1 Проблемы разработки сложных программных систем.

Рассматривается понятие сложной программы и отличия сложных программ от простых. Приводятся основные проблемы разработки сложных программ. В приложении к программной инженерии формулируются основные принципы работы со сложными системами, применимые к широкому кругу задач.

Программы «большие» и «маленькие»

Основная тема данного курса — методы разработки «больших» и сложных программ.

Каждый человек хоть раз написавший какую-либо программу, достаточно хорошо может представить себе, как разработать «небольшую» программу, решающую обычно одну конкретную несложную задачу и предназначенную, чаще всего, для использования одним человеком или узкой группой людей.

Примером может служить программа, вычисляющая достаточно много (но не слишком, не больше 30000) знаков числа π.

Воспользуемся следующими формулами.

arctan(x) = x – x3/3 + x5/5 – x7/7 + … + (-1)nx2n+1/(2n+1) + O(x2n+3)

π/4 = arctan(1) = 4*arctan(1/5) – arctan(1/239)

Соответсвующая программа на языке Java может выглядеть примерно так.

```
public class PiCalculator
{
//Позволяет при вычислениях с повышенной точностью умножать и делить на числа
// <= 42949 = ( 2^32 mod CLUSTER_SIZE )
//Эта константа должна быть степенью 10 для простоты представления чисел.
private final static long CLUSTER_SIZE = 100000;
//Определенное значение этого поля позволяет сосчитать
// numberOfClusters * lg( CLUSTER_SIZE )
//точных цифр.
private static int numberOfClusters;
private static void print(long a[])
{
for(int i = 0; i < numberOfClusters + 1; i++)
{
if (i == 0) System.out.print("" + a[i] + '.');
else
{
StringBuffer s = new StringBuffer();
long z = CLUSTER_SIZE/10;
while(z > 0)
{
if (z > a[i]) { s.append(0); z /= 10; }
else break;
} if (z != 0) s.append(a[i]);
System.out.print(s);
}
System.out.println();
}
private static void lndiv(long a[], int n)
{
for(int i = 0; i < numberOfClusters + 1; i++)
{
if (i != numberOfClusters)
{
a[i+1] += (a[i]%n)*CLUSTER_SIZE;
a[i] /= n;
```

```
} else a[i] /= n;
}
}
private static void lnadd(long a[], long b[])
{
for(int i = numberOfClusters; i >= 0; i--)
{
if (i != 0)
{
a[i-1] += (a[i] + b[i])/CLUSTER_SIZE;
a[i] = (a[i] + b[i])%CLUSTER_SIZE;
}
else
a[i] = (a[i] + b[i])%CLUSTER_SIZE;
}
}
private static void lnsub(long a[], long b[])
{
for(int i = numberOfClusters; i >= 0; i--)
{
if (i != 0)
{
if (a[i] < b[i]) { b[i-1]++; a[i] += CLUSTER_SIZE; }
a[i] -= b[i];
} else
a[i] -= b[i];
}
}
public static void main (String[] args)
{
int i, j, numberOfDigits = 100, numberOfSteps;
if (args.length > 0) numberOfDigits = Integer.parseInt(args[0]);
numberOfSteps = (int)(((numberOfDigits + 1)/(Math.log(5)/Math.log(10)) - 1)/2+1);
numberOfClusters = (int)(numberOfDigits/(Math.log(CLUSTER_SIZE)/Math.log(10))+1);
long a1[] = new long[numberOfClusters + 1];
long b1[] = new long[numberOfClusters + 1];
long c1[] = new long[numberOfClusters + 1];
long a2[] = new long[numberOfClusters + 1];
long b2[] = new long[numberOfClusters + 1];
long c2[] = new long[numberOfClusters + 1];
a1[0] = 16;
a2[0] = 4;
lndiv(a1, 5);
lndiv(a2, 239);
11
System.arraycopy(a1, 0, c1, 0, numberOfClusters + 1);
System.arraycopy(a2, 0, c2, 0, numberOfClusters + 1);
for(j = 1; j < numberOfSteps; j++)
{
lndiv(a1, 25);
lndiv(a2, 239);
lndiv(a2, 239);
System.arraycopy(a1, 0, b1, 0, numberOfClusters + 1);
System.arraycopy(a2, 0, b2, 0, numberOfClusters + 1);
lndiv(b1, 2*j+1);
lndiv(b2, 2*j+1);
ifj%2 == 0) { lnadd(c1, b1); lnadd(c2, b2); }
( else { lnsub(c1, b1); lnsub(c2, b2); }
}
lndiv(a1, 25);
lndiv(a1, 2*numberOfSteps + 1);
System.out.println("Оценка точности результата:");
print(a1);
```

```
lnsub(c1, c2);
System.out.println("Результат:");
print(c1);
}
}
```

Данная программа — «небольшая», как по размерам (~150 строк), так и по другим признакам:

• Она решает одну четко поставленную задачу (выдает десятичные цифры числа π) в хорошо известных ограничениях (не более 30000 цифр), к тому же, не очень существенную для какой-либо практической или исследовательской деятельности.

• Неважно, насколько быстро она работает — на вычисление 30000 цифр уходит не более получаса даже на устаревших компьютерах, и этого вполне достаточно.

• Ущерб от неправильной работы программы практически нулевой (за исключением возможности обрушения ею системы, в которой выполняются и другие, более важные задачи).

• Не требуется дополнять программу новыми возможностями, практически никому не нужно разрабатывать ее новые версии или исправлять найденные ошибки.

• В связи со сказанным выше не очень нужно прилагать к программе подробную и понятную документацию — для человека, который ею заинтересуется, не составит большого труда понять, как ею пользоваться, просто по исходному коду.

Сложные или «большие» программы, называемые также программными системами, программными комплексами, программными продуктами, отличаются от «небольших» не столько по размерам (хотя обычно они значительно больше), сколько по наличию дополнительных факторов, связанных с их востребованностью и готовностью пользователей платить деньги как за приобретение самой программы, так и за ее сопровождение и даже за специальное обучение работе с ней.

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, возможно, также специальную документацию для администраторов, 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 (реинжиниринг).

• В ее разработку вовлечено значительное количество людей (более 5-ти человек). «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 Java, входящая в Java Development Kit [1].

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 — как уже говорилось, их нельзя написать «нахрапом». Изучением организационных, инженерных и технических аспектов создания ПО, включая методы разработки, занимается дисциплина, называемая ***программной инженерией***. Большая часть трудностей при разработке программных систем связана с организацией экономически эффективной совместной работы многих людей, приводящей к практически полезному результату. Это требует рассмотрения следующих аспектов.

• Над программой обычно работает много людей, иногда географически удаленных друг от друга и из различных организаций. Их работа должна быть организована так, чтобы затраты на разработку были бы покрыты доходами от продаж и предоставления услуг, связанных с полученной программой. В затраты входят зарплаты разработчиков, затраты на закупленное оборудование и программные инструменты разработки, на приобретение лицензий и патентование собственных решений, часто еще и затраты на исследование потребностей клиентов, проведение рекламы и другой маркетинговой деятельности.

• Значимые доходы могут быть получены, только если программа будет предоставлять пользователям в реальных условиях их работы такие возможности, что они готовы будут заплатить за это деньги (которым, заметим, без труда можно найти другие полезные применения). Для этого нужно учесть множество аспектов. Доходы от продаж значительно снизятся, если многие из пользователей не смогут воспользоваться программой только потому, что в их компьютерах процессоры слишком медленные или мало оперативной памяти, или потому что данные к системе часто поступают в искаженном виде и она не может их обработать, или потому что они привыкли работать с графическим интерфейсом, а система требует ввода из командной строки, и т.п.

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

Большинство опытных разработчиков и исследователей считают, что практически значимые программные системы всегда содержат ошибки. При переходе от «небольших» программ к «большим» понятие «правильной» программы становится практически бессмысленным. Про программную систему, в отличие от приведенной выше программы вычисления числа π, нельзя утверждать, что она «правильная», т.е. всегда правильно решает все поставленные перед ней задачи. Этот факт связан 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.

В связи с этим, вместо рассмотрения «правильных» и «неправильных» программных систем, в силу практического отсутствия первых, рассматривают «достаточно качественные» и «недостаточно качественные».

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. Более подробно понятие качественного программного обеспечения будет обсуждаться в одной из следующих лекций.

Часто программное обеспечение (ПО) нельзя рассматривать отдельно от программно-аппаратной системы, куда оно входит в качестве составной части. Изучением вопросов, связанных с разработкой и эксплуатацией программно-аппаратных систем занимается системная инженерия. В ее рамки попадает огромное количество проблем, связанных с аппаратной частью систем и обеспечением нужного уровня интеграции программной и аппаратной составляющих. Мы только изредка будем затрагивать вопросы, касающиеся системной инженерии в целом, в основном ограничиваясь аспектами, относящимися непосредственно к ПО.

В данном курсе будут рассматриваться различные подходы к решению проблем разработки, связанных с обеими составляющими дилеммы «ресурсы-качество» при создании сложных программ. Для изложения этих подходов вводится система понятий, относящихся к программным системам и процессам их создания и позволяющих эффективно разрабатывать такие системы, оценивать и планировать их свойства. В их число входят такие понятия, как жизненный цикл ПО, качество ПО, процесс разработки ПО, требования к ПО, архитектура ПО, образцы проектирования и пр.

Кроме того, особое внимание в курсе уделяется одному из подходов к разработке сложного ПО, компонентной разработке, предлагающей строить такие системы последовательно из отдельных элементов — компонентов, каждый из которых, в свою очередь, может рассматриваться как отдельная программная система. Курс дает введение в современные компонентные технологии разработки программных систем на основе платформ J2EE и .NET.

Проблемы, связанные с управлением ресурсами разработки, в том числе — планированием отдельных действий во времени, созданием эффективных команд разработчиков, относятся к управлению проектами, которому будет посвящена последняя лекция курса.

На основании опыта конструирования больших систем 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 их входными данными и какие документы, модели, другие части программной системы должны быть подготовлены в результате каждой отдельной работы. Элементы таких методик будут упоминаться на всем протяжении курса. Также будут рассматриваться отраслевые стандарты, содержащие описание выработанных на основе большого количества реальных проектов подходов к построению сложных программных систем.

При практической разработке больших систем, однако, стоит помнить, что все общеметодические 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. Оно также может приводить к имитации усердной работы при отсутствии реальных продвижений к нужным результатам.

Протест сообщества разработчиков против подобной бюрократизации разработки программ и попыток механического использования теоретических рекомендаций вылился в популярное сейчас движение *живой разработки ПО* (Agile Software Development). Одним из примеров «живого» процесса разработки является набор техник, известный как *экстремальное программирование* (Extreme Programming, XP). Некоторые аспекты этих подходов также будут рассмотрены в данном курсе.

2 Принципы работы со сложными программными системами.

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 *переиспользование*.

• **Абстракция (abstraction) и уточнение (refinement).** Абстракция является универсальным подходом к рассмотрению сложных вещей. Интеллект одного человека достаточно ограничен и просто 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 как-то все-таки работать с такими системами, мы пользуемся своей возможностью *абстрагироваться*, т.е. отвлекаться от всего, что несущественно для достижения поставленной в данной момент частной цели и не влияет на те аспекты рассматриваемого предмета, которые для этой цели важны.

Чтобы перейти от абстрактного представления к более конкретному, используется обратный процесс последовательного *уточнения.* Рассмотрев систему в каждом аспекте в отдельности, мы пытаемся 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.

Пример абстракции и уточнения. Систему хранения идентификаторов пользователей Интернет-магазина можно представить как множество целых чисел, забыв о том, что эти числа — идентификаторы пользователей, и о том, что все это как-то связано с Интернет-магазином. Затем описанную модель системы хранения идентификаторов пользователей Интернет-магазина можно уточнить, определив конкретную реализацию множества чисел, например, на основе сбалансированных красно-черных деревьев (см. [2], раздел 14, глава III и JDK классы `java.util.TreeSet` и `java.util.TreeMap`).

Другой 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, 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 (modularity).** *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.

Пример. Примером разбиения на модули может служить структура пакетов и классов библиотеки JDK. Классы, связанные с основными сущностями языка Java и виртуальной машины, собраны в пакете `java.lang`. Вспомогательные широко применяемые в различных приложениях классы, такие, как коллекции, представления даты и пр., собраны в `java.util`. Классы, используемые для реализации потокового ввода-вывода — в пакете `java.io`, и т.д. Интерфейсом класса служат его общедоступные методы, а интерфейсом пакета — его общедоступные классы.

Другой 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 по-

лучить 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. Например, класс `java.util.Queue<type E>`, реализующий функциональность очереди элементов типа `E`, имеет следующий интерфейс.

`E element()`	Возвращает элемент, стоящий в голове очереди, не изменяя ее. Создает исключение `NoSuchElementException`, если очередь пуста.
`boolean` `offer(E o)`	Вставляет, если возможно, данный элемент в конец очереди. Возвращает `true`, если вставка прошла успешно, `false` — иначе.
`E peek()`	Возвращает элемент, стоящий в голове очереди, не изменяя ее. Возвращает `null`, если очередь пуста.
`E poll()`	Возвращает элемент, стоящий в голове очереди, и удаляет его из очереди. Возвращает `null`, если очередь пуста.
`E remove()`	Возвращает элемент, стоящий в голове очереди, и удаляет его из очереди. Создает исключение `NoSuchElementException`, если очередь пуста.

Внутренние же данные и операции одного из классов, реализующих данный интерфейс, — `PriorityBlockingQueue<E>` — достаточно сложны. Этот класс реализует очередь с эффективной синхронизацией операций, позволяющей работать с таким объектом нескольким параллельным потокам без лишних ограничений на их синхронизацию. Например, один поток может добавлять элемент в конец непустой очереди, а другой в то же время извлекать ее первый элемент.

```
package java.util.concurrent;
import java.util.concurrent.locks.*;
import java.util.*;
public class PriorityBlockingQueue<E> extends AbstractQueue<E>
implements BlockingQueue<E>, java.io.Serializable {
private static final long serialVersionUID = 5595510919245408276L;
private final PriorityQueue<E> q;
private final ReentrantLock lock = new ReentrantLock(true);
private final ReentrantLock.ConditionObject notEmpty = lock.newCondition();
public PriorityBlockingQueue() { ... }
public PriorityBlockingQueue(int initialCapacity) { … }
```

```
public PriorityBlockingQueue(int initialCapacity,
Comparator<? super E> comparator) { … }
public PriorityBlockingQueue(Collection<? extends E> c) { ... }
public boolean add(E o) { ... }
public Comparator comparator() { … }
public boolean offer(E o) { … }
public void put(E o) { … }
public boolean offer(E o, long timeout, TimeUnit unit) { … }
public E take() throws InterruptedException { … }
public E poll() { … }
public E poll(long timeout, TimeUnit unit) throws InterruptedException { … }
public E peek() { … }
public int size() { … }
public int remainingCapacity() { … }
public boolean remove(Object o) { … }
public boolean contains(Object o) { … }
public Object[] toArray() { … }
public String toString() { … }
public int drainTo(Collection<? super E> c) { … }
public int drainTo(Collection<? super E> c, int maxElements) { … }
public void clear() { … }
public <T> T[] toArray(T[] a) { … }
public Iterator<E> iterator() { … }
private class Itr<E> implements Iterator<E> {
private final Iterator<E> iter;
Itr(Iterator<E> i) { … }
public boolean hasNext() { … }
public E next() { … }
public void remove() { … }
}
private void writeObject(java.io.ObjectOutputStream s)
throws java.io.IOException { … }
}
```

- **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. Более тонкий пример — методы `element()` и `peek()`. Нужда в них возникает, если программа не должна изменять очередь, и в то же время ей нужно узнать, какой элемент лежит в ее начале. Отсутствие такой возможности потребовало бы создавать собственное дополнительное хранилище элементов в каждой такой программе.

🕓 ***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). Представленный в примере интерфейс очереди не минимален — методы `element()` и `peek()`, а также `poll()` и `remove()` можно выразить друг через друга.

Минимальный интерфейс очереди получился бы, например, если выбросить пару методов `element()` и `remove()`. 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. Скажем, весь интерфейс очереди можно было бы свести к одной операции `Object queue(Object o, boolean remove)`, которая добавляет в очередь объект, указанный в качестве первого параметра, если это не **`null`**, а также возвращает объект в голове очереди (или **`null`**, если очередь пуста) и удаляет его, если в качестве второго параметра указать **`true`**. Однако, такой интерфейс явно сложнее для понимания, чем представленный выше.

o **Разделение ответственности.** Основной принцип выделения модулей — создание отдельных модулей под каждую задачу, решаемую системой или необходимую в качестве составляющей для решения ее основных задач. *Пример.* Класс `java.util.Date` представляет временную метку, состоящую из даты и времени. Это представление должно быть независимо от используемого календаря, формы записи дат и времени в данной стране и часового пояса. Для построения конкретных экземпляров этого класса на основе строкового представления даты и времени (например, «22:32:00, June 15, 2005») в том виде, как их используют в Европе, используется класс `java.util.GregorianCalendar`, поскольку интерпретация записи даты и времени зависит от используемой календарной системы. Разные календари представляются различными объектами интерфейса `java.util.Calendar`, которые отвечают за преобразование всех дат в некоторое независимое представление. Для создания строкового представления времени и даты используется класс `java.text.SimpleDateFormat`, поскольку нужное представление, помимо календарной системы, может иметь различный порядок перечисления года, месяца и дня месяца и различное количество символов, выде-

ляемое под представление разных элементов даты (например, «22:32:00, June 15, 2005» и «05.06.15, 22:32»). 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 класса `java.text.SimpleDateFormat`. 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. Пример такого разделения — отделение интерфейса абстрактного списка `java.util.List<E>` от многих возможных реализаций этого интерфейса, например, `java.util.ArrayList<E>`, `java.util.LinkedList<E>`. Первый из этих классов реализует список на основе массива, а второй — на основе ссылочной структуры данных.

- **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. Так, можно добавить в интерфейс очереди метод `void println(String)`, отправляющий строку на стандартный вывод. Но он совсем не связан с остальными и с задачами, решаемыми очередью. Следовательно, трудоемкость анализа и внесения изменений в полученную систему будет значительно выше — ведь изменения в контексте разных задач возникают обычно независимо. Поэтому гораздо лучше поместить такой метод в другой модуль.

• **Переиспользование.** 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 эле-

мента знаний и организуется его переиспользование 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. Примером может служить организация библиотечных классов `java.util.TreeSet` и `java.util.TreeMap`. Первый класс реализует хранение множества элементов, на которых определен порядок, в виде сбалансированного дерева. Второй класс реализует то же самое для ассоциативного массива или словаря (map), если определен порядок его ключей. Все алгоритмы работы со сбалансированным деревом в обоих случаях одинаковы, поэтому имеет смысл реализовать их только один раз. Если посмотреть на код этих классов в библиотеке JDK от компании Sun, можно увидеть, что ее разработчики так и поступили — класс `TreeSet` реализован как соответствующий ассоциативный массив `TreeMap`, в котором ключи представляют собой множество хранимых значений, а значение в любой паре (ключ, значение) равно **`null`**.

3 Модульное программирование

http://vit-prog.narod.ru/page/TRPP/section_1/subject_1.3.htm

Модульное программирование основано на понятии **модуля** - логически взаимосвязанной совокупности функциональных элементов, оформленных в виде отдельных программных модулей.

Модуль характеризуют:

1. один вход и один выход - на входе программный модуль получает определенный набор исходных данных, выполняет содержательную обработку и возвращает один набор результатных данных, т.е. реализуется стандартный принцип IPO (Input - Process - Output) - *вход-процесс-выход;*
2. функциональная завершенность - модуль выполняет перечень регламентированных операций для реализации каждой отдельной функции в полном составе, достаточных для завершения начатой обработки;
3. логическая независимость - результат работы программного модуля зависит только от исходных данных, но не зависит от работы других модулей;
4. слабые информационные связи с другими программными модулями - обмен информацией между модулями должен быть по возможности минимизирован;
5. обозримый по размеру и сложности программный элемент.

Таким образом, модули содержат определение доступных для обработки данных, операции обработки данных, схемы взаимосвязи с другими модулями.

Каждый модуль состоит из спецификации и тела. Спецификации определяют правила использования модуля, а тело – способ реализации процесса обработки.

Приступая к разработке каждой программы ПС, следует иметь ввиду, что она, как правило, является большой системой, поэтому мы должны принять меры для ее упрощения. Для этого такую программу разрабатывают по частям, которые называются программными модулями. А сам такой метод разработки программ называют модульным программированием. *Программный модуль* - это любой фрагмент описания процесса, оформляемый как самостоятельный программный продукт, пригодный для использования в описаниях процесса. Это означает, что каждый программный модуль программируется, компилируется и отлаживается отдельно от других модулей программы, и тем самым, физически разделен с другими модулями программы. Более того, каждый разработанный программный модуль может включаться в состав разных программ, если выполнены условия его использования, декларированные в документации по этому модулю. Таким образом, программный модуль может рассматриваться и как средство борьбы со сложностью программ, и как средство борьбы с дублированием в программировании (т.е. как средство накопления и многократного использования программистских знаний).

Модульное программирование является воплощением в процессе разработки программ обоих общих методов борьбы со сложностью и обеспечение независимости компонент системы, и использование иерархических структур. Для воплощения первого метода формулируются определенные требования, которым должен удовлетворять программный модуль, т.е. выявляются основные характеристики "хорошего" программного модуля. Для воплощения второго метода ис-

пользуют древовидные модульные структуры программ (включая деревья со сросшимися ветвями).

