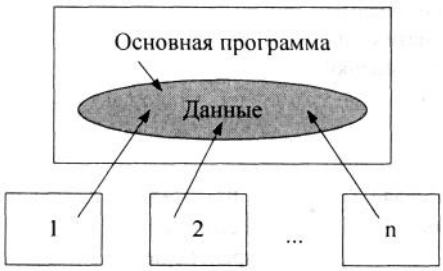


Рис. 1.3. Архитектура программы с глобальной областью данных

(Идея написания подпрограмм появилась гораздо раньше, но отсутствие средств поддержки в первых языковых средствах существенно снижало эффективность их применения.) Подпрограммы можно было сохранять и использовать в других программах. В результате были созданы огромные библиотеки расчетных и служебных подпрограмм, которые по мере надобности вызывались из разрабатываемой программы.

Типичная программа того времени состояла из основной программы, области *глобальных данных* и набора подпрограмм (в основном библиотечных), выполняющих обработку всех данных или их части (рис. 1.3). Слабым местом такой архитектуры было то, что при увеличении количества подпрограмм возрастала вероятность искажения части глобальных данных какой-либо подпрограммой. Например, подпрограмма поиска корней уравнения на заданном интервале по методу деления отрезка пополам меняет величину интервала. Если при выходе из подпрограммы не предусмотреть восстановления первоначального интервала, то в глобальной области окажется неверное значение интервала. Чтобы сократить количество таких ошибок, было предложено в подпрограммах размещать *локальные данные* (рис. 1.4).

Сложность разрабатываемого программного обеспечения при использовании подпрограмм с локальными данными по-прежнему ограничивалась возможностью программиста отслеживать процессы обработки данных, но уже на новом уровне. Однако появление средств поддержки подпрограмм позволило осуществлять разработку программного обеспечения нескольким программистам параллельно.

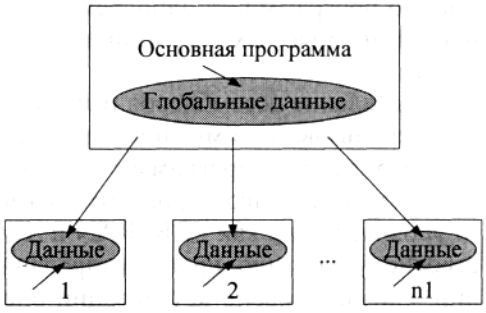


Рис. 1.4. Архитектура программы, использующей подпрограммы с локальными данными

В начале 60-х годов XX в. разразился «кризис программирования». Он выражался в том, что фирмы, взявшиеся за разработку сложного программного обеспечения, такого, как операционные системы, срывали все сроки завершения проектов.

Проект устаревал раньше, чем был готов к внедрению, увеличивалась его стоимость, и в результате многие проекты так никогда и не были завершены.

Объективно все это было вызвано несовершенством технологии программирования. Прежде всего стихийно использовалась разработка «снизу-вверх» - подход, при котором вначале проектировали и реализовывали сравнительно простые подпрограммы, из которых затем пытались построить сложную программу. В отсутствии четких моделей описания

подпрограмм и методов их проектирования создание каждой подпрограммы превращалось в непростую задачу, интерфейсы подпрограмм получались сложными, и при сборке программного продукта выявлялось большое количество ошибок согласования. Исправление таких ошибок, как правило, требовало серьезного изменения уже разработанных подпрограмм, что еще более осложняло ситуацию, так как при этом в программу часто вносились новые ошибки, которые также необходимо было исправлять... В конечном итоге процесс тестирования и отладки программ занимал более 80% времени разработки, если вообще когда-нибудь заканчивался. На повестке дня самым серьезным образом стоял вопрос разработки технологии создания сложных программных продуктов, снижающей вероятность ошибок проектирования.

Анализ причин возникновения большинства ошибок позволил сформулировать новый подход к программированию, который был назван «структурным».

Второй этап - структурный подход к программированию (60-70-е годы XXв.). *Структурный подход к программированию* представляет собой совокупность рекомендуемых технологических приемов, охватывающих выполнение всех этапов разработки программного обеспечения. В основе структурного подхода лежит *декомпозиция* (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших подпрограмм. С появлением других принципов декомпозиции (объектного, логического и т.д.) данный способ получил название *процедурной* декомпозиции.

В отличие от используемого ранее процедурного подхода к декомпозиции, структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры. Проектирование, таким образом, осуществлялось «сверху-вниз» и подразумевало реализацию общей идеи, обеспечивая проработку интерфейсов подпрограмм. Одновременно вводились ограничения на конструкции алгоритмов, рекомендовались формальные модели их описания, а также специальный метод проектирования алгоритмов - метод пошаговой детализации.

Поддержка принципов структурного программирования была заложена в основу так называемых *процедурных* языков программирования. Они включали основные «структурные» операторы передачи управления, поддерживали вложение подпрограмм, локализацию и ограничение области «видимости» данных (PL/1, ALGOL-68, Pascal, C).