1.3.2. Основные характеристики программного модуля.

Не всякий программный модуль способствует упрощению программы [7.2]. Выделить хороший с этой точки зрения модуль является серьезной творческой задачей. Для оценки приемлемости выделенного модуля используются некоторые критерии. Так, Хольт [7.4] предложил следующие два общих таких критерия:

- хороший модуль снаружи проще, чем внутри;
- хороший модуль проще использовать, чем построить.

Майерс предлагает использовать более конструктивные характеристики программного модуля для оценки его приемлемости: размер модуля; прочность модуля; сцепление с другими модулями; рутинность модуля (независимость от предыстории обращений к нему).

Размер модуля измеряется числом содержащихся в нем операторов (строк). Модуль не должен быть слишком маленьким или слишком большим. Маленькие модули приводят к громоздкой модульной структуре программы и могут не окупать накладных расходов, связанных с их оформлением. Большие модули неудобны для изучения и изменений, они могут существенно увеличить суммарное время повторных трансляций программы при отладке программы. Обычно рекомендуются программные модули размером от нескольких десятков до нескольких сотен операторов.

Прочность модуля - это мера его внутренних связей. Чем выше прочность модуля, тем больше связей он может спрятать от внешнейпо отношению к нему части программы и, следовательно, тем больший вклад в упрощение программы он может внести. Для оценки степени прочности модуля Майерс предлагает упорядоченный по степени прочности набор из семи классов модулей. Самой слабой степенью прочности обладает модуль, *прочный по совпадению*. Это такой модуль, между элементами которого нет осмысленных связей. Такой модуль может быть выделен, например, при обнаружении в разных местах программы повторения одной и той же последовательности операторов, которая и оформляется в отдельный модуль. Необходимость изменения этой последовательности в одном из контекстов может привести к изменению этого модуля, что может сделать его использование в других контекстах ошибочным. Такой класс программных модулей не рекомендуется для использования. Вообще говоря, предложенная Майерсом упорядоченность по степени прочности классов модулей не бесспорна. Однако, это не очень существенно, так как только два высших по прочности класса модулей рекомендуются для использования. Эти классы мы и рассмотрим подробнее.

Функционально прочный модуль - это модуль, выполняющий (реализующий) одну какую-либо определенную функцию. При реализации этой функции такой модуль может использовать и другие модули. Такой класс программных модулей рекомендуется для использования.

Информационно прочный модуль - это модуль, выполняющий (реализующий) несколько операций (функций) над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Для каждой из этих операций в таком модуле имеется свой вход со своей формой об-

ращения к нему. Такой класс следует рассматривать как класс программных модулей с высшей степенью прочности. Информационно-прочный модуль может реализовывать, например, абстрактный тип данных.

В модульных языках программирования как минимум имеются средства для задания функционально прочных модулей (например, модуль типа FUNCTION в языке ФОРТРАН). Средства же для задания информационно прочных модулей в ранних языках программирования отсутствовали - они появились только в более поздних языках. Так в языке программирования Ада средством задания информационно прочного модуля является пакет.

Сцепление модуля - это мера его зависимости по данным от других модулей. Характеризуется способом передачи данных. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость от других модулей. Для оценки степени сцепления Майерс предлагает упорядоченный набор из шести видов сцепления модулей. Худшим видом сцепления модулей является *сцепление по содержимому*. Таким является сцепление двух модулей, когда один из них имеет прямые ссылки на содержимое другого модуля (например, на константу, содержащуюся в другом модуле). Такое сцепление модулей недопустимо. Не рекомендуется использовать также *сцепление по общей области* - это такое сцепление модулей, когда несколько модулей используют одну и ту же область памяти. Такой вид сцепления модулей реализуется, например, при программировании на языке ФОРТРАН с использованием блоков COMMON. Единственным видом сцепления модулей, который рекомендуется для использования современной технологией программирования, является параметрическое сцепление (сцепление по данным по Майерсу) - это случай, когда данные передаются модулю либо при обращении к нему как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Такой вид сцепления модулей реализуется на языках программирования при использовании обращений к процедурам (функциям).

Рутинность модуля - это его независимость от предыстории обращений к нему. Модуль будем называть *рутинным*, если результат (эффект) обращения к нему зависит только от значений его параметров (и не зависит от предыстории обращений к нему). Модуль будем называть *зависящим от предыстории*, если результат (эффект) обращения к нему зависит от внутреннего состояния этого модуля, хранящего следы предыдущих обращений к нему. Майерс не рекомендует использовать зависящие от предыстории (непредсказуемые) модули, так как они провоцируют появление в программах хитрых (неуловимых) ошибок. Однако такая рекомендация является неконструктивной, так как во многих случаях именно зависящий от предыстории модуль является лучшей реализаций информационно прочного модуля. Поэтому более приемлема следующая (более осторожная) рекомендация:

- всегда следует использовать рутинный модуль, если это не приводит к плохим (не рекомендуемым) сцеплениям модулей;
- зависящие от предыстории модули следует использовать только в случае, когда это необходимо для обеспечения параметрического сцепления;

- в спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость таким образом, чтобы было возможно прогнозировать поведение (эффект выполнения) данного модуля при разных последующих обращениях к нему.

В связи с последней рекомендацией может быть полезным определение внешнего представления (ориентированного на информирование человека) состояний зависящего от предыстории модуля. В этом случае эффект выполнения каждой функции (операции), реализуемой этим модулем, следует описывать в терминах этого внешнего представления, что существенно упростит прогнозирование поведения данного модуля.

1.3.3. Модульная структура программных продуктов

Принципы модульного программирования программных продуктов во многом сходны с принципами нисходящего проектирования. Сначала определяются состав и подчиненность функций, а затем - набор программных модулей, реализующих эти функции.

Однотипные функции реализуются одними и теми же модулями. Функция верхнего уровня обеспечивается **главным** модулем; он управляет выполнением нижестоящих функций, которым соответствуют **подчиненные** модули.

При определении набора модулей, реализующих функции конкретного алгоритма, необходимо учитывать следующее:

1. каждый модуль вызывается на выполнение вышестоящим модулем и, закончив работу, возвращает управление вызвавшему его модулю;
2. принятие основных решений в алгоритме выносится на максимально "высокий" по иерархии уровень;
3. для использования одной и той же функции в разных местах алгоритма создается один модуль, который вызывается на выполнение по мере необходимости. В результате дальнейшей детализации алгоритма создается *функционально-модульная* схема (ФМС) алгоритма приложения, которая является основой для программирования.

Состав и вид программных модулей, их назначение и характер использования в программе в значительной степени определяются инструментальными средствами. Например, применительно к средствам СУБД отдельными модулями могут быть:

1. экранные формы ввода и/или редактирования информации базы данных;
2. отчеты генератора отчетов;
3. макросы;
4. стандартные процедуры обработки информации;
5. меню, обеспечивающее выбор функции обработки и др.

1.3.4. Методы разработки структуры программы.

Как уже отмечалось выше, в качестве модульной структуры программы принято использовать древовидную структуру, включая деревья со сросшимися ветвями. В узлах такого дерева размещаются программные модули, а направленные дуги (стрелки) показывают статическую подчиненность модулей, т.е. каждая дуга показывает, что в тексте модуля, из которого она исходит, имеется ссылка на модуль, в который она входит. Другими словами, каждый модуль может обра-

щаться к подчиненным ему модулям, т.е. выражается через эти модули. При этом модульная структура программы, в конечном счете, должна включать и совокупность спецификаций модулей, образующих эту программу. *Спецификация* программного модуля содержит, во-первых, синтаксическую спецификацию его входов, позволяющую построить на используемом языке программирования синтаксически правильное обращение к нему (к любому его входу), и, во-вторых, функциональную спецификацию модуля (описание семантики функций, выполняемых этим модулем по каждому из его входов). Функциональная спецификация модуля строится так же, как и функциональная спецификация ПС.

В процессе разработки программы ее модульная структура может поразному формироваться и использоваться для определения порядка программирования и отладки модулей, указанных в этой структуре. Поэтому можно говорить о разных методах разработки структуры программы. Обычно в литературе обсуждаются два метода [7.1, 7.7]: метод восходящей разработки и метод нисходящей разработки.

Метод *восходящей разработки* заключается в следующем. Сначала строится модульная структура программы в виде дерева. Затемпоочередно программируются модули программы, начиная с модулей самого нижнего уровня (листья дерева модульной структуры программы), в таком порядке, чтобы для каждого программируемого модуля были уже запрограммированы все модули, к которым он может обращаться. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в принципе в таком же (восходящем) порядке, в каком велось их программирование. На первый взгляд такой порядок разработки программы кажется вполне естественным: каждый модуль при программировании выражается через уже запрограммированные непосредственно подчиненные модули, а при тестировании использует уже отлаженные модули. Однако, современная технология не рекомендует такой порядок разработки программы. Во-первых, для программирования какого-либо модуля совсем не требуется текстов используемых им модулей - для этого достаточно, чтобы каждый используемый модуль был лишь специфицирован (в объеме, позволяющем построить правильное обращение к нему), а для тестирования его возможно (и даже, как мы покажем ниже, полезно) используемые модули заменять их имитаторами (заглушками). Во-вторых, каждая программа в какой-то степени подчиняется некоторым внутренним для нее, но глобальным для ее модулей соображениям (принципам реализации, предположениям, структурам данных и т.п.), что определяет ее концептуальную целостность и формируется в процессе ее разработки. При восходящей разработке эта глобальная информация для модулей нижних уровней еще не ясна в полном объеме, поэтому очень часто приходится их перепрограммировать, когда при программировании других модулей производится существенное уточнение этой глобальной информации (например, изменяется глобальная структура данных). В-третьих, при восходящем тестировании для каждого модуля (кроме головного) приходится создавать ведущую программу (модуль), которая должна подготовить для тестируемого модуля необходимое состояние информационной среды и произвести требуемое обращение к нему. Это приводит к боль-

шому объему "отладочного" программирования и в то же время не дает никакой гарантии, что тестирование модулей производилось именно в тех условиях, в которых они будут выполняться в рабочей программе.

Метод *нисходящей разработки* заключается в следующем. Как и в предыдущем методе сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модуля самого верхнего уровня (головного), переходя к программированию какого-либо другого модуля только в том случае, если уже запрограммирован модуль, который к нему обращается. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же (нисходящем) порядке. При таком порядке разработки программы вся необходимая глобальная информация формируется своевременно, т.е. ликвидируется весьма неприятный источник просчетов при программировании модулей. Существенно облегчается и тестирование модулей, производимое при нисходящем тестировании программы. Первым тестируется головной модуль программы, который представляет всю тестируемую программу и поэтому тестируется при "естественном" состоянии информационной среды, при котором начинает выполняться эта программа. При этом все модули, к которым может обращаться головной, заменяются на их имитаторы (так называемые заглушки [7.5]). Каждый *имитатор модуля* представляется весьма простым программным фрагментом, сигнализирующим, в основном, о самом факте обращения к имитируемому модулю с необходимой для правильной работы программы обработкой значений его входных параметров (иногда с их распечаткой) и с выдачей, если это необходимо, заранее запасенного подходящего результата. После завершения тестирования и отладки головного и любого последующего модуля производится переход к тестированию одного из модулей, которые в данный момент представлены имитаторами, если таковые имеются. Для этого имитатор выбранного для тестирования модуля заменяется на сам этот модуль и добавляются имитаторы тех модулей, к которым может обращаться выбранный для тестирования модуль. При этом каждый такой модуль будет тестироваться при "естественных" состояниях информационной среды, возникающих к моменту обращения к этому модулю при выполнении тестируемой программы. Таким образом большой объем "отладочного" программирования заменяется программированием достаточно простых имитаторов используемых в программе модулей. Кроме того, имитаторы удобно использовать для подыгрывания процессу подбора тестов путем задания нужных результатов, выдаваемых имитаторами.

Некоторым недостатком нисходящей разработки, приводящим к определенным затруднениям при ее применении, является необходимость абстрагироваться от базовых возможностей используемого языка программирования, выдумывая абстрактные операции, которые позже нужно будет реализовать с помощью выделенных в программе модулей. Однако способность к таким абстракциям представляется необходимым условием разработки больших программных средств, поэтому ее нужно развивать.

В рассмотренных методах восходящей и нисходящей разработок (которые мы будем называть *классическими*) модульная древовидная структуру программы должна разрабатываться до начала программирования модулей. Однако такой

подход вызывает ряд возражений: представляется сомнительным, чтобы до программирования модулей можно было разработать структуру программы достаточно точно и содержательно. На самом деле это делать не обязательно: так при конструктивном и архитектурном подходах к разработке программ модульная структура формируется в процессе программирования модулей.

Конструктивный подход к разработке программы представляет собой модификацию нисходящей разработки, при которой модульная древовидная структура программы формируется в процессе программирования модуля. Сначала программируется головной модуль, исходя из спецификации программы в целом, причем спецификация программы является одновременно и спецификацией ее головного модуля, так как последний полностью берет на себя ответственность за выполнение функций программы. В процессе программирования головного модуля, в случае, если эта программа достаточно большая, выделяются подзадачи (внутренние функции), в терминах которых программируется головной модуль. Это означает, что для каждой выделяемой подзадачи (функции) создается спецификация реализующего ее фрагмента программы, который в дальнейшем может быть представлен некоторым поддеревом модулей. Важно заметить, что здесь также ответственность за выполнение выделенной функции берет головной (может быть, и единственный) модуль этого поддерева, так что спецификация выделенной функции является одновременно и спецификацией головного модуля этого поддерева. В головном модуле программы для обращения к выделенной функции строится обращение к головному модулю указанного поддерева в соответствии с созданной его спецификацией. Таким образом, на первом шаге разработки программы (при программировании ее головного модуля) формируется верхняя начальная часть дерева, например, такая, которая показана на рис. 1.3.1.

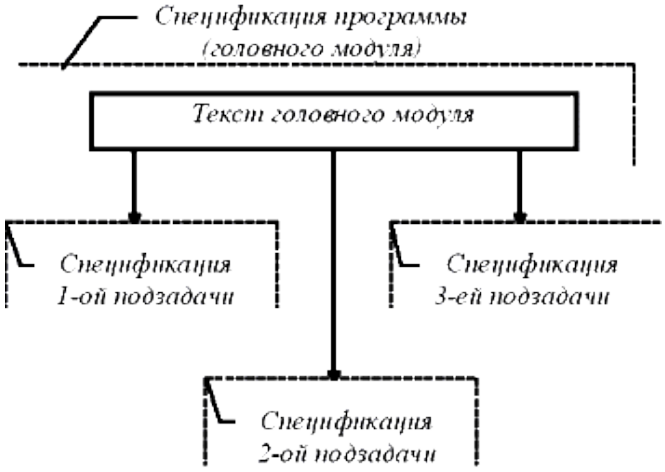


Рис. 1.3.1. Первый шаг формирования модульной структуры программы при конструктивном подходе.

Аналогичные действия производятся при программировании любого другого модуля, который выбирается из текущего состояния дерева программы из числа специфицированных, но пока еще не запрограммированных модулей. В результате

этого производится очередное доформирование дерева программы, например, такое, которое показано на рис. 1.3.2.

Архитектурный подход к разработке программы представляет собой модификацию восходящей разработки, при которой модульная структура программы формируется в процессе программирования модуля. Но при этом ставится существенно другая цель разработки: повышение уровня используемого языка программирования, а не разработка конкретной программы. Это означает, что для заданной предметной области выделяются типичные функции, каждая из которых может использоваться при решении разных задач в этой области, и специфицируются, а затем и программируются отдельные программные модули, выполняющие эти функции. Так как процесс выделения таких функций связан с накоплением и обобщением опыта решения задач в заданной предметной области, то обычно сначала выделяются и реализуются отдельными модулями более простые функции, а затем постепенно появляются модули, использующие ранее выделенные функции. Такой набор модулей создается в расчете на то, что при разработке той или иной программы заданной предметной области в рамках конструктивного подхода могут оказаться приемлемыми некоторые из этих модулей. Это позволяет существенно сократить трудозатраты на разработку конкретной программы путем подключения к ней заранее заготовленных и проверенных на практике модульных структур нижнего уровня. Так как такие структуры могут многократно использоваться в разных конкретных программах, то архитектурный подход может рассматриваться как путь борьбы с дублированием в программировании. В связи с этим программные модули, создаваемые в рамках архитектурного подхода, обычно параметризуются для того, чтобы усилить применимость таких модулей путем настройки их на параметры.

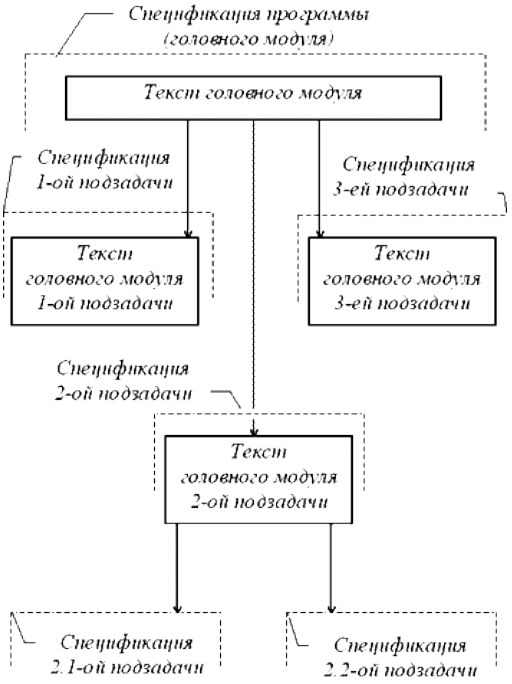


Рис. 1.3.2. Второй шаг формирования модульной структуры программы при конструктивном подходе.

В классическом методе нисходящей разработки рекомендуется сначала все модули разрабатываемой программы запрограммировать, а уж затем начинать нисходящее их тестирование. Однако такой порядок разработки не представляется достаточно обоснованным: тестирование и отладка модулей может привести к изменению спецификации подчиненных модулей и даже к изменению самой модульной структуры программы, так что в этом случае программирование некоторых модулей может оказаться бесполезно проделанной работой. Нам представляется более рациональным другой порядок разработки программы, известный в литературе как метод нисходящей реализации. В этом методе каждый запрограммированный модуль начинают сразу же тестировать до перехода к программированию другого модуля.

Все эти методы имеют еще различные разновидности в зависимости от того, в какой последовательности обходятся узлы (модули) древовидной структуры программы в процессе ее разработки. Это можно делать, например, по слоям (разрабатывая все модули одного уровня, прежде чем переходить к следующему уровню). При нисходящей разработке дерево можно обходить также в лексикографическом порядке (сверху-вниз, слева-направо). Возможны и другие варианты обхода дерева. Так, при конструктивной реализации для обхода дерева программы целесообразно следовать идеям Фуксмана, которые он использовал в предложенном им методе вертикального слоения. Сущность такого обхода заключается в следующем. В рамках конструктивного подхода сначала реализуются только те модули, которые необходимы для самого простейшего варианта программы, которая может нормально выполняться только для весьма ограниченного множества наборов входных данных, но для таких данных эта задача будет решаться до конца. Вместо других модулей, на которые в такой программе имеются ссылки, в эту программу вставляются лишь их имитаторы, обеспечивающие, в основном, контроль за выходом за пределы этого частного случая. Затем к этой программе добавляются реализации некоторых других модулей (в частности, вместо некоторых из имеющихся имитаторов), обеспечивающих нормальное выполнение для некоторых других наборов входных данных. И этот процесс продолжается поэтапно до полной реализации требуемой программы. Таким образом, обход дерева программы производится с целью кратчайшим путем реализовать тот или иной вариант (сначала самый простейший) нормально действующей программы. В связи с этим такая разновидность конструктивной реализации получила название метода *целенаправленной конструктивной реализации*. Достоинством этого метода является то, что уже на достаточно ранней стадии создается работающий вариант разрабатываемой программы. Психологически это играет роль допинга, резко повышающего эффективность разработчика. Поэтому этот метод является весьма привлекательным.

Рис. 1.3.3. Классификация методов разработки структуры программ.

Подводя итог сказанному, на рис. 1.3.3. представлена общая схема классификации рассмотренных методов разработки структуры программы.

1.3.5. Контроль структуры программы.

Для контроля структуры программы можно использовать три метода:

- статический контроль,
- смежный контроль,
- сквозной контроль.

Статический контроль состоит в оценке структуры программы сточки зрения хорошо ли программа разбита на модули с учетом значений рассмотренных выше основных характеристик модуля.

Смежный контроль сверху - это контроль со стороны разработчиков архитектуры и внешнего описания ПС. Смежный контроль снизу - это контроль спецификации модулей со стороны разработчиков этих модулей.

Сквозной контроль - это мысленное прокручивание (проверка) структуры программы при выполнении заранее разработанных тестов. Является видом динамического контроля так же, как и ручная имитация функциональной спецификации или архитектуры ПС.

Следует заметить, что характер осуществления этих методов контроля зависит от выбранного метода разработки структуры программы: при классическом подходе они применяются до начала программирования модулей, а при конструктивном и архитектурном подходах - в процессе программирования модулей (в подходящие моменты времени).

4 Функциональное программирование

Функциональное программирование — раздел дискретной математики и парадигма программирования, в которой процесс вычисленийтрактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании).