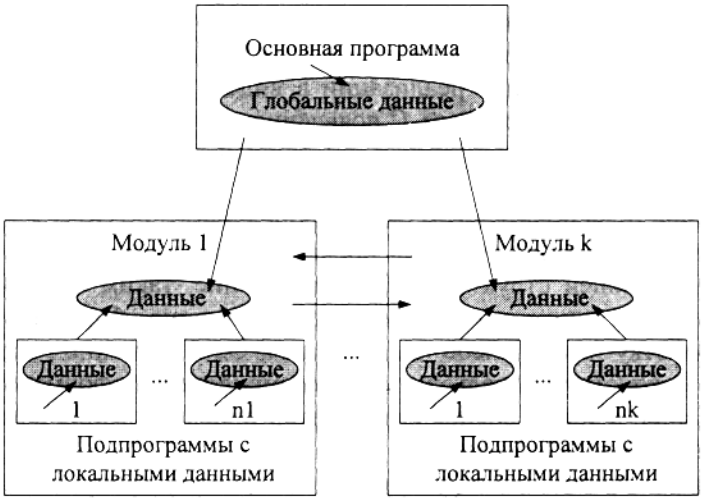
Одновременно со структурным программированием появилось огромное количество языков, базирующихся на других концепциях, но большинство из них не выдержало конкуренции. Какие-то языки были просто забыты, идеи других были в дальнейшем использованы в следующих версиях развиваемых языков.

Дальнейший рост сложности и размеров разрабатываемого ПО потребовал развития *структурирования данных.* В языках появляется возможность определения пользовательских типов данных. Одновременно усилилось стремление разграничить доступ к глобальным данным программы, чтобы уменьшить количество ошибок, возникающих при работе с глобальными данными. В результате появилась и начала развиваться технология модульного программирования.

Модульное программирование предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные в отдельно компилируемые *модули* (библиотеки подпрограмм), например, модуль графических ресурсов, модуль подпрограмм вывода на принтер (рис. 1.5). Связи между модулями при использовании данной технологии осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым «внутренним» переменным) запрещен. Эту технологию поддерживают современные версии языков Pascal и C (C++), языки Ада и Modula.

Использование модульного программирования существенно упростило разработку ПО несколькими программистами. Теперь каждый из них мог разрабатывать свои модули независимо, обеспечивая взаимодействие модулей через специально оговоренные межмодульные интерфейсы. Модули в дальнейшем без изменений можно было использовать в других разработках, что повысило производительность труда программистов.

Практика показала, что структурный подход в сочетании с модульным программированием позволяет получать достаточно надежные программы, размер которых *не превышает 100.000 операторов*.



Модули с локальными данными и попрограммами

Рис. 1.5. Архитектура программы, состоящей из модулей

Узким местом модульного программирования является то, что ошибка в интерфейсе при вызове подпрограммы выявляется только при выполнении программы (из-за раздельной компиляции модулей обнаружить эти ошибки раньше невозможно). При увеличении размера программы возрастает сложность межмодульных интерфейсов, и с некоторого момента предусмотреть взаимовлияние отдельных частей программы становится практически невозможно. Для разработки ПО большого объема было предложено использовать *объектный подход.*

Третий этап - объектный подход к программированию (с середины 80-х до конца 90-х годов XX в.). *Объектно-ориентированное программирование* определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности *объектов,* каждый из которых является экземпляром определенного типа *(класса),* а классы образуют иерархию с *наследованием* свойств. Взаимодействие программных объектов в такой системе осуществляется путем передачи сообщений (рис. 1.6).

Объектная структура программы впервые была использована в языке имитационного моделирования сложных систем Simula, появившемся еще в 60-х годах XX в. Естественный для языков моделирования способ представления программы получил развитие в другом специализированном языке моделирования - языке Smalltalk (70-е годы XX в.), а затем был использован в новых версиях универсальных языков программирования, таких, как Pascal, C++, Modula, Java.

Основным достоинством ООП по сравнению с модульным программированием является «более естественная» декомпозиция программного обеспечения, которая существенно облегчает его разработку. Это приводит к более полной локализации данных и интегрированию их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей (объектов) программы. Объектный подход предлагает новые способы организации программ, основанные на механизмах наследования, полиморфизма, композиции, наполнения. Эти механизмы позволяют конструировать сложные объекты из сравнительно простых. В результате существенно увеличился показатель повторного использования кодов и появляется возможность создания библиотек классов для различных применений.

Были созданы среды, поддерживающие *визуальное программирование* (Delphi, C++ Builder, Visual C++ и т.д.). При использовании визуальной среды у программиста появляется возможность проектировать некоторую часть, например, интерфейсы будущего продукта, с применением визуальных средств добавления и настройки специальных библиотечных компонентов. Результатом визуального проектирования является заготовка будущей программы, в которую уже внесены соответствующие коды.

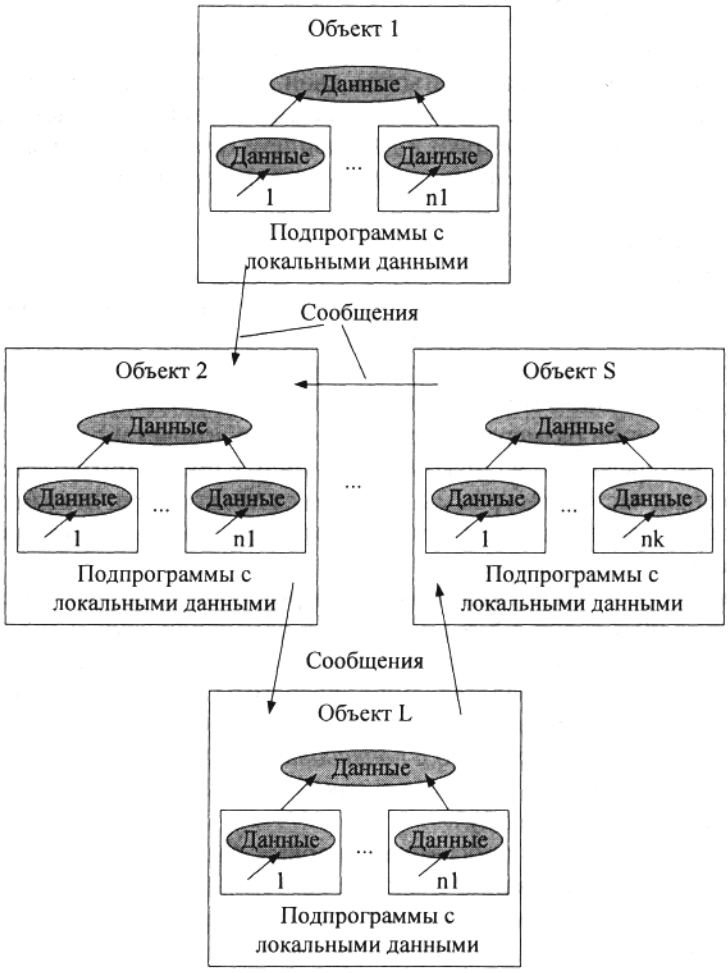


Рис. 1.6. Архитектура программы
при объектно-ориентированном программировании

Использование объектного подхода имеет много преимуществ, однако его конкретная реализация в объектно-ориентированных языках программирования, таких, как Pascal и C++, имеет существенные недостатки:

• фактически отсутствуют стандарты компоновки двоичных результатов компиляции объектов в единое целое даже в пределах одного языка программирования: компоновка объектов, полученных разными компиляторами C++ в лучшем случае проблематична, что приводит к необходимости разработки программного обеспечения с использованием средств и возможностей одного языка программирования высокого уровня и одного компилятора, а значит, требует наличия исходных кодов используемых библиотек классов;

• изменение реализации одного из программных объектов, как минимум, связано с перекомпиляцией соответствующего модуля и перекомпоновкой всего программного обеспечения, использующего данный объект.

Таким образом, при использовании этих языков программирования сохраняется зависимость модулей программного обеспечения от адресов экспортируемых полей и методов, а также структур и форматов данных. Эта зависимость объективна, так как модули должны взаимодействовать между собой, обращаясь к ресурсам друг друга. Связи модулей нельзя разорвать, но можно попробовать стандартизировать их взаимодействие, на чем и основан компонентный подход к программированию.

Четвертый этап – компонентный подход и CASE-технологии (с середины 90-х годов XX в. до нашего времени). *Компонентный подход* предполагает построение программного обеспечения из отдельных компонентов – физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через *стандартизованные двоичные интерфейсы.* В отличие от обычных объектов объекты-компоненты можно собрать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде (без исходных текстов) и использовать в любом языке программирования, поддерживающем соответствующую технологию. На сегодня рынок объектов стал реальностью, так в Интернете существуют узлы, предоставляющие большое количество компонентов, рекламой компонентов

забиты журналы. Это позволяет программистам создавать продукты, хотя бы частично состоящие из повторно использованных частей, т.е. использовать технологию, хорошо зарекомендовавшую себя в области проектирования аппаратуры.

Компонентный подход лежит в основе технологий, разработанных на базе COM (Component Object Model – компонентная модель объектов), и технологии создания распределенных приложений CORBA (Common Object Request Broker Architecture - общая архитектура с посредником обработки запросов объектов). Эти технологии используют сходные принципы и различаются лишь особенностями их реализации.