Противопоставляется парадигме императивного программирования, которая описывает процесс вычислений как последовательное измененийсостояний (в значении, подобном таковому в теории автоматов). При необходимости, в функциональном программировании вся совокупность последовательных состояний вычислительного процесса представляется явным образом, например как список.

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния (в отличие от императивного, где одной из базовых концепций является переменная, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма).

На практике отличие математической функции от понятия «функции» в императивном программировании заключается в том, что императивные функции могут опираться не только на аргументы, но и на состояние внешних по отношению к функции переменных, а также иметь побочные эффекты и менять состояние внешних переменных. Таким образом, в императивном программировании при вызове одной и той же функции с одинаковыми параметрами, но на разных этапах выполнения алгоритма, можно получить разные данные на выходе из-за влияния на функцию состояния переменных. А в функциональном языке при вызове функции с одними и теми же аргументами мы всегда получим одинаковый результат: выходные данные зависят только от входных. Это позволяет средам выполнения программ на функциональных языках кешировать результаты функций и вызывать их в порядке, не определяемом алгоритмом и распараллеливать их без каких-либо дополнительных действий со стороны программиста (см.ниже Чистые функции)

λ-исчисления являются основой для функционального программирования, многие функциональные языки можно рассматривать как «надстройку» над ними[1].

Языки функционального программирования

Основная статья: ***Язык функционального программирования***

Наиболее известными языками функционального программирования являются[2]:

- LISP — (Джон МакКарти, 1958) и множество его диалектов, наиболее современные из которых:
  - Scheme
  - Clojure
  - Common Lisp
- Erlang — (Joe Armstrong, 1986) функциональный язык с поддержкой процессов.

- APL — предшественник современных научных вычислительных сред, таких как MATLAB.
- ML (Робин Милнер, 1979, из ныне используемых диалектов известны Standard ML и Objective CAML).
- F# — функциональный язык семейства ML для платформы .NET
- Scala
- Miranda (Дэвид Тёрнер, 1985, который впоследствии дал развитие языку Haskell).
- Nemerle — гибридный функционально/императивный язык.
- XSLT и XQuery
- Haskell — чистый функциональный. Назван в честь Хаскелла Карри.

Ещё не полностью функциональные изначальные версии и Lisp и APL внесли особый вклад в создание и развитие функционального программирования. Более поздние версии Lisp, такие как Scheme, а также различные варианты APL поддерживали все свойства и концепции функционального языка[3].

Как правило, интерес к функциональным языкам программирования, особенно чисто функциональным, был скорее научный, нежели коммерческий. Однако, такие примечательные языки как Erlang, OCaml, Haskell, Scheme (после 1986) а также специфические R (статистика), Mathematica (символьная математика), J и K (финансовый анализ), и XSLT (XML) находили применение в индустрии коммерческого программирования. Такие широко распространенные декларативные языки как SQL и Lex/Yacc содержат некоторые элементы функционального программирования, например, они остерегаются использовать переменные. Языки работы с электронными таблицами также можно рассматривать как функциональные, потому что в ячейках электронных таблиц задаётся массив функций, как правило зависящих лишь от других ячеек, а при желании смоделировать переменные приходится прибегать к возможностям императивного языка макросов.

История

Лямбда-исчисление стало теоретической базой для описания и вычисления функций. Являясь математической абстракцией, а не языком программирования, оно составило базис почти всех языков функционального программирования на сегодняшний день. Сходное теоретическое понятие, комбинаторная логика, является более абстрактным, нежели λ-исчисления и было создано раньше. Эта логика используется в некоторых эзотерических языках, например в Unlambda. И λ-исчисление, и комбинаторная логика были разработаны для более ясного и точного описания принципов и основ математики[4].

Первым функциональным языком был Lisp, созданный Джоном МакКарти в период его работы в Массачусетском технологическом институте в конце пятидесятых и реализованный, первоначально, для IBM 700/7000 (англ.)русск.[5]. Lisp ввел множество понятий функционального языка, хотя при этом исповедовал не только парадигму функционального программирования[6]. Дальнейшим развитием лиспа стали такие языки как Scheme и Dylan.

Язык обработки информации (Information Processing Language (англ.)русск., IPL) иногда определяется как самый первый машинный функциональный язык[7]. Это язык ассемблерного типа для работы со списком символов. В нём было поня-

тие «генератора», который использовал функцию в качестве аргумента, а также, поскольку это язык ассемблерного уровня, он может позиционироваться как язык, имеющий функции высшего порядка. Однако, в целом IPL акцентирован на использование императивных понятий[8].

Кеннет Е. Айверсон разработал язык APL в начале шестидесятых, документировав его в своей книге A Programming Language (ISBN 9780471430148)[9]. APL оказал значительное влияние на язык FP (англ.)русск., созданный Джоном Бэкусом. В начале девяностых Айверсон и Роджер Хуэй (англ.)русск. создали преемника APL — язык программирования J. В середине девяностых Артур Витни (англ.)русск., ранее работавший с Айверсоном, создал язык K, который впоследствии использовался в финансовой индустрии на коммерческой основе.

В семидесятых в университете Эдинбурга Робин Милнер создал язык ML, а Дэвид Тернер начинал разработку языка SASL в университете Сент-Эндрюса и, впоследствии, язык Mirandaв университете города Кент. В конечном итоге на основе ML были созданы несколько языков, среди которых наиболее известные Objective Caml и Standard ML. Также в семидесятых осуществлялась разработка языка программирования, построенного по принципу Scheme (реализация не только функциональной парадигмы), получившего описание в известной работе «Lambda Papers», а также в книге восемьдесят пятого года «Structure and Interpretation of Computer Programs», в которой принципы функционального программирования были донесены до более широкой аудитории.[*источник не указан 1405 дней*]

В восьмидесятых Пер Мартин-Лёф создал интуиционистскую теорию типов (также называемую конструктивной). В этой теории функциональное программирование получило конструктивное доказательство того, что ранее было известно как зависимый тип. Это дало мощный толчок к развитию диалогового доказательства теорем и к последующему созданию множества функциональных языков. Haskell был создан в конце восьмидесятых в попытке соединить множество идей, полученных в ходе исследования функционального программирования.[3]

Концепции

Некоторые концепции и парадигмы специфичны для функционального программирования и в основном чужды императивному программированию (включая объектно-ориентированное программирование). Тем не менее, языки программирования обычно представляют собой гибрид нескольких парадигм программирования, поэтому «большей частью императивные» языки программирования могут использовать какие-либо из этих концепций.[*источник не указан 1405 дней*]

Функции высших порядков

Основная статья: ***Функция высшего порядка***

Функции высших порядков — это такие функции, которые могут принимать в качестве аргументов и возвращать другие функции.[10] Математики такую функцию чаще называютоператором, например, оператор взятия производной или интегральный оператор.

Функции высших порядков позволяют использовать карринг — преобразование функции от пары аргументов в функцию, берущую свои аргументы по одному. Это преобразование получило свое название в честь Х. Карри.

Чистые функции

Чистыми называют функции, которые не имеют побочных эффектов ввода-вывода и памяти (они зависят только от своих параметров и возвращают только свой результат). Чистые функции обладают несколькими полезными свойствами, многие из которых можно использовать для оптимизации кода:

- Если результат чистой функции не используется, он может быть удален без вреда для других выражений.
- Результат вызова чистой функции может быть мемоизирован, то есть сохранен в таблице значений вместе с аргументами вызова. Если в дальнейшем функция вызывается с этими же аргументами, ее результат может быть взят прямо из таблицы, не вычисляясь (иногда это называется принципом прозрачности ссылок). Мемоизация, ценой небольшого расхода памяти, позволяет существенно увеличить производительность и уменьшить порядок роста некоторых рекурсивных алгоритмов.
- Если нет никакой зависимости по данным между двумя чистыми функциями, то порядок их вычисления можно поменять или распараллелить (говоря иначе вычисление чистых функций удовлетворяет принципам thread-safe)
- Если весь язык не допускает побочных эффектов, то можно использовать любую политику вычисления. Это предоставляет свободу компилятору комбинировать и реорганизовывать вычисление выражений в программе (например, исключить древовидные структуры).

Хотя большинство компиляторов императивных языков программирования распознают чистые функции и удаляют общие подвыражения для вызовов чистых функций, они не могут делать это всегда для предварительно скомпилированных библиотек, которые, как правило, не предоставляют эту информацию. Некоторые компиляторы, такие как gcc, в целях оптимизации предоставляют программисту ключевые слова для обозначения чистых функций[11]. Fortran 95 позволяет обозначать функции как «pure» (чистые)[12].

Рекурсия

Основная статья: ***Рекурсия***

В функциональных языках цикл обычно реализуется в виде рекурсии. Строго говоря, в функциональной парадигме программирования нет такого понятия, как цикл. Рекурсивные функции вызывают сами себя, позволяя операции выполняться снова и снова. Для использования рекурсии может потребоваться большой стек, но этого можно избежать в случаехвостовой рекурсии. Хвостовая рекурсия может быть распознана и оптимизирована компилятором в код, получаемый после компиляции аналогичной итерации в императивном языке программирования.[13] Стандарты языка Scheme требуют распознавать и оптимизировать хвостовую рекурсию. Оптимизировать хвостовую рекурсию можно путём преобразования программы в стиле использования продолжений при её компиляции, как один из способов.[14]

Рекурсивные функции можно обобщить с помощью функций высших порядков, используя, например, катаморфизм и анаморфизм (или «свертка» и «раз-

вертка»). Функции такого рода играют роль такого понятия как цикл в императивных языках программирования.[*источник не указан 1405 дней*]

Подход к вычислению аргументов

Функциональные языки можно классифицировать по тому, как обрабатываются аргументы функции в процессе её вычисления. Технически различие заключается в денотационной семантике выражения. К примеру, при строгом подходе к вычислению выражения

```
print(len([2+1, 3*2, 1/0, 5-4]))
```

на выходе будет ошибка, так как в третьем элементе списка присутствует деление на ноль. При нестрогом подходе значением выражения будет 4, поскольку для вычисления длины списка значения его элементов, строго говоря, не важны и могут вообще не вычисляться. При строгом (аппликативном) порядке вычисления заранее подсчитываются значения всех аргументов перед вычислением самой функции. При нестрогом подходе (нормальный порядок вычисления) значения аргументов не вычисляются до тех пор, пока их значение не понадобится при вычислении функции.

Как правило, нестрогий подход реализуется в виде редукции графа. Нестрогое вычисление используется по умолчанию в нескольких чисто функциональных языках, в том числеMiranda, Clean и Haskell.

ФП в нефункциональных языках

Принципиально нет препятствий для написания программ в функциональном стиле на языках, которые традиционно не считаются функциональными, точно так же, как программы вобъектно-ориентированном стиле можно писать на структурных языках. Некоторые императивные языки поддерживают типичные для функциональных языков конструкции, такие как функции высшего порядка и списковые включения (list comprehensions), что облегчает использование функционального стиля в этих языках. Примером может быть функциональное программирование на языке Python.

В языке C указатели на функцию в качестве типов аргументов могут быть использованы для создания функций высшего порядка. Функции высшего порядка и отложенная списковая структура реализованы в библиотеках C++. В языке C# версии 3.0 и выше можно использовать λ-функции для написания программы в функциональном стиле. В сложных языках, типаАлгол-68, имеющиеся средства метапрограммирования (фактически, дополнения языка новыми конструкциями) позволяют создать специфичные для функционального стиля объекты данных и программные конструкции, после чего можно писать функциональные программы с их использованием.[*источник не указан 1405 дней*]

Стили программирования

Императивные программы имеют склонность акцентировать последовательности шагов для выполнения какого-то действия, а функциональные программы к расположению и композиции функций, часто не обозначая точной последовательности шагов. Простой пример двух решений одной задачи (используется один и тот же язык Python) иллюстрирует это.

```
# императивный стиль
target = []  # создать пустой список
```

```
for item in source_list:  # для каждого элемента исходного списка
    trans1 = G(item)  # применить функцию G()
    trans2 = F(trans1)  # применить функцию F()
    target.append(trans2)  # добавить преобразованный элемент в список
```

Функциональная версия выглядит по-другому:

```
# функциональный стиль
# языки ФП часто имеют встроенную функцию compose()
compose2 = lambda A, B: lambda x: A(B(x))
target = map(compose2(F, G), source_list)
```

В отличие от императивного стиля, описывающего шаги, ведущие к достижению цели, функциональный стиль описывает математические отношения между данными и целью.

Особенности

Основной особенностью функционального программирования, определяющей как преимущества, так и недостатки данной парадигмы, является то, что в ней реализуется *модель вычислений без состояний*. Если императивная программа на любом этапе исполнения имеет состояние, то есть совокупность значений всех переменных, и производит побочные эффекты, то чисто функциональная программа ни целиком, ни частями состояния не имеет и побочных эффектов не производит. То, что в императивных языках делается путём присваивания значений переменным, в функциональных достигается путём передачи выражений в параметры функций. Непосредственным следствием становится то, что чисто функциональная программа не может изменять уже имеющиеся у неё данные, а может лишь порождать новые путём копирования и/или расширения старых. Следствием того же является отказ от циклов в пользу рекурсии.

Сильные стороны

]Повышение надёжности кода

Привлекательная сторона вычислений без состояний — повышение надёжности кода за счёт чёткой структуризации и отсутствия необходимости отслеживания побочных эффектов. Любая функция работает только с локальными данными и работает с ними всегда одинаково, независимо от того, где, как и при каких обстоятельствах она вызывается. Невозможность мутации данных при пользовании ими в разных местах программы исключает появление труднообнаруживаемых ошибок (таких, например, как случайное присваивание неверного значения глобальной переменной в императивной программе).

Удобство организации модульного тестирования

Поскольку функция в функциональном программировании не может порождать побочные эффекты, менять объекты нельзя как внутри области видимости, так и снаружи (в отличие от императивных программ, где одна функция может установить какую-нибудь внешнюю переменную, считываемую второй функцией). Единственным эффектом от вычисления функции является возвращаемый ей результат, и единственный фактор, оказывающий влияние на результат — это значения аргументов.

Таким образом, имеется возможность протестировать каждую функцию в программе, просто вычислив её от различных наборов значений аргументов. При

этом можно не беспокоиться ни о вызове функций в правильном порядке, ни о правильном формировании внешнего состояния. Если любая функция в программе проходит модульные тесты, то можно быть уверенным в качестве всей программы. В императивных программах проверка возвращаемого значения функции недостаточна: функция может модифицировать внешнее состояние, которое тоже нужно проверять, чего не нужно делать в функциональных программах.

Возможности оптимизации при компиляции

Традиционно упоминаемой положительной особенностью функционального программирования является то, что оно позволяет описывать программу в так называемом «декларативном» виде, когда жесткая последовательность выполнения многих операций, необходимых для вычисления результата, в явном виде не задаётся, а формируется автоматически в процессе вычисления функций. Это обстоятельство, а также отсутствие состояний даёт возможность применять к функциональным программам достаточно сложные методы автоматической оптимизации.

Возможности параллелизма

Ещё одним преимуществом функциональных программ является то, что они предоставляют широчайшие возможности для автоматического распараллеливания вычислений. Поскольку отсутствие побочных эффектов гарантировано, в любом вызове функции всегда допустимо параллельное вычисление двух различных параметров — порядок их вычисления не может оказать влияния на результат вызова.

Недостатки

Недостатки функционального программирования вытекают из тех же самых его особенностей. Отсутствие присваиваний и замена их на порождение новых данных приводят к необходимости постоянного выделения и автоматического освобождения памяти, поэтому в системе исполнения функциональной программы обязательным компонентом становится высокоэффективный сборщик мусора. Нестрогая модель вычислений приводит к непредсказуемому порядку вызова функций, что создает проблемы при вводе-выводе, где порядок выполнения операций важен. Кроме того, очевидно, функции ввода в своем естественном виде (например, getchar из стандартной библиотеки языка C) не являются чистыми, поскольку способны возвращать различные значения для одних и тех же аргументов, и для устранения этого требуются определенные ухищрения.

Для преодоления недостатков функциональных программ уже первые языки функционального программирования включали не только чисто функциональные средства, но и механизмы императивного программирования (присваивание, цикл, «неявный PROGN» были уже в LISPe). Использование таких средств позволяет решить некоторые практические проблемы, но означает отход от идей (и преимуществ) функционального программирования и написание императивных программ на функциональных языках. В чистых функциональных языках эти проблемы решаются другими средствами, например, в языке Haskell ввод-вывод реализован при помощи монад — нетривиальной концепции, позаимствованной из теории категорий.

5 Логическое программирование

Логическое программирование - это парадигма программирования, в которой программы пишутся не в виде последовательности инструкций, а в виде множества фактов и правил, а процесс выполнения программы сводится к выводу нужных результатов из этого множества. Логическое программирование относится к декларативному программированию, поскольку программа на нём скорее описывает свойство задачи, нежели алгоритм её решения.

Основные языки логического программирования

Наиболее известным языком логического программирования является Prolog. Его наиболее известные реализации для платформы .NET - это Prolog.NET и P#. Оба языка позволяют компилировать Пролог-программу в множество .NET-классов на C#, а P# содержит также диалоговый интерпретатор. Тем, кто хочет разобраться во внутреннем устройстве Пролог-системы, будет интересен проект C#Prolog, реализация Пролог-интерпретатора на C#.

Вот как реализуется на Прологе вычисление факториала:

```
fact(1,1).
fact(N,F) :- N>1, N1 is N-1, fact(N1,F1), F is F1*N.
?- fact(4,X).
X=120
```

Среди современных языков логического программирования также следует упомянуть Mercury, разрабатываемый университетом Мельбурна. Это язык со строгой типизацией и мощными средствами проверки корректности программ.

6 Компонентное программирование

Допустим, компонентное программирование это - программирование из компонент. Что компонентным программированием вы именуете? Хороший вопрос:

Понятие 'компонентное программирование' полисемантично. И зависит от уровня, на котором рассматривается явление. В этом нет ничего необычного, само понятие 'программирование' при сохранении некоторой общей парадигмы деятельности имеет немалое число возможных интерпретаций в различных областях - от строго формальных до просто шарлатанских. Не менее полисемантично в этих областях и понятие 'компонента'.

В моем понимании, на самом верхнем уровне 'компонентное программирование' это - подход. Примерно такое же явление, как 'объектное программирование'. Только вот, 'объект' - понятие больше философского и модельного качества, а 'компонента' - конструкторского. Всякий, кто перед тем, как начать писать программы на C++ писал перед этим программы на C должен меня очень хорошо понять в этом определении. Ведь C++ не сильно от C и отличается. Синтаксис похож, компилятор - один и тот же... а какие разные языки! Разные, прежде всего тем, что для эффективного их использования программист должен мыслить разными категориями. Процедурный подход (C) кристаллизует 'чистое действие', а объектный (C++) - 'взаимодействие', т.е. C++ явно и формально вводит понятие 'объекта, в отношении которого совершается действие'. И переход в собственной голове от одной парадигмы мышления к другой - не такая и простая задача. Зато потом, когда парадигма моделирования взаимодействия объектов становится естественной и привычной обнаруживается, что эта парадигма намного больше, чем сам язык C++, что C++ - всего-навсего частная реализация 'средства моделирования'. А сам подход может быть назван не меньше, чем 'философией взгляда' и реализуется он исключительно в голове программиста - написать на C++ ужасную, непонятную и абсолютно ненадёжную программу намного легче, чем, скажем, на Visual Basic!

Всё то же можно сказать и про 'компонентное программирование' - это философия взгляда программиста на конструирование своих изделий. Конечно, сейчас существуют технологии и средства, которые построены на этой философии, но всё равно это остаётся философией, а не 'способом производства'. Которая, кстати, может быть реализована с использованием совершенно различных средств и в очень даже разных областях. И сведение компонентного программирования только к использованию специальных средств и технологий представляется мне не просто искусственным сужением взгляда, а даже и некоторой его неадекватностью. Почему?

Несколько ранее говорилось, что объектно-компонентный способ решения проблем - очень человеческий способ, соответствующий психологии восприятия человека. Но это ещё и очень человеческий способ в сугубо экономическом смысле - удачно спроектированные компоненты имеют высокую степень повторной используемости, т.е. очень существенно экономят труд программиста. Ведь создание программы есть не только 'написание кода', но и выдумывание её внутренней

конструкции, отладка, тестирование, создание документации, обучение других людей приёмам её использования. Создание повторно используемой компоненты экономит труд всех упомянутых людей. В том числе и труд программистов, которые используют компоненту - они могут использовать свои знания о ней во многих и многих случаях своей работы. Такая концептуальная ёмкость компонентного похода и заставляет говорить о нем как о прежде всего инженерной философии, а не о технологии или методе.

Поэтому здесь и ниже 'компонентное программирование' понимается как стремление программиста-конструктора углядеть существование компонент всюду, где только возможно. В машиностроении существует строгое определение детали: 'деталь - это изделие, изготовленное из однородного материала без применения сборочных операций'. Аналогичным образом можно определить понятие программной компоненты: программная компонента - это относительно самостоятельная часть программного комплекса, на которые вся конструкция может быть легко разъята, а отдельные части одного и того же назначения могут быть заменены между собой без применения операций повторной сборки модуля. Использование библиотек динамической компоновки в составе проекта - очень характерный пример этого.