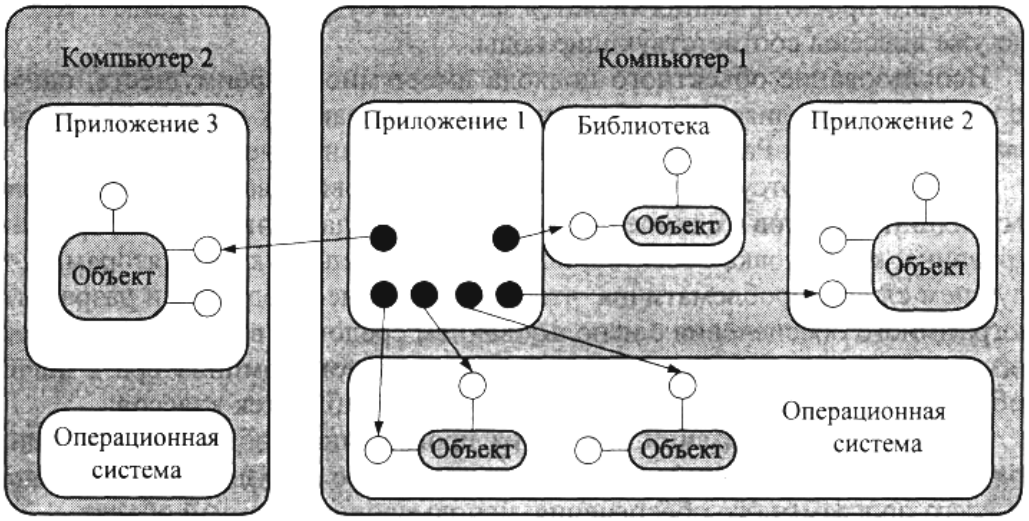


Рис. 1.7. Взаимодействие программных компонентов различных типов

Технология COM фирмы Microsoft является развитием технологии OLE I (Object Linking and Embedding – связывание и внедрение объектов), которая использовалась в ранних версиях Windows для создания составных документов. Технология COM определяет *общую парадигму взаимодействия программ любых типов:* библиотек, приложений, операционной системы, т. е. позволяет одной части программного обеспечения использовать функции *(службы),* предоставляемые другой, независимо от того, функционируют ли эти части в пределах одного процесса, в разных процессах на одном компьютере или на разных компьютерах (рис. 1.7). Модификация COM, обеспечивающая передачу вызовов между компьютерами, называется DCOM (Distributed COM – распределенная COM).

По технологии COM приложение предоставляет свои службы, используя специальные объекты – *объекты COM,* которые являются экземплярами *классов COM.* Объект COM так же, как обычный объект включает поля и методы, но в отличие от обычных объектов каждый объект COM может реализовывать несколько интерфейсов, обеспечивающих доступ к его полям и функциям. Это достигается за счет организации отдельной таблицы адресов методов для каждого интерфейса (по типу таблиц виртуальных методов). При этом интерфейс обычно объединяет несколько однотипных функций. Кроме того, классы COM поддерживают *наследование интерфейсов,* но не поддерживают *наследования реализации,* т. е. не наследуют код методов, хотя при необходимости объект класса-потомка может вызвать метод родителя.

Каждый интерфейс имеет имя, начинающееся с символа «I» и глобальный уникальный идентификатор IID (Interface IDentifier). Любой объект COM обязательно реализует интерфейс lUnknown (на схемах этот интерфейс всегда располагают сверху). Использование этого интерфейса позволяет получить доступ к остальным интерфейсам объекта.

Объект всегда функционирует в составе *сервера* – динамической библиотеки или исполняемого файла, которые обеспечивают функционирование объекта. Различают три типа серверов:

• внутренний сервер – реализуется динамическими библиотеками, которые подключаются к приложению-клиенту и работают в одном с ними адресном пространстве – наиболее эффективный сервер, кроме того, он не требует специальных средств;

• локальный сервер – создается отдельным процессом (модулем, exe), который работает на одном компьютере с клиентом;

• удаленный сервер – создается процессом, который работает на другом компьютере.

Например, Microsoft Word является локальным сервером. Он включает множество объектов, которые могут использоваться другими приложениями.

Для обращения к службам клиент должен получить указатель на соответствующий интерфейс. Перед первым обращением к объекту клиент посылает запрос к библиотеке COM, хранящей информацию обо всех, зарегистрированных в системе классах COM объектов, и передает ей имя класса, идентификатор интерфейса и тип сервера. Библиотека запускает необходимый сервер, создает требуемые объекты и возвращает указатели на объекты и интерфейсы. Получив указатели, клиент может вызывать необходимые функции объекта.

Взаимодействие клиента и сервера обеспечивается базовыми механизмами COM или DCOM, поэтому клиенту безразлично местонахождение объекта. При использовании локальных и удаленных серверов в адресном пространстве клиента создается *proxy-объект* – заместитель объекта COM, а в адресном пространстве сервера COM – *заглушка,* соответствующая клиенту. Получив задание от клиента, заместитель упаковывает его параметры и, используя службы операционной системы, передает вызов заглушке. Заглушка распаковывает задание и передает его объекту COM. Результат возвращается клиенту в обратном порядке.

На базе технологии COM и ее распределенной версии DCOM были разработаны компонентные технологии, решающие различные задачи разработки программного обеспечения.

OLE-automation или просто Automation (автоматизация) – технология создания программируемых приложений, обеспечивающая программируемый доступ к внутренним службам этих приложений. Вводит понятие *диспинтерфейса* (dispinterface) – специального интерфейса, облегчающего вызов функций объекта. Эту технологию поддерживает, например, Microsoft Excel, предоставляя другим приложениям свои службы.

ActiveX – технология, построенная на базе OLE-automation, предназначена для создания программного обеспечения как сосредоточенного на одном компьютере, так и распределенного в сети. Предполагает использование визуального программирования для создания компонентов – элементов управления ActiveX. Полученные таким образом элементы управления можно устанавливать на компьютер дистанционно с удаленного сервера, причем устанавливаемый код зависит от используемой операционной системы. Это позволяет применять элементы управления ActiveX в клиентских частях приложений Интернет.

Основными преимуществами технологии ActiveX, обеспечивающими ей широкое распространение, являются:

• быстрое написание программного кода – поскольку все действия, связанные с организацией взаимодействия сервера и клиента берет на программное обеспечение COM, программирование сетевых приложений становится похожим на программирование для отдельного компьютера;

• открытость и мобильность – спецификации технологии недавно были переданы в Open Group как основа открытого стандарта;

• возможность написания приложений с использованием знакомых средств разработки, например, Visual Basic, Visual C++, Borland Delphi, Borland C++ и любых средств разработки на Java;

• большое количество уже существующих бесплатных программных элементов ActiveX (к тому же, практически любой программный компонент OLE совместим с технологиями ActiveX и может применяться без модификаций в сетевых приложениях);

• стандартность - технология ActiveX основана на широко используемых стандартах Internet (TCP/IP, HTML, Java), с одной стороны, и стандартах, введенных в свое время Microsoft и необходимых для сохранения совместимости (COM, OLE).

MTS (Microsoft Transaction Server – сервер управления транзакциями) -технология, обеспечивающая безопасность и стабильную работу распределенных приложений при больших объемах передаваемых данных.

MIDAS (Multitier Distributed Application Server - сервер многозвенных распределенных приложений) – технология, организующая доступ к данным разных компьютеров с учетом балансировки нагрузки сети.

Все указанные технологии реализуют компонентный подход, заложенный в COM. Так, с точки зрения COM элемент управления ActiveX – внутренний сервер, поддерживающий технологию OLE-automation. Для программиста же элемент ActiveX – «черный ящик», обладающий свойствами, методами и событиями, который можно использовать как строительный блок при создании приложений.

Технология CORBA, разработанная группой компаний OMG (Object Management Group – группа внедрения объектной технологии программирования), реализует подход, аналогичный COM, на базе объектов и интерфейсов CORBA. Программное ядро CORBA реализовано для всех основных аппаратных и программных платформ и потому эту технологию можно использовать для создания распределенного программного обеспечения в гетерогенной (разнородной) вычислительной среде. Организация взаимодействия между объектами клиента и сервера в CORBA осуществляется с помощью специального посредника, названного VisiBroker, и другого специализированного программного обеспечения.

Отличительной особенностью современного этапа развития технологии программирования, кроме изменения подхода, является создание и внедрение автоматизированных технологий разработки и сопровождения программного обеспечения - CASE-технологии (Computer-Aided Software/System Engineering - разработка программного обеспечения/программных систем с использованием компьютерной поддержки). Существуют CASE-технологии, поддерживающие структурный и объектный (в том числе и компонентный) подходы к программированию.

Появление нового подхода не означает, что все ПО будет создаваться из программных компонентов, но анализ существующих проблем разработки сложного ПО показывает, что он будет применяться достаточно широко.

1.2. Проблемы разработки сложных программных систем

Большинство современных программных систем объективно очень сложны. Эта сложность обуславливается многими причинами, главной из которых является *логическая сложность решаемых* ими *задач.*

Пока вычислительных установок было мало, и их возможности были ограничены, ЭВМ применяли в очень узких областях науки и техники, причем, в первую очередь, там, где решаемые задачи были хорошо детерминированы и требовали значительных вычислений. В наше время, когда созданы мощные компьютерные сети, появилась возможность переложить на них решение сложных ресурсоемких задач, о компьютеризации которых раньше никто и не думал. Сейчас в процесс компьютеризации вовлекаются совершенно новые предметные области, а для уже освоенных областей усложняются уже сложившиеся постановки задач.

Дополнительными факторами, увеличивающими сложность разработки программных систем, являются:

- сложность формального определения требований к программным системам;
- отсутствие удовлетворительных средств описания поведения дискретных систем с большим числом состояний при недетерминированной последовательности входных воздействий;
- коллективная разработка;
- необходимость увеличения степени повторяемости кодов.

Сложность определения требований к программным системам. Сложность определения требований к программным системам обусловливается двумя факторами. Во-первых, при определении требований необходимо учесть большое количество различных факторов. Во-вторых, разработчики программных систем не являются специалистами в автоматизируемых предметных областях, а специалисты в предметной области, как правило, не могут сформулировать проблему в нужном ракурсе.

Отсутствие удовлетворительных средств формального описания поведения дискретных систем. В процессе создания программных систем используют языки сравнительно низкого уровня. Это приводит к ранней детализации операций в процессе создания программного обеспечения и увеличивает объем описаний разрабатываемых продуктов, который, как правило, превышает сотни тысяч операторов языка программирования. Средств же, позволяющих детально описывать *поведение* сложных дискретных систем на более высоком уровне, чем универсальный язык программирования, не существует.