Понятие 'компонента', как видно, не тождественно ни понятию 'процедура', ни понятию 'объект'. Она может ими быть, но компоненты есть то, на что готовая конструкция 'может быть разобрана в процессе эксплуатации'. Поэтому библиотечная процедура компонентой не является, компонентой является вся библиотека. Правда, тут надо сделать небольшую оговорку - всё это зависит от того, что считать 'готовой конструкцией', а это уже зависит от того, о каком изделии говорить. Если вы создаёте нечто, поставляемое в исходных текстах, то 'компонентой' может оказаться уже и процедура в составе библиотеки. Важно при этом понимать, что характерное отличие компоненты - свойство лёгкой взаимной заменяемости. Поэтому процедура в составе библиотеки - компонента, а вот класс - нет. И 'библиотека классов' (как пример - Microsoft Foundation Classes) не есть конструкция из компонент.

Вообще же у компоненты, какой бы она ни была и в каком бы виде ни существовала, обнаруживается устойчивое множество характерных черт.

Пожалуй, наиболее существенным нововведением идеологии Microsoft .NET является компонентно-ориентированный подход к программированию.

Во-первых, следует отметить то обстоятельство, что компонентно-ориентированный подход к проектированию и реализации программных систем и комплексов является в некотором смысле развитием объектно-ориентированного и практически более пригоден для разработки крупных и распределенных систем (например, корпоративных приложений).

Прежде всего, сформулируем основополагающее для рассматриваемого подхода определение компонента. Под компонентом будем далее иметь в виду независимый модульпрограммного кода, предназначенный для повторного использования и развертывания. Как видно из определения, применение компонентного программирования призвано обеспечить более простую, быструю и прямолиней-

ную процедуру первоначальной инсталляции прикладного программного обеспечения, а также увеличить процент повторного использования кода, т.е. усилить основные преимущества ООП.

Говоря о свойствах компонентов, следует прежде всего отметить, что это существенно более крупные единицы, чем объекты (в том смысле, что объект представляет собой конструкцию уровня языка программирования). Другими отличиями компонентов от традиционных объектов являются возможность содержать множественные классы и (в большинстве случаев) независимость от языка программирования.

Заметим, что, автор и пользователь компонента, вообще говоря, территориально распределены и используют разные языки. Вполне возможно, что они не только пишут программы, но и говорят на разных языках.

Получив представление о компонентах и их отличиях от традиционных объектов ООП, рассмотрим существующие подходы к моделированию вариаций компонентного подхода в современной практике проектирования и реализации программных комплексов и систем.

Прежде всего, поскольку основной вычислительной средой для исследования в данном курсе является Microsoft .NET, обсудим особенности модели для компонентно-ориентированной разработки программного обеспечения, предложенной корпорацией Microsoft. Эта модель называется компонентной объектной моделью (или Component Object Model, COM) и является изначальным стандартом, принятым для компонентной разработки приложений в корпорации Microsoft.

Компонентная модель COM определяет протокол для конкретизации (т.е. создания экземпляров) и использования компонент (по аналогии с классами и объектами) как внутри одного и того же процесса, так и между различными процессами или компьютерами, предназначенными для выполнения того или иного программного проекта, основанного на компонентной технологии. Модель COM является достаточно универсальной и используется в качестве фундамента для таких технологий проектирования и реализации программного обеспечения, как ActiveX, OLE и целого ряда других технологий. Приложения для COM-модели могут создаваться средствами таких языков и сред разработки как Visual Basic, C++, .NET и т.д.

Другим известным стандартом для компонентной модели является стандарт компании Sun Microsystems, известный как JavaBeans, который не обладает свойством независимости от языка программирования. Еще один широко используемый стандарт компонентного программирования – архитектура объектных запросов CORBA (несмотря на поддержку многоязычной разработки приложений, существуют сложности, связанные с отображением одного языка реализации в другой; для этой цели применяется достаточно громоздкий интерфейс, основанный на специальном языке описания IDL).

В рамках компонентного подхода сборкой называется логическая единица, содержащая множество модулей, необходимых для осуществления инсталляции программного обеспечения. Сборка характеризуется уникальностью, которая обеспечивается идентификатором версии сборки и цифровой подписью автора. Сборка является самодостаточной единицей для установки про-

граммного обеспечения и не требует никаких дополнений. Возможно как индивидуальное, так и коллективное (сетевое) использование сборки на основе компонентной технологии. Сборка обеспечивает простой и удобный механизм инсталляции и экономит средства, необходимые дляразвертывания программного обеспечения, сводя к минимуму затраты времени и сил на установку.

Описание сборки содержится в так называемом манифесте, где хранятся метаданные о компонентах сборки, идентификация автора и версии, сведения о типах и зависимостях, а также режим и политика использования. Метаданные типов манифеста исчерпывающе описывают все типы, определенные в сборке, а именно, свойства, методы, аргументы, возвращаемые значения, атрибуты, базовые классы и т.д.

Заметим, что промежуточный язык IL всегда компилируется в естественный (native) код до выполнения программы.

Для более эффективного манипулирования системой типизации компонент создаваемого программного обеспечения в рамках модели COM, концепция .NET предусматривает механизм пространств имен (namespace).

Пространством имен будем называть механизм среды Microsoft .NET, предназначенный для идентификации типов объектов языков программирования и среды реализации.

Описания пространств имен по аналогии с описаниями типов данных размещаются в файлах. Перечислим основные свойства, которыми характеризуются пространства имен в среде Microsoft .NET. Прежде всего, пространства имен могут как объединять различные сборки, так и быть вложенными друг в друга. Кроме того, файлы с описаниями могут содержать множественные пространства имен. Важно отметить, что между пространствами имен и файлами не существует однозначного соответствия. Наконец, полное имя типа должно содержать все необходимые пространства имен.

Приведем пример файла xxx.cs с описанием множественных пространств имен на языке C#:

```
xxx.cs
namespace A {
...
}
namespace B {
...
}
namespace C {
...
}
```

Приведем пример файла xxx.cs с описанием пространств имен на языке C# с неоднозначным (с учетом предыдущего примера) соответствием файлов и пространств имен:

```
xxx.cs
namespace A {
```

```
class C {
...
}
}
```

Кроме свойств, перечисленных выше, механизм пространств имен в среде вычислений .NET обладает еще целым рядом важных особенностей.

Так, допускается импорт пространств имен с использованием зарезервированного слова using языка программирования C# (похожий подход реализован в модулях языка Modula-2). Проиллюстрируем эту особенность фрагментом программы на языке C#:

```
using System;
```

Кроме того, существует возможность импорта одних пространств имен в другие. Проиллюстрируем это свойство фрагментом программы на языке C#:

```
using A;
namespace B {
using C;
...
}
```

Отметим также, что для явного указания полных и сокращенных имен разрешаются псевдонимы (alias). Проиллюстрируем это свойство следующим примером программы на языке C#:

```
using F = System.Windows.Forms;
...
F.Button b;
```

Как видно из приведенного фрагмента программы, для удобства вызова стандартного пространства имен .NET под именем System.Windows.Forms применяется псевдоним F.

Рассмотрим особенности использования механизма сборок, важнейшей концепции компонентного программирования, применительно к языку C#.

Сборка является минимальной единицей для развертывания приложений, т.е. представляет собой своеобразный атом компонентного программирования.

Каждый тип сборки характеризуется уникальным идентификатором – номером версии сборки. Таким образом, каждый программный проект формируется в виде сборки, которая является самодостаточным компонентом для развертывания, тиражирования и повторного использования. Сборка идентифицируется цифровой подписью автора и уникальным номером версии.

Между сборками и пространствами имен существует следующее соотношение. Сборка может содержать несколько пространств имен. В то же время, пространство имен может занимать несколько сборок.

Сборка может иметь в своем составе как один, так и несколько файлов, которые объединяются в так называемом манифесте или описании сборки, который подобен оглавлению книги. Манифест содержит метаданные о компонентах сборки, идентификацию автора и версии, сведения о типах и зависимостях, а также описывает режим и политику использования сборки. Метаданные типов манифеста в полной мере описывают все типы, которые содержатся в сборке.

В ходе обсуждения реализации механизма сборок в языке программирования C# следует отметить еще несколько важных характеристик.

Прежде всего, сборка как элемент языка программирования C# является аналогом компонента в среде проектирования и реализации программного обеспечения Microsoft .NET.

В результате компиляции программного кода на языке C# в среде вычислений .NET (например, в .NET Framework SDK) создается либо сборка, либо так называемый модуль.

При этом сборка существует в форме исполняемого файла (с расширением EXE), а также файла динамически присоединяемой библиотеки (с расширением DLL). Естественно, в состав сборки входит манифест. Модуль представляет собой файл с расширением .NETMODULE и, в отличие от сборки, не содержит в своем составе манифеста.

Заметим, что интеграция в программный проект других модулей и ресурсов (в частности, типов и метаданных) может быть осуществлена посредством системного программного обеспечения, известного под названием компоновщика сборок.

Подводя итоги рассмотрения основных аспектов концепции компонентно-ориентированного подхода к программированию и особенностей реализации этой концепции применительно к языку программирования C#, кратко отметим достоинства подхода.

Прежде всего, важным практическим следствием реализации концепции компонентного подхода для экономики программирования является снижение стоимости проектирования и реализации программного обеспечения.

Еще одно очевидное достоинство компонентного программирования – возможность усовершенствования стратегии повторного использования кода. Код с более высокимуровнем абстракции не требует существенной модификации при адаптации к изменившимся условиям задачи или новым типам данных.

Кроме того, к преимуществам концепции компонентного программирования следует отнести унификацию обработки объектов различной природы. В самом деле, абстрактные классы и методы позволяют единообразно оперировать гетерогенными данными, причем для адаптации к новым классам и типам данных не требуется реализации дополнительного программного кода.

Важно также отметить, что идеология компонентного программирования основана на строгом математическом фундаменте (в частности, в виде формальной системы ламбда-исчисления), что обеспечивает интуитивную прозрачность исходного текста для математически мыслящего программиста, а также верифицируемость программного кода.

Наконец, концепция компонентного программирования, по сути, универсальна и в равной степени применима для различных подходов к программированию, включая функциональный и объектно-ориентированный.

Завершая вторую часть учебного курса, посвященного исследованию современных языков программирования (на примере языка программирования C#) и поддерживающих их сред вычислений (на примере инструментально-технологической платформы .NET), кратко резюмируем содержание рассмотренных вопросов и проблем.

Прежде всего, нами был рассмотрен объектно-ориентированный подход к проектированию и реализации программного обеспечения в сопоставлении с другими подходами.

Затем было дано представление о важнейших концепциях, которые составляют теоретическое и практическое основание объектно-ориентированного подхода к программированию. В частности, были рассмотрены концепции объекта, класса, метода, абстракции, инкапсуляции, наследования, полиморфизма, а также подходы к их формализации на основе исчисления ламбда-конверсий, комбинаторной логики, теории решеток и концептуализации.

Далее, посредством перечисленных теорий были формализованы такие важнейшие аспекты объектно-ориентированных языков программирования, как синтаксис и семантика.

Кроме того, было исследовано понятие типа, изучены основы теории типов и типизации в языках программирования, реализованных в среде вычислений .NET.

Затем мы перешли к рассмотрению вопросов, связанных с событийно-ориентированным программированием.

Наконец, было исследована трансформация ООП в приложении к среде .NET, которая привела к появлению компонентного подхода, основанного на объектной модели COM.

Вполне естественно, что, исходя из обширного спектра и глубины рассматриваемой проблематики, ряд важнейших аспектов реализации современных языков программирования (в рамках изложенного курса) был лишь обозначен или изложен весьма конспективно.

В связи с этим, представляет интерес систематическое изучение следующих вопросов:

компонентная разработка интегрированных гетерогенных программных систем на профессиональном уровне (на примере языков программирования SML и C#);

разработка событийно-управляемых приложений;

математически строгий сравнительный анализ функционального и объектно-ориентированного подходов к программированию на основе изученных теоретических разделов computer science;

формализация семантики событийно управляемого (и, возможно, компонентно-ориентированного) программирования.

Согласно общей концепции изложения курса, дальнейшие исследования имеет смысл вести синтетически по направлениям теоретического обоснования программирования на основе изученных формальных систем computer science и современной практики проектирования и реализации программного обеспечения на основе универсальной и прогрессивной программно-инструментальной платформы Microsoft .NET.

7 Распределенное программирование

Развитие современной вычислительной аппаратуры характеризуется четко выраженной тенденцией распространения многопроцессорных компьютеров и вычислительных сетей как локальных, так и глобальных. Уже недалеко то время, когда даже персональные компьютеры будут иметь несколько процессоров а самые мощные из них могут даже иметь сетевую архитектуру.

Поэтому в области программного обеспечения вызывают все больший интерес языки и другие инструментальные средства, поддерживающие разработку распределенных программ. При этом можно выделить три уровня распределенного программирования:

- распределенное программирование в малом;
- распределенное программирование в большом;
- распределенные прикладные программные системы.

Распределенное программирование в малом связано с параллельным выполнением операторов одной программы (например, параллельным выполнением циклов). Технология разработки распределенных программ связана с реализацией коммуникационных пакетов, а также языков программирования и библиотек, облегчающих использование таких пакетов. Примерами коммуникационных пакетов могут служить такие широко используемые пакеты как PVM и MPI. Из языков, опирающихся на эти пакеты следует выделить прежде всего HPF/2 (High Performance Fortran, version 2) и язык, разработанный в ИСП РАН, - mpC. Среди многочисленных библиотек наиболее интересны различные алгебраические библиотеки (например, ScalaPack), а также библиотеки, вводящие параллельные конструкции в стандартные языки программирования (одна из таких библиотек DPJ - Data Parallel Java - разрабатывается в ИСП РАН).

Распределенное программирование в большом состоит в разработке распределенных программных систем. В последнее время широко используются объектно-ориентированные технологии разработки таких систем. Наиболее известные такие технологии связаны с системой стандартов OMG (стандарт CORBA).

Распределенные прикладные программные системы предназначены для поддержки коллективных разработок и других видов коллективной деятельности в самых различных областях. В таких системах используются новейшие достижения в области вычислительной техники и коммуникаций. Примером такой системы может служить система POLITeam (GMD-FIT, Германия), основной целью которой является обеспечение оперативной работы правительства Германии, расположенного в двух городах - Берлине и Бонне, в настоящее время имеются в наличии как аппаратные, так и программные средства, позволившие реализовать и успешно эксплуатировать такую систему. С проектом POLITeam связано одно из ведущих научных направлений GMD - "Системы поддержки совместных разработок на основе коммуникации" (Communication and Cooperation Systems). Это одно из интенсивно развивающихся научно-исследовательских и прикладных направлений современной информатики. Широко известны такие системы для организации совместной работы как система Totem, разработанная в Университете г. Санта-Барбара, Калифорния, США, система Transis, разработанная в Университете г.

Иерусалима, Израиль, система Horus Корнельского Университета (США) и многие другие. При разработке и реализации перечисленных систем применялись различные подходы, что позволяет разработать их аналитический обзор с целью выбора наиболее дешевого и приемлемого подхода. Это относится и к инструментальным программным средствам, используемым для реализации систем поддержки совместных разработок. Одним из широко известных инструментальных средств, используемых при разработке и реализации таких систем, является пакет Rampart Toolkit, разработанный в AT&T Bell Laboratories, Нью-Джерси, США. Для реализации системы поддержки совместных разработок в рамках проекта POLITeam использовалась другая не менее известная инструментальная система LinkWorks фирмы Digital Equipment Corporation, США.

8 Стандарты жизненного цикла (ISO, IEEE, CMM).

Чтобы получить представление о возможной структуре жизненного цикла ПО, обратимся сначала к соответствующим стандартам, описывающим технологические процессы. Международными организациями, такими, как:

• IEEE — читается «ай-трипл-и», Institute of Electrical and Electronic Engineers, Институт инженеров по электротехнике и электронике;

• ISO — International Standards Organization, Международная организация по стандартизации;

• EIA — Electronic Industry Association, Ассоциация электронной промышленности;

• IEC — International Electrotechnical Commission, Международная комиссия по электротехнике;

а также некоторыми национальными и региональными институтами и организациями (в основном, американскими и европейскими, поскольку именно они оказывают наибольшее влияние на развитие технологий разработки ПО во всем мире):

• ANSI — American National Standards Institute, Американский национальный институт стандартов;

• SEI — Software Engineering Institute, Институт программной инженерии;

• ECMA — European Computer Manufactures Association, Европейская ассоциация производителей компьютерного оборудования;

разработан набор стандартов, регламентирующих различные аспекты жизненного цикла и вовлеченных в него процессов. Список и общее содержание этих стандартов представлены ниже.

Группа стандартов ISO

• **ISO/IEC 12207 Standard for Information Technology — Software Life Cycle Processes** [1] (процессы жизненного цикла ПО, есть его российский аналог **ГОСТ Р-1999** [2]). Определяет общую структуру жизненного цикла ПО в виде 3-х ступенчатой модели, состоящей из процессов, видов деятельности и задач. Стандарт описывает вводимые элементы в терминах их целей и результатов, тем самым задавая неявно возможные взаимосвязи между ними, но не определяя четко структуру этих связей, возможную организацию элементов в рамках проекта и метрики, по которым можно было бы отслеживать ход работ и их результативность.

Самыми крупными элементами являются *процессы жизненного цикла ПО (lifecycle processes).* Всего выделено 18 процессов, которые объединены в 4 группы.

Основные процессы	Поддерживающие процессы	Организационные процессы	Адаптация
Приобретение ПО; Передача ПО (в использование); Разработка ПО; Эксплуатация ПО; Поддержка ПО	Документирование; Управление конфигурациями; Обеспечение качества; Верификация; Валидация; Совместные экспертизы; Аудит; Разрешение проблем	Управление проектом; Управление инфраструктурой; Усовершенствование процессов; Управление персоналом	Адаптация описываемых стандартом процессов под нужды конкретного проекта

Процессы строятся из отдельных *видов деятельности (activities)*. Стандартом определены 74 вида деятельности, связанной с разработкой и поддержкой ПО. Здесь мы упомянем только некоторые из них.

- Приобретение ПО включает такие деятельности, как инициация приобретения, подготовка запроса предложений, подготовка контракта, анализ поставщиков, получение ПО и завершение приобретения.

- Разработка ПО включает развертывание процесса разработки, анализ системных требований, проектирование (программно-аппаратной) системы в целом, анализ требований к ПО, проектирование архитектуры ПО, детальное проектирование, кодирование и отладочное тестирование, интеграцию ПО, квалификационное тестирование ПО, системную интеграцию, квалификационное тестирование системы, развертывание (установку или инсталляцию) ПО, поддержку процесса получения ПО.
- Поддержка ПО включает развертывание процесса поддержки, анализ возникающих проблем и необходимых изменений, внесение изменений, экспертизу и передачу измененного ПО, перенос ПО с одной платформы на другую, изъятие ПО из эксплуатации.
- Управление проектом включает запуск проекта и определение его рамок, планирование, выполнение проекта и надзор за его выполнением, экспертизу и оценку проекта, свертывание проекта.

Каждый вид деятельности нацелен на решение одной или нескольких *задач (tasks)*. Всего определено 224 различные задачи. Например:

- Развертывание процесса разработки состоит из определения модели жизненного цикла, документирования и контроля результатов отдельных работ, выбора используемых стандартов, языков и инструментов и пр.
- Перенос ПО между платформами состоит из разработки плана переноса, оповещения пользователей, выполнения анализа произведенных действий и пр.

• **ISO/IEC 15288 Standard for Systems Engineering — System Life Cycle Processes** [3] (процессы жизненного цикла систем). Отличается от предыдущего нацеленностью на рассмотрение программно-аппаратных систем в целом. В данный момент продолжается работа по приведению этого стандарта в соответствие с предыдущим. ISO/IEC 15288 предлагает похожую схему рассмотрения жизненного цикла системы в виде набора процессов. Каждый процесс описывается набором его *результатов (outcomes)*, которые достигаются при помощи различных видов деятельности.

Всего выделено 26 процессов, объединяемых в 5 групп.

Процессы выработки соглашений	**Процессы уровня организации**	**Процессы уровня проекта**	**Технические процессы**	**Специальные процессы**
Приобретение системы; Поставка системы	Управление окружением; Управление инвестициями; Управление процессами; Управление ресурсами; Управление качеством	Планирование; Оценивание; Мониторинг; Управление рисками; Управление конфигурацией; Управление информацией; Выработка решений	Определение требований; Анализ требований; Проектирование архитектуры; Реализация; Интеграция; Верификация; Валидация; Передача в использование; Эксплуатация; Поддержка; Изъятие из эксплуатации	Адаптация описываемых стандартом процессов под нужды конкретного проекта

Таблица 2. Процессы жизненного цикла систем по ISO 15288.

Помимо процессов, определено 123 различных результата и 208 видов деятельности, нацеленных на их достижение. Например, определение требований имеет следующие результаты.

- Должны быть поставлены технические задачи, которые предстоит решить.
- Должны быть сформулированы системные требования.

Деятельности в рамках этого процесса следующие.

- Определение границ функциональности системы.
- Определение функций, которые необходимо поддерживать.
- Определение критериев оценки качества при использовании системы.
- Анализ и выделение требований по безопасности.
- Анализ требований защищенности.
- Выделение критических для данной системы аспектов качества и требований к ним.

- Анализ целостности системных требований.
- Демонстрация прослеживаемости требований.
- Фиксация и поддержка набора системных требований.