Коллективная разработка. Из-за больших объемов проектов разработка программного обеспечения ведется коллективом специалистов. Работая в коллективе, отдельные специалисты должны взаимодействовать друг с другом, обеспечивая целостность проекта, что при отсутствии удовлетворительных средств описания поведения сложных систем, упоминавшемся выше, достаточно сложно. Причем, чем больше коллектив разработчиков, тем сложнее организовать процесс работы.

Необходимость увеличения степени повторяемости кодов. На сложность разрабатываемого программного продукта влияет и то, что для увеличения производительности труда компании стремятся к созданию библиотек компонентов, которые можно было бы использовать в дальнейших разработках. Однако в этом случае компоненты приходится делать более универсальными, что в конечном итоге увеличивает сложность разработки.

Вместе взятые, эти факторы существенно увеличивают сложность процесса разработки. Однако очевидно, что все они напрямую связаны со сложностью объекта разработки - программной системы.

1.3. Блочно-иерархический подход к созданию сложных систем

Практика показывает, что подавляющее большинство сложных систем как в природе, так и в технике имеет иерархическую внутреннюю структуру. Это связано с тем, что обычно связи элементов сложных систем различны как по типу, так и по силе, что и позволяет рассматривать эти системы как некоторую *совокупность взаимозависимых подсистем.* Внутренние связи элементов таких подсистем сильнее, чем связи между подсистемами. Например, компьютер состоит из процессора, памяти и внешних устройств, а Солнечная система включает Солнце и планеты, вращающиеся вокруг него.

В свою очередь, используя то же различие связей, можно каждую подсистему разделить на подсистемы и т.д. до самого нижнего «элементарного» уровня, причем выбор уровня, компоненты которого следует считать элементарными, остается за исследователем. На элементарном уровне система, как правило, состоит из немногих типов подсистем, по-разному скомбинированных и организованных. Иерархии такого типа получили название «целое-часть».

Поведение системы в целом обычно оказывается сложнее поведения отдельных частей, причем из-за более сильных внутренних связей особенности системы в основном обусловлены отношениями между ее частями, а не частями как таковыми.

В природе существует еще один вид иерархии – иерархия «простое-сложное» или иерархия развития (усложнения) систем в процессе эволюции. В этой иерархии любая функционирующая система является результатом развития более простой системы. Именно данный вид иерархии реализуется механизмом наследования объектно-ориентированного программирования.

Будучи в значительной степени отражением природных и технических систем, программные системы обычно являются иерархическими, т. е. обладают описанными выше свойствами. На этих свойствах иерархических систем строится *блочно-иерархический подход* к их исследованию или созданию. Этот подход предполагает сначала создавать части таких объектов (блоки, модули), а затем собирать из них сам объект.

Процесс разбиения сложного объекта на сравнительно независимые части получил название *декомпозиции.* При декомпозиции учитывают, что связи между отдельными частями должны быть слабее, чем связи элементов внутри частей. Кроме того, чтобы из полученных частей можно было собрать разрабатываемый объект, в процессе декомпозиции необходимо определить все виды связей частей между собой.

При создании очень сложных объектов процесс декомпозиции выполняется многократно: каждый блок, в свою очередь, декомпозируют на части, пока не получают блоки, которые сравнительно легко разработать. Данный метод разработки получил название *пошаговой детализации.*

Существенно и то, что в процессе декомпозиции стараются выделить аналогичные блоки, которые можно было бы разрабатывать на общей основе. Таким образом, как уже упоминалось выше, обеспечивают увеличение степени повторяемости кодов и, соответственно, снижение стоимости разработки.

Результат декомпозиции обычно представляют в виде схемы *иерархии,* на нижнем уровне которой располагают сравнительно простые блоки, а на верхнем – объект, подлежащий разработке. На каждом иерархическом уровне описание блоков выполняют с определенной степенью детализации, *абстрагируясъ* от несущественных деталей. Следовательно, для каждого уровня используют свои формы документации и свои модели, отражающие сущность процессов, выполняемых каждым блоком. Так для объекта в целом, как правило, удается сформулировать лишь самые общие требования, а блоки нижнего уровня должны быть специфицированы так, чтобы из них действительно можно было собрать работающий

объект. Другими словами, *чем больше блок, тем более абстрактным должно быть его описание* (рис. 1.8).

При соблюдении этого принципа разработчик сохраняет возможность осмысления проекта и, следовательно, может принимать наиболее правильные решения на каждом этапе, что называют *локальной оптимизацией* (в отличие от глобальной оптимизации характеристик объектов, которая для действительно сложных объектов не всегда возможна).

Примечание. Следует иметь в виду, что понятие сложного объекта по мере совершенствования технологий изменяется, и то, что было сложным вчера, не обязательно останется сложным завтра.

Итак, в основе блочно-иерархического подхода лежат декомпозиция и иерархическое упорядочение. Важную роль играют также следующие принципы:

• непротиворечивость - контроль согласованности элементов между собой;

• полнота - контроль на присутствие лишних элементов;

• формализация – строгость методического подхода;

• повторяемость – необходимость выделения одинаковых блоков для удешевления и ускорения разработки;

• локальная оптимизация – оптимизация в пределах уровня иерархии. Совокупность языков моделей, постановок задач, методов описаний некоторого иерархического уровня принято называть *уровнем проектирования.*

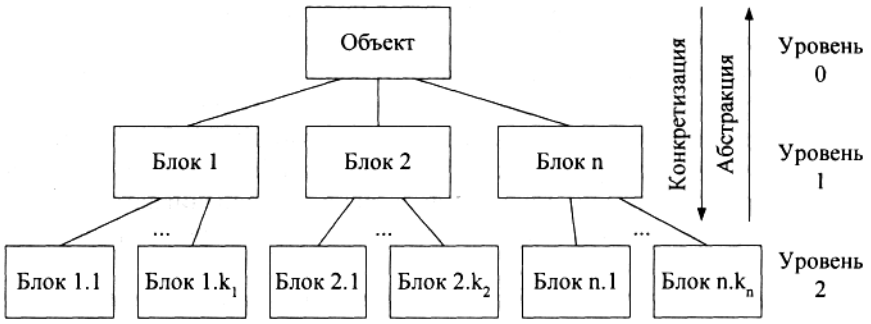


Рис. 1.8. Соотношение абстрактного и конкретного в описании блоков при блочно-иерархическом подходе

Каждый объект в процессе проектирования, как правило, приходится рассматривать с нескольких сторон. Различные взгляды на объект проектирования принято называть *аспектами проектирования.*

Помимо того, что использование блочно-иерархического подхода делает возможным создание сложных систем, он также:

• упрощает проверку работоспособности, как системы в целом, так и отдельных блоков;

• обеспечивает возможность модернизации систем, например, замены ненадежных блоков с сохранением их интерфейсов.

Необходимо отметить, что использование блочно-иерархического подхода применительно к программным системам стало возможным только после конкретизации общих положений подхода и внесения некоторых изменений в процесс проектирования. При этом структурный подход учитывает только свойства иерархии «целое-часть», а объектный – использует еще и свойства иерархии «простое-сложное».

1.7. Оценка качества процессов создания программного обеспечения

Как уже упоминалось выше, текущий период на рынке программного обеспечения характеризуется переходом от штучного ремесленного производства программных продуктов к их промышленному созданию. Соответственно возросли требования к качеству разрабатываемого программного обеспечения, что требует совершенствования процессов их разработки. На настоящий момент существует несколько стандартов, связанных с оценкой качества этих процессов, которое обеспечивает организация-разработчик. К наиболее известным относят:

• международные стандарты серии ISO 9000 (ISO 9000 - ISO 9004):

• CMM – Capability Maturity Model – модель зрелости (совершенствования) процессов создания программного обеспечения, предложенная SEI (Software Engineering Institute – институт программирования при университете Карнеги-Меллон);

• рабочая версия международного стандарта ISO 1EC 15504: Information Technology – Software Process Assessment: эта версия более известна под названием SPICE – (Software Process Improvement and Capability dEtermination – определение возможностей и улучшение процесса создания программного обеспечения).

Серия стандартов ISO 9000. В серии ISO 9000 сформулированы необходимые условия для достижения некоторого минимального уровня организации процесса, но не дается никаких рекомендаций по дальнейшему совершенствованию процессов.

CMM. CMM представляет собой совокупность критериев оценки зрелости организации-разработчика и рецептов улучшения существующих процессов.

Примечание. Изначально CMM разрабатывалась и развивалась как методика, позволяющая крупным правительственным организациям США выбирать наилучших поставщиков программного обеспечения. Для этого предполагалось создать исчерпывающее описание способов оценки процессов разработки программного обеспечения и методики их дальнейшего усовершенствования. В итоге авторы смогли добиться такой степени подробности и детализации, что стандарт оказался пригодным и для обычных компаний-разработчиков, желающих качественно улучшить существующие процессы разработки, привести их к определенным стандартам.

CMM определяет пять уровней зрелости организаций-разработчиков, причем каждый следующий уровень включает в себя все ключевые характеристики предыдущих.