• **ISO/IEC 15504 (SPICE) Standard for Information Technology — Software Process Assessment** [4] (оценка процессов разработки и поддержки ПО). Определяет правила оценки процессов жизненного цикла ПО и их возможностей, опирается на модель CMMI (см. ниже) и больше ориентирован на оценку процессов и возможностей их улучшения. В качестве основы для оценки процессов определяет некоторую базовую модель, аналогичную двум описанным выше. В ней выделены категории процессов, процессы и виды деятельности.

Определяются 5 категорий, включающих 35 процессов и 201 вид деятельности.

Отношения заказчик-поставщик	**Процессы уровня организации**	**Процессы уровня проекта**	**Инженерные процессы**	**Процессы поддержки**
Приобретение ПО; Составление контракта; Определение нужд заказчика; Проведение совместных экспертиз и аудитов; Подготовка к передаче; Поставка и развертывание; Поддержка эксплуатации; Предоставление услуг; Оценка удовлетворенности заказчиков	Развитие бизнеса; Определение процессов; Усовершенствование процессов; Обучение; Обеспечение переиспользования; Обеспечение инструментами; Обеспечение среды для работы	Планирование жизненного цикла; Планирование проекта; Построение команды; Управление требованиями; Управление качеством; Управление рисками; Управление ресурсами и графиком работ; Управление подрядчиками	Выделение системных требований и проектирование системы в целом; Выделение требований к ПО; Проектирование ПО; Реализация, интеграция и тестирование ПО; Интеграция и тестирование системы; Сопровождение системы и ПО	Разработка документации; Управление конфигурацией; Обеспечение качества; Разрешение проблем; Проведение экспертиз

Таблица 3. Процессы жизненного цикла ПО и систем по ISO 15504.

Например, приобретение ПО включает такие виды деятельности, как определение потребности в ПО, определение требований, подготовку стратегии покупки, подготовку запроса предложений, выбор поставщика.

Группа стандартов IEEE

• **IEEE 1074-1997 — IEEE Standard for Developing Software Life Cycle Processes** [5] (стандарт на создание процессов жизненного цикла ПО). Нацелен на описание того, как создать специализированный процесс разработки в рамках конкретного проекта. Описывает ограничения, которым должен удовлетворять

любой такой процесс, и, в частности, общую структуру процесса разработки. В рамках этой структуры определяет основные виды деятельностей, выполняемых в этих процессах и документы, требующиеся на входе и возникающие на выходе этих деятельностей. Всего рассматриваются 5 подпроцессов, 17 групп деятельностей и 65 видов деятельности. Например, подпроцесс разработки состоит из групп деятельностей по выделению требований, по проектированию и по реализации. Группа деятельностей по проектированию включает архитектурное проектирование, проектирование баз данных, проектирование интерфейсов, детальное проектирование компонентов.

• **IEEE/EIA 12207-1997 — IEEE/EIA Standard: Industry Implementation of International Standard ISO/IEC 12207:1995 Software Life Cycle Processes** [6-8] (промышленное использование стандарта ISO/IEC 12207 на процессы жизненного цикла ПО). Аналог ISO/IEC 12207, сменил ранее использовавшиеся стандарты J-Std-016-1995 EIA/IEEE Interim Standard for Information Technology — Software Life Cycle Processes — Software Development Acquirer-Supplier Agreement (промежуточный стандарт на процессы жизненного цикла ПО и соглашения между поставщиком и заказчикам ПО) и стандарт министерства обороны США MIL-STD-498.

Группа стандартов CMM, разработанных SEI

• **Модель зрелости возможностей CMM (Capability Maturity Model)** [9,10] предлагает унифицированный подход к оценке возможностей организации выполнять задачи различного уровня. Для этого определяются 3 уровня элементов: *уровни зрелости организации (maturity levels), ключевые области процесса (key process areas)* и *ключевые практики (key practices)*. Чаще всего под моделью CMM имеют в виду модель уровней зрелости. В настоящий момент CMM считается устаревающей и сменяется моделью CMMI (см. ниже).

o Уровни зрелости. CMM описывает различные степени зрелости процессов в организациях, определяя 5 уровней организаций.

- Уровень 1, начальный (initial). Организации, разрабатывающие ПО, но не имеющие осознанного процесса разработки, не производящие планирования и оценок своих возможностей.
- Уровень 2, повторяемый (repeatable). В таких организациях ведется учет затрат ресурсов и отслеживается ход проектов, установлены правила управления проектами, основанные на имеющемся опыте.
- Уровень 3, определенный (defined). В таких организациях имеется принятый, полностью документированный, соответствующий реальному положению дел и доступный персоналу процесс разработки и сопровождения ПО. Он должен включать как управленческие, так и технические подпроцессы, а также обучение сотрудников работе с ним.
- Уровень 4, управляемый (manageable). В этих организациях, помимо установленного и описанного процесса, используются измеримые показатели качества продуктов и результативности процессов, которые позволяют достаточно точно предсказывать объем ресурсов (времени, денег, персонала), необходимый для разработки продукта с определенным качеством.
- Уровень 5, совершенствующийся (optimizing). В таких организациях, помимо процессов и методов их оценки, имеются методы определения слабых

мест, определены процедуры поиска и оценки новых методов и техник разработки, обучения персонала работе с ними и их включения в общий процесс организации в случае повышения эффективности производства.

- Ключевые области процесса. Согласно CMM, уровни зрелости организации можно определять по использованию четко определенных техник и процедур, относящихся к различным ключевым областям процесса. Каждая такая область представляет собой набор связанных видов деятельности, которые нацелены на достижение целей, существенных для общей оценки результативности технологического процесса. Всего выделяется 18 областей. Множество областей, которые должны поддерживаться организацией, расширяется при переходе к более высоким уровням зрелости.
  - К первому уровню не предъявляется никаких требований.
  - Организации второго уровня зрелости должны поддерживать управление требованиями, планирование проектов, надзор за ходом проекта, управление подрядчиками, обеспечение качества ПО, управление конфигурацией.
  - Организации третьего уровня должны, помимо деятельностей второго уровня, поддерживать проведение экспертиз, координацию деятельности отдельных групп, разработку программного продукта, интегрированное управление разработкой и сопровождением, обучение персонала, выработку и поддержку технологического процесса организации, контроль соблюдения технологического процесса организации.
  - К деятельностям организаций четвертого уровня добавляются: управление качеством ПО и управление процессом, основанное на измеримых показателях.
  - Организации пятого уровня зрелости должны дополнительно поддерживать управление изменениями процесса, управление изменениями используемых технологий и предотвращение дефектов.
- Ключевые практики. Ключевые области процесса описываются с помощью наборов ключевых практик. Ключевые практики классифицированы на несколько видов: обязательства (commitments to perform), возможности (abilities to perform), деятельности (activities performed), измерения (measurements and analysis) и проверки (verifying implementations). Например, управление требованиями связано со следующими практиками.
  - Обязательство. Проекты должны следовать определенной политике организации по управлению требованиями.
  - Возможности. В каждом проекте должен определяться ответственный за анализ системных требований и привязку их к аппаратному, программному обеспечению и другим компонентам системы. Требования должны быть документированы. Для управления требованиями должны быть выделены адекватные ресурсы и бюджет. Персонал должен проходить обучение в области управления требованиями.
  - Деятельности. Прежде чем быть включенными в проект, требования подвергаются анализу на полноту, адекватность, непротиворечивость и пр. Выделенные требования используются в качестве основы для планирования и выполне-

ния других работ. Изменения в требованиях анализируются и включаются в проект.

- Измерение. Производится периодическое определение статуса требований и статуса деятельности по управлению ими.
- Проверки. Деятельность по управлению требованиями периодически анализируется старшими менеджерами. Деятельность по управлению требованиями периодически и на основании значимых событий анализируется менеджером проекта. Группа обеспечения качества проводит анализ и аудит деятельности по управлению требованиями и отчитывается по результатам этого анализа.

Таблица 4 суммирует информацию о количестве практик различных видов, приписанных к разным ключевым областям процесса.

Уровни	Область процесса	Обязательства	Возможности	Деятельности	Измерения	Проверки
2	Управление требованиями	1	4	3	1	3
	Планирование проектов	2	4	15	1	3
	Надзор за ходом проекта	2	5	13	1	3
	Управление подрядчиками	2	3	13	1	3
	Обеспечение качества ПО	1	4	8	1	3
	Управление конфигурацией	1	5	10	1	4
3	Контроль соблюдения технологического процесса	3	4	7	1	1
	Выработка и поддержка технологического процесса	1	2	6	1	1
	Обучение персонала	1	4	6	2	3
	Интегрированное управление	1	3	11	1	3
	Разработка программного продукта	1	4	10	2	3
	Координация деятельности	1	5	7	1	3

	групп					
	Проведение экспертиз	1	3	3	1	1
4	Управление процессом на основе метрик	2	5	7	1	3
	Управление качеством ПО	1	3	5	1	3
5	Предотвращение дефектов	2	4	8	1	3
	Управление изменениями технологий	3	5	8	1	2
	Управление изменениями процесса	2	4	10	1	2

Таблица 4. Количество ключевых практик в разных областях процесса по CMM версии 1.1.

9 Унифицированный процесс Rational.

Унифицированный процесс Rational *(Rational Unified Process, RUP)* является примером так называемого *«тяжелого»* процесса, детально описанного и предполагающего поддержку собственно разработки исходного кода ПО большим количеством вспомогательных действий. Примерами подобных действий являются разработка планов, технических заданий, многочисленных проектных моделей, проектной документации, и пр. Основная цель такого процесса — отделить успешные практики разработки и сопровождения ПО от конкретных людей, умеющих их применять. Многочисленные вспомогательные действия дают надежду сделать возможным успешное решение задач по конструированию и поддержке сложных систем с помощью имеющихся работников, не обязательно являющихся суперпрофессионалами.

Для достижения этого выполняется иерархическое пошаговое детальное описание предпринимаемых в той или иной ситуации действий, чтобы можно было научить обычного работника действовать аналогичным образом. В ходе проекта создается много промежуточных документов, позволяющих разработчикам последовательно разбивать стоящие перед ними задачи на более простые. Эти же документы служат для проверки правильности решений, принимаемых на каждом шаге, а также отслеживания общего хода работ и уточнения оценок ресурсов, необходимых для получения желаемых результатов.

Экстремальное программирование, наоборот, представляет так называемые *«живые» (agile)* методы разработки, называемые также *«легкими»* процессами. Они делают упор на использовании хороших разработчиков, а не хорошо отлаженных процессов разработки. Живые методы избегают фиксации четких схем действий, чтобы обеспечить б***о***льшую гибкость в каждом конкретном проекте, а также выступают против разработки дополнительных документов, не вносящих непосредственного вклада в получение готовой работающей программы.

Унифицированный процесс Rational

RUP [1,2] является довольно сложной, детально проработанной итеративной моделью жизненного цикла ПО.

Исторически RUP является развитием модели процесса разработки, принятой в компании Ericsson в 70-х–80-х годах XX века. Эта модель была создана Джекобсоном (Ivar Jacobson), впоследствии, в 1987, основавшим собственную компанию Objectory AB именно для развития технологического процесса разработки ПО как отдельного продукта, который можно было бы переносить в другие организации. После вливания Objectory в Rational в 1995 разработки Джекобсона были интегрированы с работами Ройса (Walker Royce, сын автора «классической» каскадной модели), Крухтена (Philippe Kruchten) и Буча (Grady Booch), а также с развивавшимся параллельно *универсальным языком моделирования (Unified Modeling Language, UML).*

RUP основан на трех ключевых идеях.

- Весь ход работ направляется итоговыми целями проекта, выраженными в виде ***вариантов использования (use cases)*** — сценариев взаимодействия результирующей программной системы с пользователями или другими системами, при

выполнении которых пользователи получают значимые для них результаты и услуги. Разработка начинается с выделения вариантов использования и на каждом шаге контролируется степенью приближения к их реализации.

• Основным решением, принимаемым в ходе проекта, является ***архитектура*** результирующей программной системы. Архитектура устанавливает набор компонентов, из которых будет построено ПО, ответственность каждого из компонентов (т.е. решаемые им подзадачи в рамках общих задач системы), четко определяет интерфейсы, через которые они могут взаимодействовать, а также способы взаимодействия компонентов друг с другом.

Архитектура является одновременно основой для получения качественного ПО и базой для планирования работ и оценок проекта в терминах времени и ресурсов, необходимых для достижения определенных результатов. Она оформляется в виде набора графических моделей на языке UML.

• Основой процесса разработки являются *планируемые и управляемые итерации*, объем которых (реализуемая в рамках итерации функциональность и набор компонентов) определяется на основе архитектуры.

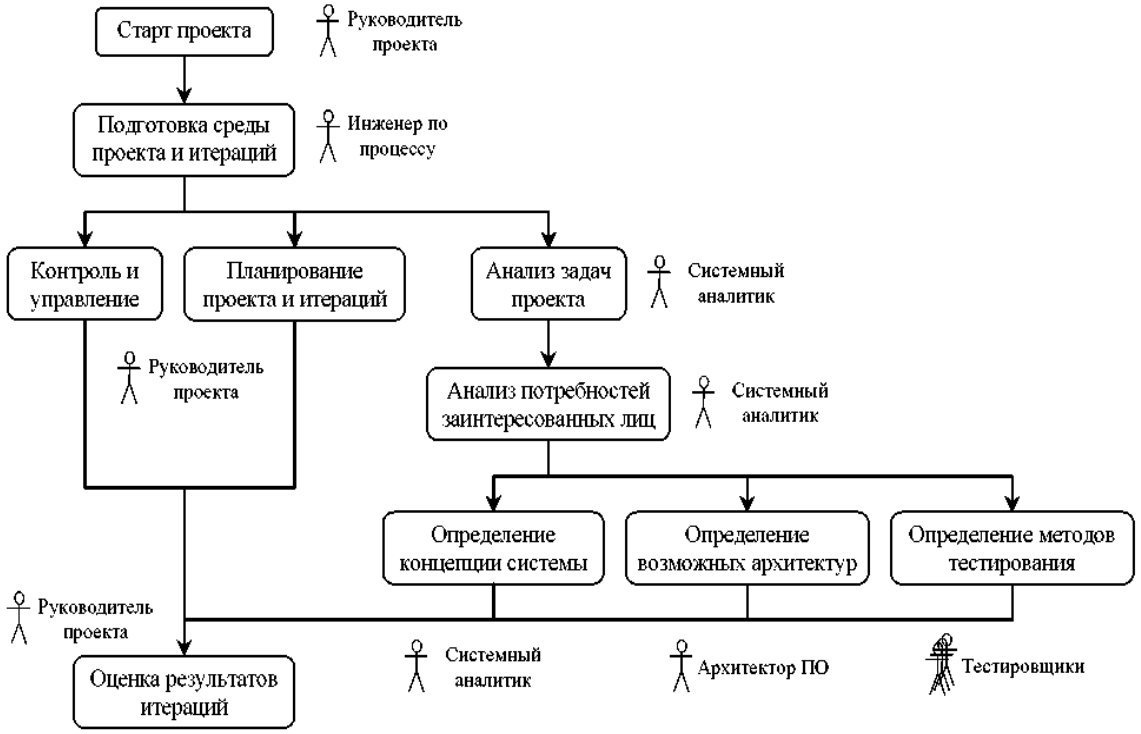


Рисунок 7. Пример хода работ на фазе начала проекта.

RUP выделяет в жизненном цикле на 4 основные фазы, в рамках каждой из которых возможно проведение нескольких итераций. Кроме того, разработка системы может пройти через несколько циклов, включающих все 4 фазы.

1. **Фаза начала проекта (Inception)**

Основная цель этой фазы — достичь компромисса между всеми заинтересованными лицами относительно задач проекта и выделяемых на него ресурсов. На этой стадии определяются основные цели проекта, руководитель и бюджет, основные средства выполнения — технологии, инструменты, ключевые исполните-

ли. Также, возможно, происходит апробация выбранных технологий, чтобы убедиться в возможности достичь целей с их помощью, и составляются предварительные планы проекта.

На эту фазу может уходить около 10% времени и 5% трудоемкости одного цикла. Пример хода работ показан на Рис. 7.

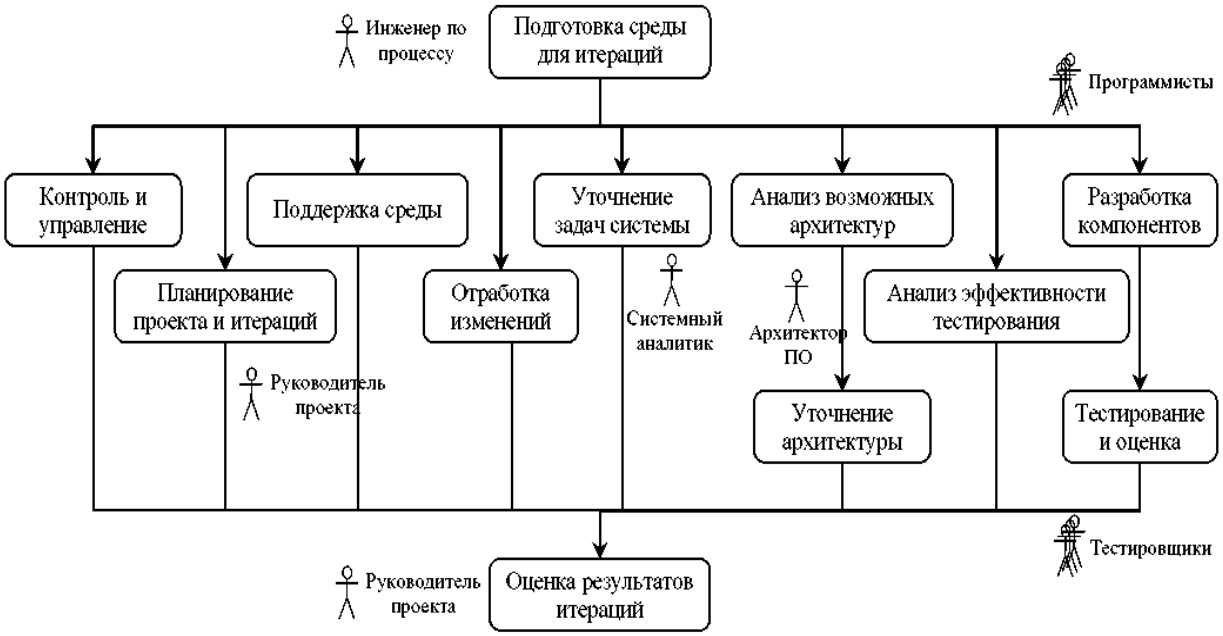


Рисунок 8. Пример хода работ на фазе проектирования.

2. **Фаза проектирования (Elaboration)**

Основная цель этой фазы — на базе основных, наиболее существенных требований разработать стабильную базовую архитектуру продукта, которая позволяет решать поставленные перед системой задачи и в дальнейшем используется как основа разработки системы.

На эту фазу может уходить около 30% времени и 20% трудоемкости одного цикла. Пример хода работ представлен на Рис. 8.

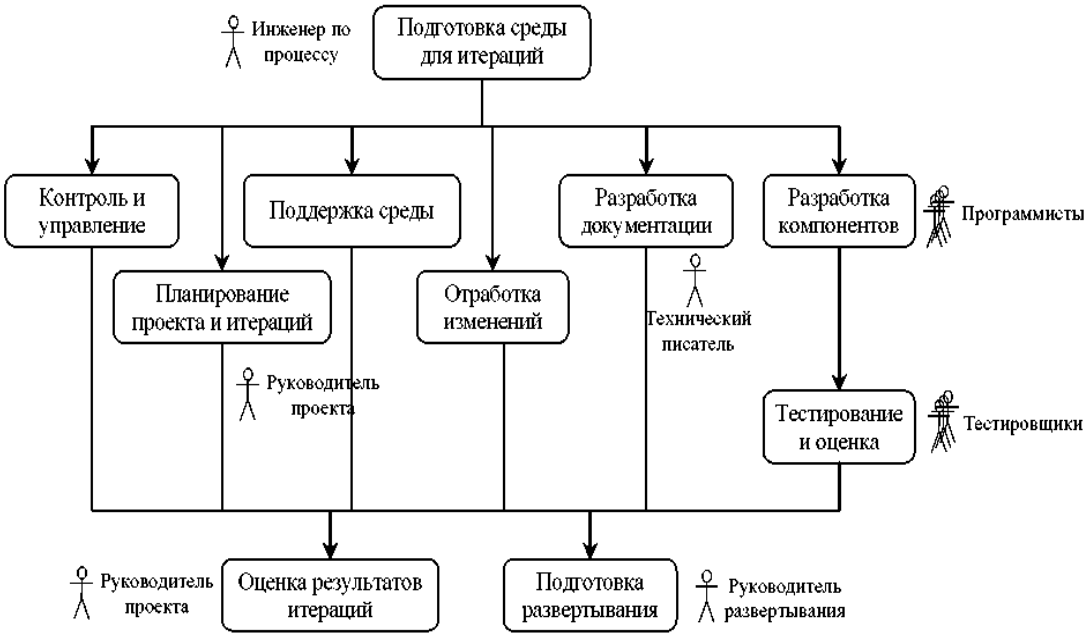


Рисунок 9. Пример хода работ на фазе построения.

3. **Фаза построения (Construction)**

Основная цель этой фазы — детальное прояснение требований и разработка системы, удовлетворяющей им, на основе спроектированной ранее архитектуры. В результате должна получиться система, реализующая все выделенные варианты использования.

На эту фазу уходит около 50% времени и 65% трудоемкости одного цикла. Пример хода работ на этой фазе представлен на Рис. 9.

4. **Фаза внедрения (Transition)**

Цель этой фазы — сделать систему полностью доступной конечным пользователям. На этой стадии происходит развертывание системы в ее рабочей среде, бета-тестирование, подгонка мелких деталей под нужды пользователей.

На эту фазу может уходить около 10% времени и 10% трудоемкости одного цикла. Пример хода работ представлен на Рис. 10.

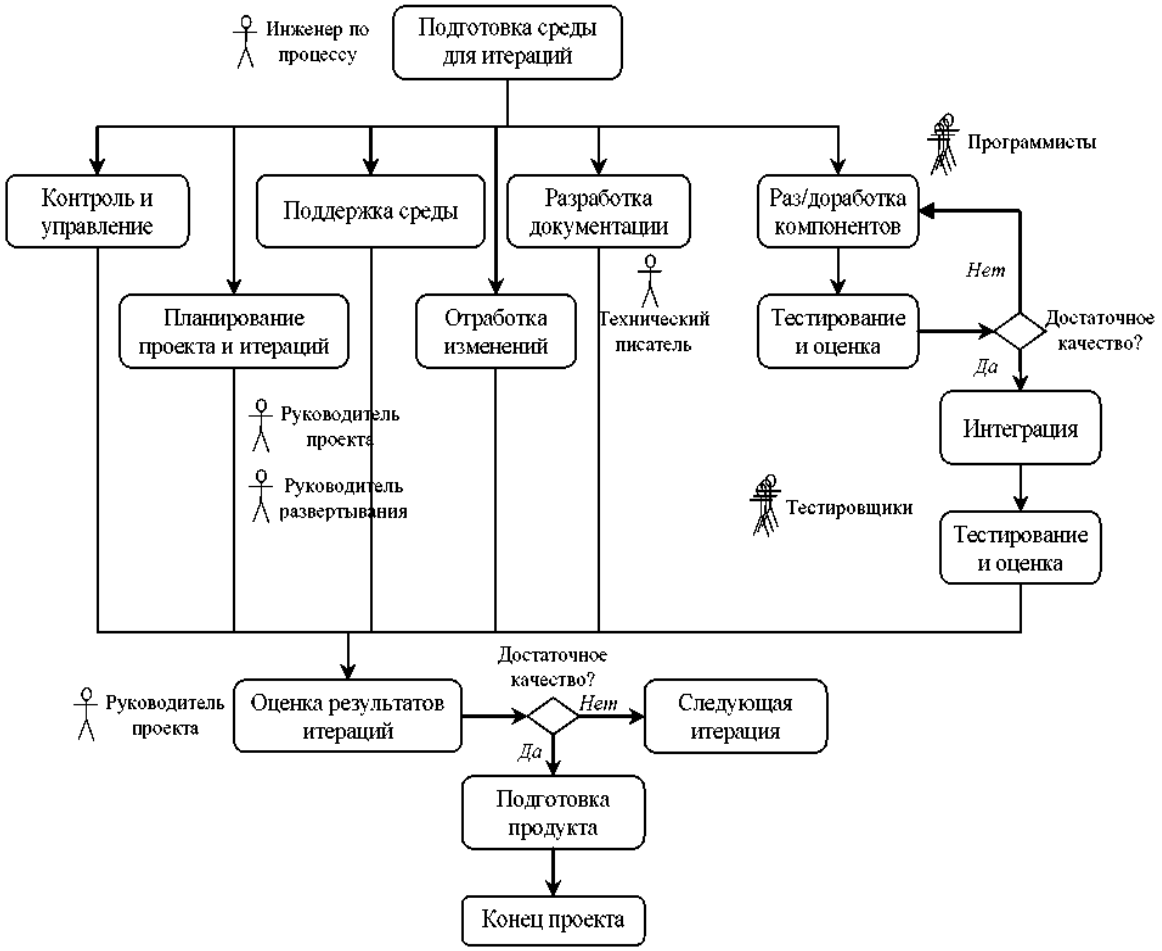


Рисунок 10. Пример хода работ на фазе внедрения.

Артефакты, вырабатываемые в ходе проекта, могут быть представлены в виде баз данных и таблиц с информацией различного типа, разных видов документов, исходного кода и объектных модулей, а также моделей, состоящих из отдельных элементов. Основные артефакты и потоки данных между ними согласно RUP изображены на Рис. 11.

Наиболее важные с точки зрения RUP артефакты проекта — это модели, описывающие различные аспекты будущей системы. Большинство моделей представляют собой наборы диаграмм UML. Основные используемые виды моделей следующие.

- **Модель вариантов использования (Use-Case Model).**

Эта модель определяет *требования к ПО* — то, что система должна делать — в виде набора вариантов использования. Каждый вариант использования задает сценарий взаимодействия системы с ***действующими лицами (actors)*** или ролями, дающий в итоге значимый для них результат. Действующими лицами могут быть не только люди, но и другие системы, взаимодействующие с рассматриваемой. Вариант использования определяет основной ход событий, развивающийся в нормальной ситуации, а также может включать несколько альтернативных сценариев, которые начинают работать только при специфических условиях.

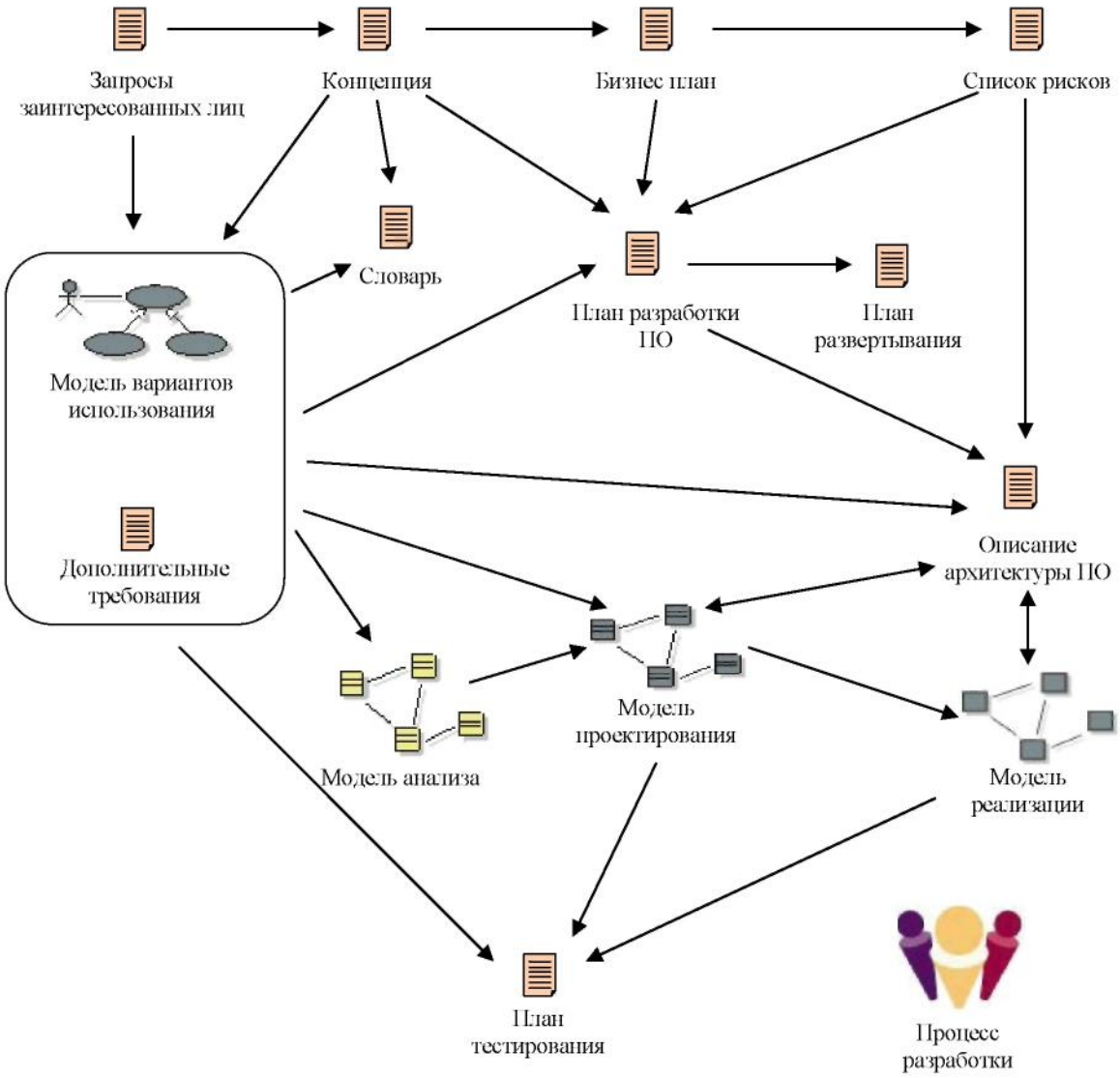


Рисунок 11. Основные артефакты проекта по RUP и потоки данных между ними.

Модель вариантов использования служит основой для проектирования и оценки готовност системы к внедрению.

Примером варианта использования может служить сценарий действий клиента Интернет-магазина по отношению к сайту этого магазина, в результате которых клиент заказывает товар, например, книги. Такой вариант использования можно назвать «Заказ товара». Если нас интересует сайт магазина только как программная система, результатом можно считать то, что запись о сделанном заказе занесена в базу данных, а оператору заказов отправлено электронное письмо, содержащее всю информацию, которая необходима для того, чтобы можно было сформировать заказ. В нее входит контактная информация покупателя, идентификатор заказа и, например, список заказанных книг с их ISBN, их количество для каждого наименования и номера партий для удобства их поиска на складе. При этом выполнение остальной части варианта использования — это дело других составляющих системы под названием «Интернет-магазин». Эта работа может включать звонок или письмо клиенту и подтверждение, что именно он сделал заказ, вопрос об удобных для него форме, времени и адресе доставки и форме опла-

ты, формирование заказа, передача его для доставки курьеру, доставка и подтверждение получения заказа и оплаты. В нашем примере действующими лицами являются клиент, делающий заказ, и оператор заказов.

Альтернативные сценарии в рамках данного варианта могут выполняться, если, например, заказанного пользователем товара нет на складе или сам пользователь находится на плохом счету в магазине из-за неоплаченных прежних заказов, или, наоборот, он является привилегированным клиентом или представителем крупной организации.

Рисунок 12. Пример варианта использования и действующих лиц.

- **Модель анализа (Analysis Model).**

Она включает основные классы, необходимые для реализации выделенных вариантов использования, а также возможные связи между классами. Выделяемые классы разбиваются на три разновидности — *интерфейсные*, *управляющие* и *классы данных*. Эти классы представляют собой набор сущностей, в терминах которых работа системы должна представляться пользователям. Они являются понятиями, с помощью которых достаточно удобно объяснять себе и другим происходящее внутри системы, не слишком вдаваясь в детали.

Интерфейсные классы (boundary classes) соответствуют устройствам или способам обмена данными между системой и ее окружением, в том числе пользователями. ***Классы данных (entity classes)*** соответствуют наборам данных, описывающих некоторые однотипные сущности внутри системы. Эти сущности являются абстракциями представлений пользователей о данных, с которыми работает система. ***Управляющие классы (control classes)*** соответствуют алгоритмам, реализующим какие-то значимые преобразования данных в системе и управляющим обменом данными с ее окружением в рамках вариантов использования.

В нашем примере с Интернет-магазином можно было бы выделить следующие классы в модели анализа: интерфейсный класс, предоставляющий информацию о товаре и возможность сделать заказ; интерфейсный класс, представляющий сообщение оператору; управляющий класс, обрабатывающий введенную пользователем информацию и преобразующий ее в данные о заказе и сообщение оператору; класс данных о заказе. Соответствующая модель приведена на Рис. 13.

Каждый класс может играть несколько ролей в реализации одного или нескольких вариантов использования. Каждая роль определяет его обязанности и свойства, тоже являющиеся частью модели анализа.

В рамках других подходов модель анализа часто называется ***концептуальной моделью*** системы. Она состоит из набора классов, совместно реализующих

все варианты использования и служащих основой для понимания работы системы и объяснения ее правил всем заинтересованным лицам.

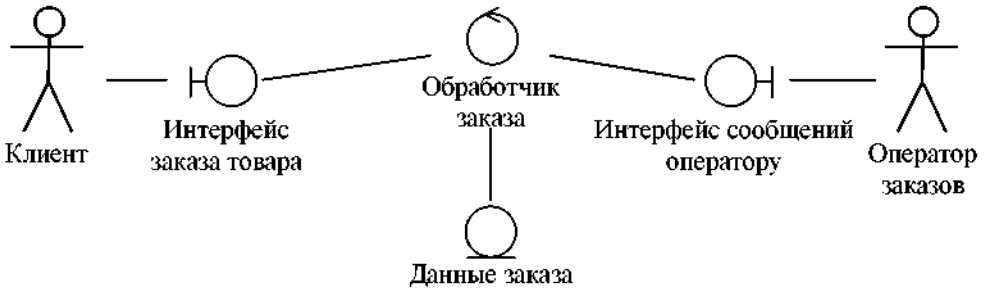


Рисунок 13. Пример модели анализа для одного варианта использования.

• **Модель проектирования (Design Model).**

Модель проектирования является детализацией и специализацией модели анализа. Она также состоит из классов, но более четко определенных, с более точным и детальным распределением обязанностей, чем классы модели анализа. Классы модели проектирования должны быть специализированы для конкретной используемой платформы. Каждая такая платформа может включать операционные системы всех вовлеченных машин; используемые языки программирования; интерфейсы и классы конкретных компонентных сред, таких как J2EE, .NET, COM или CORBA; интерфейсы выбранных для использования систем управления базами данных, СУБД, например, Oracle или MS SQL Server; используемые библиотеки разработки пользовательского интерфейса, такие как swing или swt в Java, MFC или gtk; интерфейсы взаимодействующих систем и пр.

В нашем примере, прежде всего, необходимо детализировать классы, уточнить их функциональность. Скажем, для того, чтобы клиенту было удобнее делать заказ, нужно предоставить ему список имеющихся товаров, какие-то способы навигации и поиска в этом списке, а также детальную информацию о товаре. Это значит, что интерфейс заказа товара реализуется в виде набора классов, представляющих, например, различные страницы сайта магазина. Точно так же данные заказа должны быть детализированы в виде нескольких таблиц в СУБД, включающих, как правило, данные самого заказа (дату, ссылку на данные клиента, строки с количеством отдельных товаров и ссылками на товары), данные товаров, клиента и пр. Кроме того, для реляционной СУБД понадобятся классы-посредники между ее таблицами и объектной структурой остальной программы. Обработчик заказа может быть реализован в виде набора объектов нескольких классов, например, с выделенным отдельно набором часто изменяемых политик (скидки на определенные категории товаров и определенным категориям клиентов, сезонные скидки, рекламные комплекты и пр.) и более постоянным общим алгоритм обработки.

Далее, приняв, например, решение реализовывать систему с помощью технологий J2EE или .NET, мы тем самым определяем дополнительные ограничения на структуру классов, да и на само их количество. О правилах построения ПО на основе этих технологий рассказывается в следующих лекциях.

- **Модель реализации (Implementation Model)**.

Под *моделью реализации* в рамках RUP и UML понимают набор *компонентов* результирующей системы и связей между ними. Под компонентом здесь имеется в виду ***компонент сборки*** — минимальный по размерам кусок кода системы, который может участвовать или не участвовать в определенной ее конфигурации, единица сборки и конфигурационного управления. Связи между компонентами представляют собой зависимости между ними. Если компонент зависит от другого компонента, он не может быть поставлен отдельно от него.

Часто компоненты представляют собой отдельные файлы с исходным кодом. Далее мы познакомимся с компонентами J2EE, состоящими из нескольких файлов.

- **Модель развертывания (Deployment Model)**.

Модель развертывания представляет собой набор ***узлов*** системы, являющихся физически отдельными устройствами, которые способны обрабатывать информацию — серверами, рабочими станциями, принтерами, контроллерами датчиков и пр., со *связями* между ними, образованными различного рода сетевыми соединениями. Каждый узел может быть нагружен некоторым множеством компонентов, определенных в модели реализации. Цель построения модели развертывания — определить физическое положение компонентов распределенной системы, обеспечивающее выполнение ею нужных функций в тех местах, где эти функции будут доступны и удобны для пользователей. В нашем примере Web-сайта магазина узлами системы являются один или несколько компьютеров, на которых развернуты Web-сервер, пересылающий по запросу пользователя текст нужной странички, набор программных компонентов, отвечающих за генерацию страничек, обработку действий пользователя и взаимодействие с базой данных, и СУБД, в рамках которой работает база данных системы. Кроме того, строго говоря, в систему входят все компьютеры клиентов, на которых работает Web-браузер, делающий возможным просмотр страничек сайта и пересылку кодированных действий пользователя для их обработки.

- **Модель тестирования (Test Model** или **Test Suite)**.

В рамках этой модели определяются ***тестовые варианты*** или ***тестовые примеры (test cases)*** и *тестовые процедуры (test scripts)*. Первые являются определенными сценариями работы одного или нескольких действующих лиц с системой, разворачивающимися в рамках одного из вариантов использования. Тестовый вариант включает, помимо входных данных на каждом шаге, где они могут быть введены, условия выполнения отдельных шагов и корректные ответы системы для всякого шага, на котором ответ системы можно наблюдать. В отличие от вариантов использования, в тестовых вариантах четко определены входные данные, и, соответственно, тестовый вариант либо вообще не имеет альтернативных сценариев, либо предусматривает альтернативный порядок действий в том случае, если система может вести себя недетерминировано и выдавать разные результаты в ответ на одни и те же действия. Все другие альтернативы обычно заканчиваются вынесением вердикта о некорректной работе системы.

Тестовая процедура представляет собой способ выполнения одного или нескольких тестовых вариантов и их составных элементов (отдельных шагов и про-

верок). Это может быть инструкция по ручному выполнению входящих в тестовый вариант действий или программный компонент, автоматизирующий запуск тестов.

Для выделенного варианта использования «Заказ товара» можно определить следующие тестовые варианты:

- заказать один из имеющихся на складе товаров и проверить, что сообщение об этом заказе поступило оператору;
- заказать большое количество товаров и проверить, что все работает так же;
- заказать отсутствующий на складе товар и проверить, что в ответ приходит сообщение о его отсутствии;
- сделать заказ от имени пользователя, помещенного в «черный список», и проверить, что в ответ приходит сообщение о неоплаченных прежних заказах.

RUP также определяет *дисциплины,* включающие различные наборы деятельностей, которые в разных комбинациях и с разной интенсивностью выполняются на разных фазах. В документации по процессу каждая дисциплина сопровождается довольно большой диаграммой, поясняющей действия, которые нужно выполнить в ходе работ в рамках данной дисциплины, артефакты, с которыми надо иметь дело, и роли вовлеченных в эти действия лиц.

- **Моделирование предметной области (бизнес-моделирование, Business Modeling)**

Задачи этой деятельности — понять предметную область или бизнес-контекст, в которых должна будет работать система, и убедиться, что все заинтересованные лица понимают его одинаково, осознать имеющиеся проблемы, оценить их возможные решения и их последствия для бизнеса организации, в которой будет работать система.

В результате моделирования предметной области должна появиться ее модель в виде набора диаграмм классов (объектов предметной области) и деятельностей (представляющих бизнес-операции и бизнес-процессы). Эта модель служит основой модели анализа.

- **Определение требований (Requirements)**

Задачи — понять, что должна делать система, и убедиться во взаимопонимании по этому поводу между заинтересованными лицами, определить границы системы и основу для планирования проекта и оценок затрат ресурсов в нем. Требования принято фиксировать в виде модели вариантов использования.

- **Анализ и проектирование (Analysis and Design)**

Задачи — выработать архитектуру системы на основе требований, убедиться, что данная архитектура может быть основой работающей системы в контексте ее будущего использования.

В результате проектирования должна появиться модель проектирования, включающая диаграммы классов системы, диаграммы ее компонентов, диаграммы взаимодействий между объектами в ходе реализации вариантов использования, диаграммы состояний для отдельных объектов и диаграммы развертывания.

- **Реализация (Implementation)**

Задачи — определить структуру исходного кода системы, разработать код ее компонентов и протестировать их, интегрировать систему в работающее целое.

- **Тестирование (Test)**

Задачи — найти и описать дефекты системы (проявления недостатков ее качества), оценить ее качество в целом, оценить выполнены или нет гипотезы, лежащие в основе проектирования, оценить степень соответствия системы требованиям.

- **Развертывание (Deployment)**

Задачи — установить систему в ее рабочем окружении и оценить ее работоспособность на том месте, где она должна будет работать.

- **Управление конфигурациями и изменениями (Configuration and Change Management)** Задачи — определение элементов, подлежащих хранению в репозитории проекта и правил построения из них согласованных конфигураций, поддержание целостности текущего состояния системы, проверка согласованности вносимых изменений.

- **Управление проектом (Project Management)**

Задачи — планирование, управление персоналом, обеспечение взаимодействия на благо проекта между всеми заинтересованными лицами, управление рисками, отслеживание текущего состояния проекта.