1. *Начальный уровень* (initial level) – описан в стандарте в качестве основы для сравнения со следующими уровнями. На предприятии такого уровня организации не существует стабильных условий для создания качественного программного обеспечения. Результат любого проекта целиком и полностью зависит от личных качеств менеджера и опыта программистов, причем успех в одном проекте может быть повторен только в случае назначения тех же менеджеров и программистов на следующий проект. Более того, если эти менеджеры или программисты уходят с предприятия, то резко снижается качество производимых программных продуктов. В стрессовых ситуациях процесс разработки сводится к написанию кода и его минимальному тестированию.
2. *Повторяемый уровень* (repeatable level) – на предприятии внедрены *технологии управления проектами.* При этом планирование и управление проектами основывается на накопленном опыте, существуют *стандарты* на разрабатываемое программное обеспечение (причем обеспечивается следование этим стандартам) и специальная *группа обеспечения качества.* В случае необходимости организация может взаимодействовать с субподрядчиками. В критических условиях процесс имеет тенденцию скатываться на начальный уровень.
3. *Определенный уровень* (defined level) – характеризуется тем, что стандартный процесс создания и сопровождения программного обеспечения полностью документирован (включая и разработку ПО, и управление проектами). Подразумевается, что в процессе стандартизации происходит переход на наиболее эффективные практики и технологии. Для создания и поддержания подобного стандарта в организации должна быть создана специальная группа. Наконец, обязательным условием для достижения данного уровня является наличие на предприятии программы постоянного *повышения квалификации и обучения сотрудников.* Начиная с этого уровня, организация перестает зависеть от качеств конкретных разработчиков, и процесс не имеет тенденции скатываться на уровень ниже в стрессовых ситуациях.
4. *Управляемый уровень* (managed level) – в организации устанавливаются *количественные показатели качества* как на программные продукты, так и на процесс в целом. Таким образом, более совершенное управление проектами достигается за счет уменьшения отклонений различных показателей проекта. При этом осмысленные вариации в производительности процесса можно отличить от случайных вариаций (шума), особенно в хорошо освоенных областях.
5. *Оптимизирующий уровень* (optimizing level) – характеризуется тем, что мероприятия по улучшению применяются не только к существующим процессам, но и для оценки эффективности ввода новых технологий. Основной задачей всей организации на этом уровне является *постоянное улучшение* существующих процессов. При этом улучшение

процессов в идеале должно помогать предупреждать возможные ошибки или дефекты. Кроме того, должны вестись работы по уменьшению стоимости разработки программного обеспечения, например с помощью создания и повторного использования компонентов.

Сертификационная оценка соответствия всех ключевых областей проводится по 10-балльной шкале. Для успешной квалификации данной ключевой области необходимо набрать не менее 6 баллов. Оценка ключевой области осуществляется по следующим показателям:

• заинтересованность руководства в данной области, например, планируется ли практическое внедрение данной ключевой области, существует ли понимание у руководства необходимости данной области и т.д.;

• насколько широко данная область применяется в организации, например, оценке в 4 балла соответствует фрагментарное применение;

• успешность использования данной области на практике, например, оценке в 0 баллов соответствует полное отсутствие какого-либо эффекта, а оценка в 8 баллов выставляется при наличии систематического и измеримого положительного результата практически во всей организации.

В принципе, можно сертифицировать только один процесс или подразделение организации, например, подразделение разработки программного обеспечения компании IBM сертифицировано на пятый уровень. Кстати, в мире существует совсем немного компаний, которые могут похвастаться наличием у них пятого уровня CMM хотя бы в одном из подразделений – таких всего около 50-ти. С другой стороны, насчитывается несколько тысяч компаний, сертифицированных по третьему или четвертому уровням, т.е. существует колоссальный разрыв между оптимизированным уровнем зрелости и предыдущими уровнями. Однако еще больший разрыв наблюдается между количеством организаций начального уровня и числом их более продвинутых собратьев – по некоторым оценкам, свыше 70 % всех компаний-разработчиков находится на первом уровне CMM [3].

SPICE. Стандарт SPICE унаследовал многие черты более ранних стандартов, в том числе и уже упоминавшихся ISO 9001 и CMM. Больше всего SPICE напоминает CMM. Точно так же, как и в CMM, основной задачей организации является постоянное улучшение процесса разработки программного обеспечения. Кроме того, в SPICE тоже используется схема с различными уровнями возможностей (в SPICE определено 6 различных уровней), но эти уровни применяются не только к организации в целом, но и к отдельно взятым процессам.

В основе стандарта лежит *оценка процессов.* Эта оценка выполняется путем сравнения процесса разработки программного обеспечения, существующего в данной организации, с описанной в стандарте моделью. Анализ результатов, полученных на этом этапе, помогает определить сильные и слабые стороны процесса, а также внутренние риски, присущие данному процессу. Это помогает оценить эффективность процессов, определить причины ухудшения качества и связанные с этим издержки во времени или стоимости.

Затем выполняется *определение возможностей процесса,* т.е. возможностей его улучшения. В результате в организации может появиться понимание необходимости *улучшения* того или иного *процесса.* К этому моменту цели совершенствования процесса уже четко сформулированы и остается только техническая реализация поставленных задач. После этого весь цикл работ начинается сначала.

Совершенствование процессов жизненного цикла ПО необходимо.

Использование формальных моделей и методов позволяет создавать понятные, непротиворечивые спецификации на разрабатываемое программное обеспечение. Конечно, внедрение таких методов имеет смысл, хотя оно весьма дорого и трудоемко, а возможности их применения весьма ограничены. Основная же проблема – проблема сложности разрабатываемого программного обеспечения с совершенствованием процессов разработки пока не разрешена. Создание программного обеспечения по-прежнему предъявляет повышенные требования к квалификации тех, кто этим занимается: проектировщикам программного обеспечения и непосредственно программистам.