- **Управление средой проекта (Environment)**

Задачи — подстройка процесса под конкретный проект, выбор и замена технологий и инструментов, используемых в проекте.

Первые пять дисциплин считаются рабочими, остальные — поддерживающими. Распределение объемов работ по дисциплинам в ходе проекта выглядит, согласно руководству по RUP, примерно так, как показано на Рис. 14.

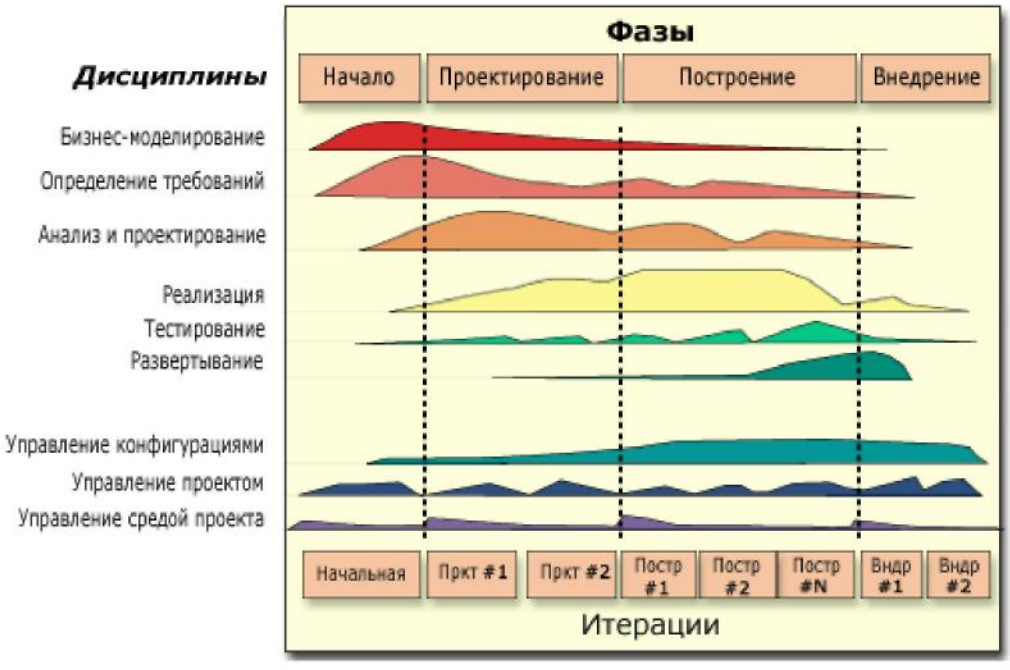


Рисунок 14. Распределение работ между различными дисциплинами в проекте по RUP.

Напоследок перечислим техники, используемые в RUP согласно [3].

- Выработка концепции проекта (project vision) в его начале для четкой постановки задач.
- Управление по плану.
- Снижение рисков и отслеживание их последствий, как можно более раннее начало работ по преодолению рисков.
- Тщательное экономическое обоснование всех действий — делается только то, что нужно заказчику и не приводит к невыгодности проекта.
- Как можно более раннее формирование базовой архитектуры.
- Использование компонентной архитектуры.
- Прототипирование, инкрементная разработка и тестирование.
- Регулярные оценки текущего состояния.
- Управление изменениями, постоянная отработка изменений извне проекта.
- Нацеленность на создание продукта, работоспособного в реальном окружении.
- Нацеленность на качество.
- Адаптация процесса под нужды проекта.

10 Экстремальное программирование.

Экстремальное программирование (Extreme Programming, XP) [4] возникло как эволюционный метод разработки ПО «снизу-вверх». Этот подход является примером так называемого метода *«живой» разработки (Agile Development Method)*. В группу «живых» методов входят, помимо экстремального программирования, методы SCRUM, DSDM (Dynamic Systems Development Method, метод разработки динамичных систем), Feature-Driven Development (разработка, управляемая функциями системы) и др.

Основные принципы «живой» разработки ПО зафиксированы в манифесте «живой» разработки [5], появившемся в 2000 году.

- Люди, участвующие в проекте, и их общение более важны, чем процессы и инструменты.
- Работающая программа более важна, чем исчерпывающая документация.
- Сотрудничество с заказчиком более важно, чем обсуждение деталей контракта.
- Отработка изменений более важна, чем следование планам.

«Живые» методы появились как протест против чрезмерной бюрократизации разработки ПО, обилия побочных, не являющихся необходимыми для получения конечного результата документов, которые приходится оформлять при проведении проекта в соответствии с большинством «тяжелых» процессов, дополнительной работы по поддержке фиксированного процесса организации, как это требуется в рамках, например, CMM. Большая часть таких работ и документов не имеет прямого отношения к разработке ПО и обеспечению его качества, а предназначена для соблюдения формальных пунктов контрактов на разработку, получения и подтверждения сертификатов на соответствие различным стандартам.

«Живые» методы позволяют большую часть усилий разработчиков сосредоточить собственно на задачах разработки и удовлетворения реальных потребностей пользователей. Отсутствие кипы документов и необходимости поддерживать их в связном состоянии позволяет более быстро и качественно реагировать на изменения в требованиях и в окружении, в котором придется работать будущей программе.

Тем не менее, XP имеет свою схему процесса разработки (хотя, вообще говоря, широко используемое понимание «процесса разработки» как достаточно жесткой схемы действий противоречит самой идее «живости» разработки), приведенную на Рис. 15.

По утверждению авторов XP, эта методика представляет собой не столько следование каким-то общим схемам действий, сколько применение комбинации следующих техник. При этом каждая техника важна, и без ее использования разработка считается идущей не по XP, согласно утверждению Кента Бека (Kent Beck) [4], одного из авторов этого подхода, наряду с Уордом Каннингемом (Ward Cunningham), и Роном Джефрисом (Ron Jeffries).

- **Живое планирование (planning game)**

Его задача — как можно быстрее определить объем работ, которые нужно сделать до следующей версии ПО. Решение принимается, в первую очередь, на

основе приоритетов заказчика (т.е. его потребностей, того, что нужно ему от системы для более успешного ведения своего бизнеса) и, во вторую, на основе технических оценок (т.е. оценок трудоемкости разработки, совместимости с остальными элементами системы и пр.). Планы изменяются, как только они начинают расходиться с действительностью или пожеланиями заказчика.

Рисунок 15. Схема потока работ в XP.

- **Частая смена версий (small releases)** Самая первая работающая версия должна появиться как можно быстрее и тут же должна начать использоваться. Следующие версии подготавливаются через достаточно короткие промежутки времени (от нескольких часов при небольших изменениях в небольшой программе, до месяца-двух при серьезной переработке крупной системы).
- **Метафора (metaphor) системы**

Метафора в достаточно простом и понятном команде виде должна описывать основной механизм работы системы. Это понятие напоминает архитектуру, но должно гораздо проще, всего в виде одной-двух фраз описывать основную суть принятых технических решений.

- **Простые проектные решения (simple design)**

В каждый момент времени система должна быть сконструирована настолько просто, насколько это возможно. Не надо добавлять функции заранее — только после явной просьбы об этом. Вся лишняя сложность удаляется, как только обнаруживается.

- **Разработка на основе тестирования (test-driven development)**

Разработчики сначала пишут тесты, потом пытаются реализовать свои модули так, чтобы тесты срабатывали. Заказчики заранее пишут тесты, демонстрирующие основные возможности системы, чтобы можно было увидеть, что система действительно заработала.

- **Постоянная переработка (refactoring)**

Программисты постоянно перерабатывают систему для устранения излишней сложности, увеличения понятности кода, повышения его гибкости, но без изменений в его поведении, что проверяется прогоном после каждой переделки те-

стов. При этом предпочтение отдается более элегантным и гибким решениям, по сравнению с просто дающими нужный результат. Неудачно переработанные компоненты должны выявляться при выполнении тестов и откатываться к последнему целостному состоянию (вместе с зависимыми от них компонентами).

- **Программирование парами (pair programming)**

Кодирование выполняется двумя программистами на одном компьютере. Объединение в пары произвольно и меняется от задачи к задаче. Тот, в чьих руках клавиатура, пытается наилучшим способом решить текущую задачу. Второй программист анализирует работу первого и дает советы, обдумывает последствия тех или иных решений, новые тесты, менее прямые, но более гибкие решения.

- **Коллективное владение кодом (collective ownership)**

В любой момент любой член команды может изменить любую часть кода. Никто не должен выделять свою собственную область ответственности, вся команда в целом отвечает за весь код.

- **Постоянная интеграция (continuous integration)**

Система собирается и проходит интеграционное тестирование как можно чаще, по несколько раз в день, каждый раз, когда пара программистов оканчивает реализацию очередной функции.

- **40-часовая рабочая неделя**

Сверхурочная работа рассматривается как признак больших проблем в проекте. Не допускается сверхурочная работа 2 недели подряд — это истощает программистов и делает их работу значительно менее продуктивной.

- **Включение заказчика в команду (on-site customer)**

В составе команды разработчиков постоянно находится представитель заказчика, который доступен в течение всего рабочего дня и способен отвечать на все вопросы о системе. Его обязанностью являются достаточно оперативные ответы на вопросы любого типа, касающиеся функций системы, ее интерфейса, требуемой производительности, правильной работы системы в сложных ситуациях, необходимости поддерживать связь с другими приложениями и пр.

- **Использование кода как средства коммуникации**

Код рассматривается как важнейшее средство общения внутри команды. Ясность кода — один из основных приоритетов. Следование стандартам кодирования, обеспечивающим такую ясность, обязательно. Такие стандарты, помимо ясности кода, должны обеспечивать минимальность выражений (запрет на дублирование кода и информации) и должны быть приняты всеми членами команды.

- **Открытое рабочее пространство (open workspace)**

Команда размещается в одном, достаточно просторном помещении, для упрощения коммуникации и возможности проведения коллективных обсуждений при планировании и принятии важных технических решений.

- **Изменение правил по необходимости (just rules)**

Каждый член команды должен принять перечисленные правила, но при возникновении необходимости команда может поменять их, если все ее члены пришли к согласию по поводу этого изменения.

Как видно из применяемых техник, XP рассчитано на использование в рамках небольших команд (не более 10 программистов), что подчеркивается и авто-

рами этой методики. Больший размер команды разрушает необходимую для успеха простоту коммуникации и делает невозможным применение многих перечисленных приемов.

Достоинствами XP, если его удается применить, является большая гибкость, возможность быстро и аккуратно вносить изменения в ПО в ответ на изменения требований и отдельные пожелания заказчиков, высокое качество получающегося в результате кода и отсутствие необходимости убеждать заказчиков в том, что результат соответствует их ожиданиям.

Недостатками этого подхода являются невыполнимость в таком стиле достаточно больших и сложных проектов, невозможность планировать сроки и трудоемкость проекта на достаточно долгую перспективу и четко предсказать результаты длительного проекта в терминах соотношения качества результата и затрат времени и ресурсов. Также можно отметить неприспособленность XP для тех случаев, в которых возможные решения не находятся сразу на основе ранее полученного опыта, а требуют проведения предварительных исследований

XP как совокупность описанных техник впервые было использовано в ходе работы на проектом C3 (Chrysler Comprehensive Compensation System, разработка системы учета выплат работникам компании Daimler Chrysler). Из 20-ти участников этого проекта 5 (в том числе упомянутые выше 3 основных автора XP) опубликовали еще во время самого проекта и в дальнейшем 3 книги и огромное количество статей, посвященных XP. Этот проект неоднократно упоминается в различных источниках как пример использования этой методики [6,7,8]. Приведенные ниже данные собраны на основе упомянутых статей [9], за вычетом не подтверждающихся сведений, и иллюстрируют проблемы некоторых техник XP при их применении в достаточно сложных проектах.

Проект стартовал в январе 1995 года. С марта 1996 года, после включения в него Кента Бека, он проходил с использованием XP. К этому времени он уже вышел за рамки бюджета и планов поэтапной реализации функций. Команда разработчиков была сокращена, и в течение примерно полугода после этого проект развивался довольно успешно. В августе 1998 года появился прототип, который мог обслуживать около 10000 служащих. Первоначально предполагалось, что проект завершится в середине 1999 года и результирующее ПО будет использоваться для управления выплатами 87000 служащим компании. Он был остановлен в феврале 2000 года после 4-х лет работы по XP в связи с полным несоблюдением временных рамок и бюджета. Созданное ПО ни разу не использовалось для работы с данными о более чем 10000 служащих, хотя было показано, что оно справится с данными 30000 работников компании. Человек, игравший роль включенного в команду заказчика в проекте, уволился через несколько месяцев такой работы, не выдержав нагрузки, и так и не получил адекватной замены до конца проекта.

Windows Presentation Foundation (WPF) — рекомендуемая технология реализации пользовательских интерфейсов для Windows-приложений. Она позволяет создавать такие функционально насыщенные и визуально привлекательные приложения, о которых вы раньше не могли и мечтать. WPF дает возможность естественно объединять в одной программе традиционные интерфейсы, трехмерную графику, аудио и видео, анимацию, динамическую смену обложек, мультисенсорный ввод, форматированные документы и распознавание речи.

Базовые технологии большинства интерфейсов в Windows — интерфейс графического устройства (Graphics Device Interface, GDI) и подсистема USER - появились в Windows 1.0 еще в 1985 году. В начале 1990-х годов компания Silicon Graphics разработала ставшую популярной графическую библиотеку OpenGL для двумерной и трехмерной графики как в Windows, так и в других системах. Она была с восторгом принята компаниями, работающими в сфере создания систем автоматизированного проектирования, программ визуализации научных данных и игр. Технология Microsoft DirectX, представленная в 1995 году обеспечила высокоскоростную альтернативу для 2D-графики, ввода, сетевого взаимодействия, работы со звуком, а со временем и 3D-графики (которая стала возможной с версией DirectX 2, вышедшей в 1996 году). В последствии и в GDI, и в DirectX было внесено много существенных улучшений.

После появления каркаса .NET и управляемого кода (в 2002 году) разработчики получили очень продуктивную модель для создания Windows и веб-приложений. Включѐнная в неѐ технология Windows Forms (основанная на GDI+) стала основным способом создания пользовательских интерфейсов в Windows для разработчиков на C#, Visual Basic и (в меньшей степени) C++. Она пользовалась успехом и оказалась весьма продуктивной, но имела фундаментальные ограничения, уходящие корнями в GDI+ и подсистему USER.

Корпорация Microsoft понимала, что необходимо нечто совершенно новое свободное от ограничений GDI+ и подсистемы USER, но не менее продуктивное и удобное в использовании, чем Windows Forms.

Перечислим основные возможности, которые предоставляет WPF.

- Широкая интеграция. До WPF разработчикам в Windows, которые хотели использовать одновременно 3D-графику, видео, речь и форматированные документы в дополнение к обычной двумерной графике и элемент управления, приходилось изучать несколько независимых технологий, плохо совместимых между собой и не имеющих встроенных средств сопряжения. А в WPF все это входит в состав внутренне согласованной модели программирования, поддерживающей композицию и визуализацию разнородных элементов.
- Независимость от разрешающей способности. Представьте себе пользовательский интерфейс, который прекрасно выглядит и на маленьком

нетбуке и на полутораметровом экране телевизора! WPF все это обеспечивает и дает возможность уменьшать или увеличивать элементы на экране независимо от его разрешения. Это стало возможным благодаря тому, что WPF основана на использовании векторной графики.

- Аппаратное ускорение. Поскольку WPF основана на технологии DirectX®, то все содержимое в WPF-приложении, будь то двумерная или трехмерная графика, изображения или текст, преобразуется в трехмерные треугольники, текстуры и другие объекты Direct3D, а потом отрисовываются аппаратной графической подсистемой компьютера. Таким образом, WPF-приложения задействуют все возможности аппаратного ускорения графики, что позволяет добиться более качественного изображения и одновременно повысить производительность (поскольку часть работы перекладывается с центральных процессоров на графические).

Язык XAML

Язык XML активно используется в технологиях .NET для выражения различной функциональности в ясном, декларативном стиле. В этом смысле языку XAML (диалекту XML), появившемуся уже в первой версии WPF в 2006 году, отведена особая роль. Часто его неправильно считают всего лишь средством описания пользовательского интерфейса, чем-то вроде HTML. Основное назначение XAML заключается в том, чтобы предоставить программистам и специалистам в других областях возможность совместной работы.

В контексте WPF графические дизайнеры могут с помощью таких средств разработки, как Expression Blend, создавать элегантные пользовательские интерфейсы независимо от программистов, пишущих код приложения. Такая кооперация дизайнеров и программистов стала возможной не только благодаря общему языку XAML, но и потому, что значительная часть функциональности, обеспечиваемой различными API, сделана доступной для декларативного объявления. Например, для описания сложных анимаций переходов состояний не требуется писать процедурный код.

- Язык XAML может стать очень компактным средством описания пользовательского интерфейса и других иерархий объектов.
- XAML позволяет легко отделить внешний вид приложения от его внутренней логики, что сильно упрощает последующее сопровождение, даже если команда разработчиков состоит всего из одного человека.
- Код на XAML легко скопировать в различные средства разработки, например Visual Studio, Expression Blend или какую-нибудь небольшую автономную программу, и сразу увидеть результат без какой либо компиляции.

- Именно код XAML генерируют практически все средства разработки, связанные с WPF.

XAML базируется на языке расширенной разметки *XML (Extensible Markup Language)* и его синтакс определяется следующими правилами:

- каждый элемент XAML-документа отображается на некоторый экземпляр класса .NET. Имя такого элемента в точности соответствует имени класса. Например, элемент < *Button* > служит для WPF инструкцией для построения объекта класса Button;
- элементы XAML можно вкладывать друг в друга. Вложение элементов разметки обычно отображает вложенность элементов интерфейса;
- свойства класса определяются с помощью атрибутов или с помощью вложенных дескрипторов со специальным синтаксисом.

Язык XAML характеризуется самоописанием. Каждый элемент в XAML-документе представляет имя типа (такие как *Button*, *Window* или *Page*) в рамках заданного пространства имен. Атрибуты элементов используются для задания свойств (*Name*, *Height*, *Width* и т.п.) и событий (*Click*, *Load* и т.д.) соответствующих объектов.

Пространство имен в XAML-документе задается с помощью атрибута **xmlns**. В приведенном выше документе объявлено два базовых пространство имен:

- ***xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"*** *– это базовое пространство имен WPF, которое охватывает все классы WPF, включая элементы* управления, которые применяются при построении пользовательского интерфейса. Так как данное пространство имен объявлено без префикса, то оно распространяется на весь XAML-документ;
- ***xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"*** – пространство имен XAML. Оно включает различные свойства утилит XAML, которые позволяют влиять на то, как XAML-документ следует интерпретировать. Данное пространство имен отображается на префикс *x*. Этот префикс можно помещать перед именем элемента (*x:ИмяЭлемента*).

В XAML-документе WPF-приложения часто возникает необходимость обеспечить доступ к каким-либо другим пространствам имен проекта. В этом случае необходимо определить новый префикс и задать пространство имен. Если в проекте имеется пространство имен *MyFirstWpfProject.Commands*, то подключение его к XAML-документу WPF-приложения будет иметь следующий вид (command – используется в качестве префикса).

```
xmlns:command="clr-namespace: MyFirstWpfProject.Commands"
```

Префикс (command) используется для ссылки на пространство имен в XAML-документе. Лексеме ***clr-namespace*** присваивается название пространства имен .NET в сборке.

Атрибуты и элементы свойств

Простые свойства задаются в XAML-документе в соответствии со следующим синтаксисом:

ИмяСвойства ="значение"

При необходимости задать свойство, которое является полноценным объектом, используются *сложные свойства* в соответствии с синтаксисом "свойство-элемент":

Родитель.ИмяСвойства

Синтаксис элементов свойств можно использовать и для простых значений свойств. В следующем примере для Button с помощью атрибутов устанавливаются два свойства (Content и Background).

```
<Button xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
   Content="OK" Background="White"/>
```

То же самое можно записать иначе, задав упомянутые свойства с помощью элементов:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
   <Button.Content>
      OK
   </Button.Content>
   <Button.Background>
      White
   </Button.Background>
 </Button>
```

Что нужно написать, чтобы свойству Content кнопки был присвоен объект Rectangle. XAML предлагает следующий синтаксис для установки составных свойств — элементы свойств. Выглядит это следующим образом:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
   <Button.Content>
      <Rectangle Height="40" Width="40" Fill="Black"/>
   </Button.Content>
</Button>
```

Элементы свойств всегда имеют вид ИмяТипа.ИмяСвойства и обязательно вложены в объектный элемент вида ИмяТипа. У элементов свойств не может быть собственных атрибутов.

Для программного управления элементами управления, описанными в XAML-документе необходимо для элемента управления задать XAML атрибут *Name*. Так для задания имени элементу *Grid* необходимо записать следующую разметку:

```
<Grid Name="grid">

</Grid>
```

Свойство Content

В большинстве классов WPF имеется свойство (задаваемое с помощью атрибута), значением которого является содержимое данного XML-элемента. Оно называется свойством. В некотором смысле свойство содержимого похоже на свойства по умолчанию.

Так составное содержимое Button, например

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
   <Button.Content>
      <Rectangle Height ="40" Width="40" Fill="Black"/>
   </Button.Content>
 </Button >
```

можно переписать так:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
   <Rectangle Height ="40" Width="40" Fill="Black"/>
 </Button >
```

Специальные символы

При определении интерфейса в XAML мы можем столкнуться с некоторыми ограничениями. В частости, мы не можем использовать специальные символы, такие как знак амперсанда &, кавычки " и угловые скобки < и >. Например, мы хотим, чтобы текст кнопки был следующим: `<"Hello">`. У кнопки есть свойство `Content`, которое задает содержимое кнопки. И можно предположить, что нам надо написать так:

```
1<Button Content="<"Hello">" />
```

Но такой вариант ошибочен и даже не скомпилирутся. В этом случае нам надо использовать специальные коды символов:

Символ	Код
<	&lt;
>	&gt;
&	&amp;
"	&quot;

Например:

```
1<Button Content="&lt;&quot;Hello&quot;&gt;" />
```

Еще одна проблема, с которой мы можем столкнуться в XAML - добавление пробелов. Возьмем, к примеру, следующее определение кнопки:

```
1<Button>
2    Hello          World
3</Button>
```

Здесь свойство Content задается неявно в виде содержимого между тегами `<Button>....</Button>`. Но несмотря на то, что у нас несколько пробелов между словами "Hello" и "World", XAML по умолчанию будет убирать все эти пробелы. И чтобы сохранить пробелы, нам надо использовать атрибут `xml:space="preserve"`:

```
1<Button xml:space="preserve">
2    Hello          World
3</Button>
```

Расширения разметки

Для задания значения свойства из выделенного класса используется ***расширение разметки,*** которое обеспечивает расширение грамматики XAML новой функциональностью. Расширения разметки могут использоваться во вложенных дескрипторах или в XAML-атрибутах. Когда это используется в атрибутах, то необходимо применять фигурные скобки {...}.

Например, для задания атрибуту ***Foreground*** объекта ***Button*** статического свойства, определенного в другом классе, необходимо создать следующее XAML-описание.

```
<Button
Foreground="{x:Static SystemColors.ActiveCaptionBrush}"
  />
```

Присоединенные свойства

Присоединенные свойства описывают свойства, которые могут применяться к нескольким элементам управления, но которые определены в другом классе. В WPF-приложениях присоединенные свойства часто используются для управления компоновкой элементов интерфейса. Синтаксис присоединенных свойств следующий.

```
ОпределяющийТип.ИмяСвойства
```

Например, если необходимо расположить кнопку в нулевой строке сетки, то необходимо сделать следующее XAML-описание.

```
<Button . .. Grid.Row="0" >
....
</Button>
```

Здесь присоединенным свойством является *Grid.Row*, то есть свойство *Row* элемента *Grid*, которое не является свойством объекта *Button*. Свойство *Row* присоединяется к свойствам объекта *Button*, поскольку данный объект располагается в контейнере *Grid*.

Атрибуты объектов *могут использоваться для присоединения обработчиков событий*, используя следующий синтаксис.

```
ИмяСобытия = "ИмяМетодаОбработчикаСобытия"
```

Например, для кнопки событию её нажатия *Click* можно установить обработчик события ***Exit_Click***.

```
<Button Name="Exit" Content="Выход" Click="Exit_Click" />
```

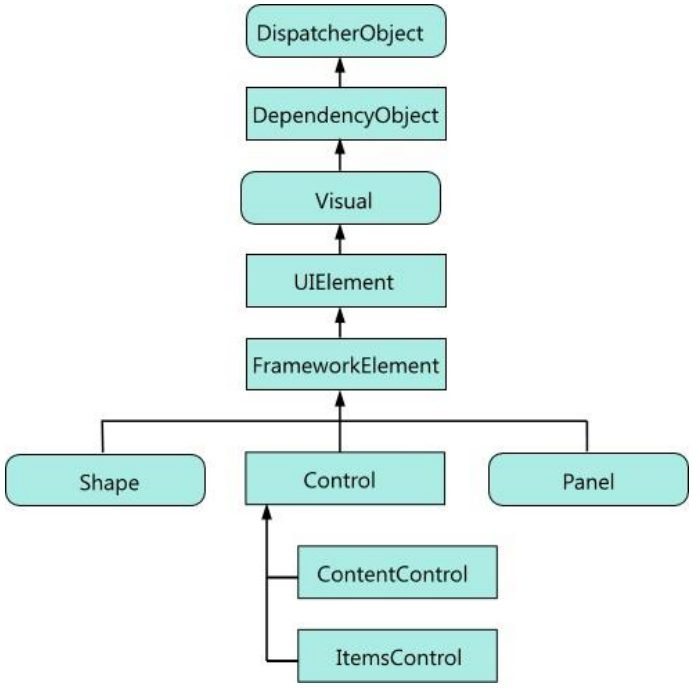
Определяя в XAML-описании обработчик ***Exit_Click***, необходимо в коде класса иметь метод с корректной сигнатурой.

Основные принципы WPF

Обзор иерархии классов Классы, входящие в состав WPF, образуют очень глубокую иерархию наследования, поэтому сразу трудно уложить в голове назначение и взаимосвязи различных классов. Но есть несколько фундаментальных для работы WPF классов, о которых стоит упомянуть, прежде чем двигаться дальше. На рис. 3.1 показаны 12 наиболее важных классов и соотношения между ними.

- Object - базовый класс, которому наследуют все остальные классы .NET, и единственный из представленных на рисунке, не имеющий прямого отношения к WPF.

- DispatcherObject - базовый класс, предназначенный для объектов, к которым можно обращаться только из того потока, где они были созданы. Большинство классов WPF наследуют DispatcherObject и, следовательно, принципиально небезопасны относительно потоков.
- DependencyObject - базовый класс, предназначенный для объектов, поддерживающих свойства зависимости; это одна из центральных тем данной главы.
- Visual - базовый класс для объектов, имеющих двумерное визуальное представление.
- UlElement - базовый класс для двумерных визуальных объектов с поддержкой маршрутизации событий, привязки команд, компоновки и захвата фокуса.
- FrameworkElement - базовый класс, добавляющий поддержку стилей, привязки к данным, ресурсов и нескольких механизмов, общих для всех элементов управления в Windows, в частности всплывающих подсказок и контекстных меню.
- Control - базовый класс для таких хорошо знакомых элементов управления, как Button, ListBox и StatusBar. Класс Control добавляет к своему базовому классу FrameworkElement множество свойств, например Foreground, Background и FontSize, а также возможность тотального изменения стиля.
- Shape является базовым для построения таких геометрических форм как прямоугольник, эллипс, многоугольник, линия и путь.
- ContentControl и ItemsControl являются базовыми для элементов управления, которые могут иметь содержание единственное или коллекцию соответственно.
- Panel является базовым для всех контейнеров компоновки – элементов, которые могут содержать один или большее число дочерних элементов.



Логические и визуальные деревья

Естественность применения языка XAML для описания пользовательских интерфейсов объясняется его иерархической природой. В WPF пользовательский интерфейс представляет собой дерево объектов, которое называется логическим деревом.

Идея логического дерева представляется очевидной, так зачем о нем вообще говорить? Затем, что поведение чуть ли не всех механизмов WPF (свойств, событий, ресурсов и т.д.) так или иначе связано с логическим деревом. Например, значения свойств иногда автоматически распространяются вниз по дереву на дочерние элементы, а генерируемые события могут распространяться как вниз, так и вверх.

Наследование значений свойств

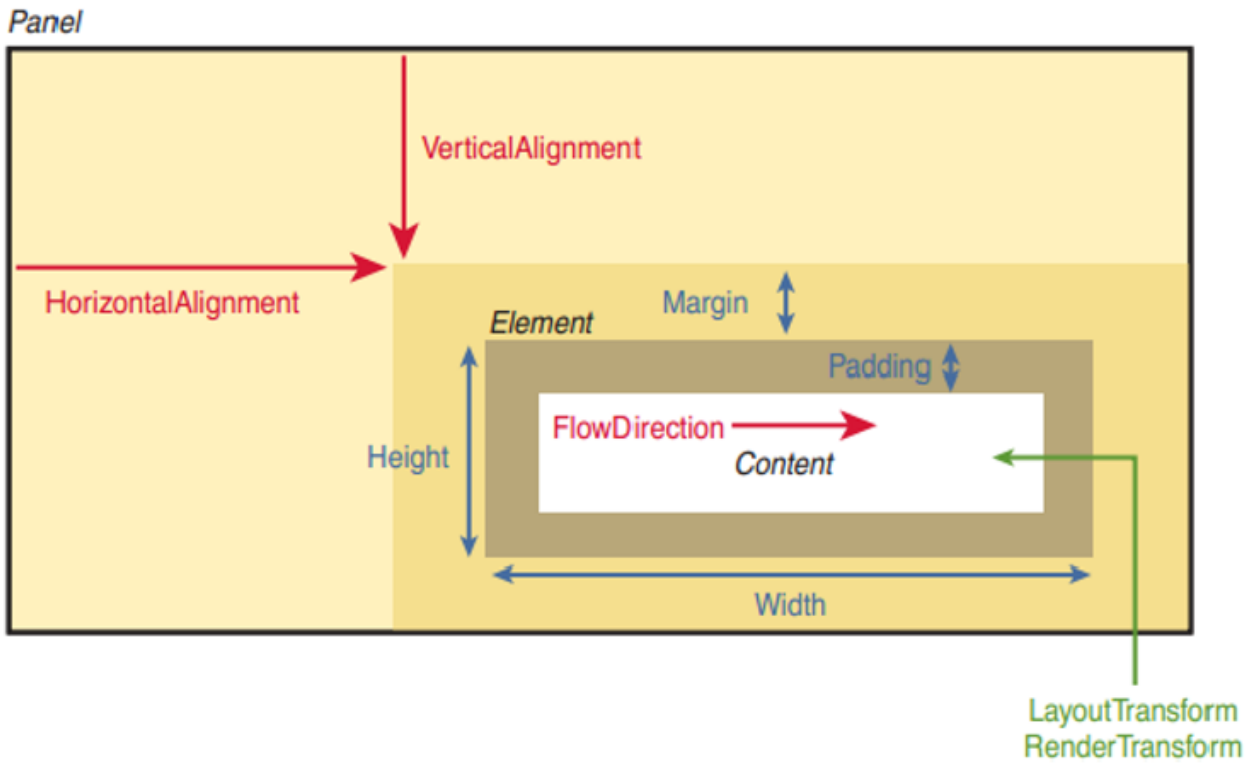
Словосочетание «наследование значений свойств» (или просто «наследование свойств») относится не к традиционному для объектно-ориентированного программирования наследованию классов, а к распространению значений свойств вдоль дерева элементов.

Свой вклад в вычисление базового значения вносят почти все поставщики значений свойств. Ниже приведен перечень десяти поставщиков, которые могут устанавливать значения большинства свойств зависимости, - в порядке убывания приоритета:

1. . Локальное значение
2. . Триггер в шаблоне родителя
3. . Шаблон родителя
4. . Триггеры в стиле
5. . Триггеры в шаблоне
6. . Установщики стиля
7. . Триггеры стиля темы
8. . Установщики стиля темы
9. . Наследование значения свойства
10.. Значение по умолчанию

С некоторыми поставщиками значений свойств вы уже встречались, например, с механизмом наследования свойств (9). Локальное значение (1) технически означает любое обращение к методу DependencyObject.SetValue, но обычно имеет вид простого присваивания свойства в XAML или процедурном. Под значением по умолчанию (10) понимается начальное значение, указанное при регистрации свойства зависимости понятно, что его приоритет наименьший. Остальные поставщики связаны со стилями и шаблонами.

Задание размера, положения и преобразований элементов.



Управление размером

Всякий раз, когда требуется произвести компоновку дочерние элементы сообщают родительской панели свой предпочтительный размер. Обычно элементы WPF стремятся подстроиться под размер своего содержимого, но не больше. Отдельные элементы могут влиять на выбор этого размера с помощью нескольких простых свойств.

Свойства Height и Width

Во всех классах, производных от FrameworkElement, есть свойства Height (высота) и Width (ширина) (типа double), а также MinHeight, MaxHeight, MinWidth и MaxWidth, которыми можно пользоваться для задания допустимых диапазоне» значений. Все эти свойства можно задавать в любой комбинации как в процедурном коде, так и в XAML.

Обычно элемент стремится принять минимально возможный размер, поэтому если задано свойство MinHeight или MinWidth, то при визуализации выбирается именно такая высота или ширина при условии, что содержимое не вынуждает увеличить размер. Но увеличение можно ограничить с помощью свойств MaxHeight и MaxWidth. Если одновременно с минимальными и максимальными значениями заданы свойства Height и Width, то последние имеют приоритет при условии, что попадают внутрь диапазона между Min и Max. По умолчанию MinHeight и MinWidth равны 0, а MaxHeight и MaxWidth - величине Double.PositiveInfinity

Избегайте явного задания размеров! Если явно задавать размеры элементов управления, особенно производных от класса ContentControl, например Button и Label, то возникает риск отсечения текста в случае, когда пользователь изменяет системный шрифт или текст переводится на другие языки.

Если значения свойств Height и Width не заданы явно, то они будут иметь значение Double. NaN, каким бы ни оказался истинный размер элемента.

Свойства Margin и Padding

Очень похожие свойства Margin и Padding тоже связаны с размером элемента. Свойство Margin определено для всех объектов, производных от FrameworkElement, свойство Padding во всех элементах управления, производных от класса Control (а также в классе Border). Различие в том, что Margin задает внешнее поле вокруг элемента, а Padding - внутренний отступ между содержимым элемента и его границами. Оба свойства имеют тип System.Windows.Thickness; это любопытный класс, который может представлять одно, два или четыре значения типа double. Свойство Margin (но не Padding) может принимать отрицательные значения.

Свойство Visibility

Может показаться странным, что мы обсуждаем свойство в контексте компоновки, однако оно действительно относится именно к этой теме. Тип свойства элемента Visibility - не Boolean, а перечисление System.Windows.Visibility с тремя состояниями, то есть оно может принимать три значения:

- Visible – элемент виден и участвует в компоновке.
- Collapsed – элемент не виден и не участвует в компоновке.
- Hidden – элемент не виден, но тем не менее участвует в компоновке.

Свернутый (Collapsed) элемент, по существу, имеет нулевой размер, тогда как скрытый (Hidden) элемент сохраняет свой первоначальный размер. (Так, значения его свойств ActualHeight и ActualWidth не изменяются.).

Выравнивание

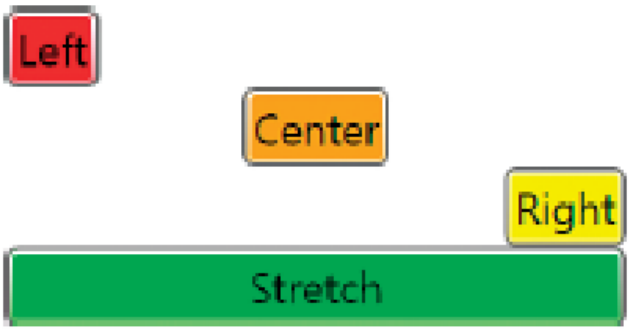
С помощью свойств HorizontalAlignment и VerticalAlignment элемент может управлять распределением избыточного пространства, выделенного ему родителем. Значениями свойств являются одноименные перечисления, которые определены в пространстве имен System.Windows:

- HorizontalAlignment - Left, Center, Right, Stretch
- VerticalAlignment - Top, Center, Bottom, Stretch

По умолчанию оба свойства принимают значение Stretch, хотя в стилях тем для различных элементов управления оно может быть переопределено.

Чтобы посмотреть, как свойство HorizontalAlignment влияет на компоновку, достаточно поместить несколько кнопок Button на панель StackPanel и задаться них разные значения перечисления:

<StackPanel> <Button HorizontalAlignment="Left" Background="Red">Left</Button> <Button HorizontalAlignment="Center" Background="Orange">Center</Button> <Button HorizontalAlignment="Right" Background="Yellow">Right</Button> <Button HorizontalAlignment="Stretch" Background="Lime">Stretch</Button> </StackPanel>



Влияние HorizontalAlignment на размещение кнопок на панели StackPanel.

Выравнивание содержимого

Помимо свойств HorizontalAlignment и VerticalAlignment, в классе Control имеются свойства HorizontalContentAlignment и VerticalContentAlignment. Они определяют порядок размещения содержимого внутри элемента управления.

Свойства, управляющие выравниванием содержимого, принадлежат тем же типам перечисления, что и соответствующие свойства выравнивания, следовательно, и возможности у них точно такие же. Однако по умолчанию свойство HorizontalContentAlignment равно Left, а VerticalContentAlignment равно Top.

Свойство FlowDirection

Свойство FlowDirection, определенное в классе FrameworkElement (и еще нескольких), позволяет изменить направление визуализации внутреннего содержимого элемента. Оно применимо к некоторым панелям, где влияет на размещение дочерних элементов, а также модифицирует способ выравнивания содержимого внутри дочерних элементов. Тип этого свойства - перечисление System.Windows.FlowDirection, принимающее два значения: LeftToRight (по умолчанию в классе FrameworkElement) и RightToLeft.

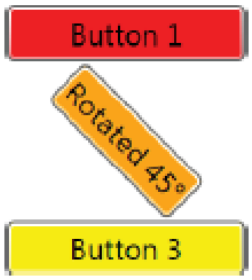
Отметим, что свойство FlowDirection не оказывает влияния на направление записи букв в надписи внутри кнопок. Английские буквы всегда записываются слева направо, арабские - справа налево.

Применение преобразований

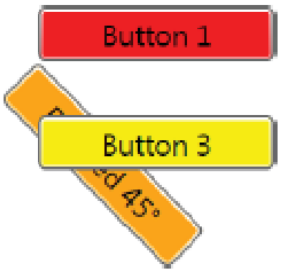
В WPF имеется целый ряд встроенных классов двумерных геометрических преобразований (производных от System.Windows.Media.Transform), которые позволяют изменять размер и положение элементов независимо от ранее рассмотренных свойств. Некоторые преобразования изменяют элементы и более экзотическими способами, например, поворачивают или наклоняют их.

Во всех подклассах FrameworkElement имеется два свойства типа Transform, позволяющих применять преобразования:

- LayoutTransform - применяется до компоновки элемента
- RenderTransform (унаследовано от UIElement) - применяется после завершения компоновки (непосредственно перед визуализацией элемента)



Поворот, примененный в режиме LayoutTransform



Поворот, примененный в режиме RenderTransform

В классе UIElement имеется также полезное свойство RenderTransformOrigin, представляющее начальную точку преобразования (только для RenderTransform). Свойство RenderTransformOrigin имеет тип System.Windows.Point и по умолчание равно (0,0). Этой точке соответствует левый верхний угол элемента, как показано на рис. Точка (0,1) представляет левый нижний угол, (1,0) - правый верхний угол, а (1,1) - правый нижний угол. Можно задавать и числа, большие 1, - тогда начальная точка окажется вне границ элемента. Дробные значения также допустимы.

Преобразование RotateTransform

Преобразование RotateTransform, продемонстрированное в предыдущем разделе, поворачивает элемент в соответствии со следующими тремя свойствами типа double:

- Angle - угол поворота в градусах (по умолчанию 0)
- CenterX - абсцисса центра поворота (по умолчанию 0)
- CenterY - ордината центра поворота (по умолчанию 0)

Точка (CenterX,CenterY), равная по умолчанию (0,0), соответствует левому верхнему углу. Свойства CenterX и CenterY принимаются во внимание, только если преобразование применяется в режиме RenderTransform, потому что для

преобразования в режиме LayoutTransform положение центра поворота определяется родительской панелью.

CenterX и CenterY задают абсолютное положение начальной точки, тогда как RenderTransformOrigin - относительное. Значения задаются в независимых от устройства пикселах, так что для правого верхнего угла элемента с шириной Width, равной 20, свойство CenterX будет равно 20, а CenterY - 0, а не (1,0), как для RenderTransformOrigin.

Преобразование ScaleTransform

Преобразование ScaleTransform увеличивает или уменьшает элемент по горизонтали, по вертикали или в обоих направлениях. У него есть четыре свойства типа double:

- ScaleX - коэффициент изменения ширины элемента (по умолчанию 1)
- ScaleY - коэффициент изменения высоты элемента (по умолчанию 1)
- CenterX-начальная точка для масштабирования по горизонтали (по умолч. 0)
- CenterY - начальная точка для масштабирования по вертикали (по умолч. 0)

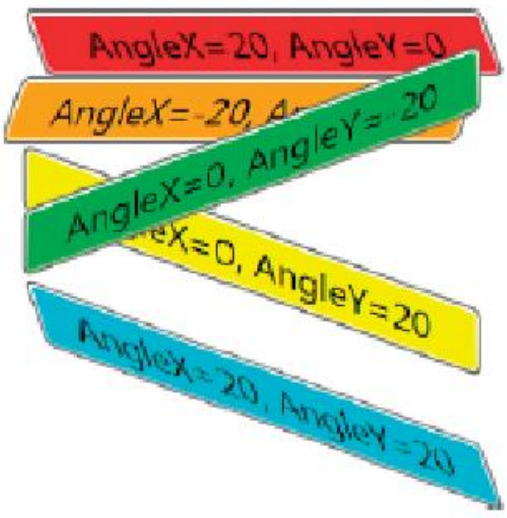
Если ScaleX равно 0.5, то ширина рисуемого элемента уменьшается вдвое, а если ScaleX равно 2, то вдвое увеличивается. Смысл свойств CenterX и CenterY такой же, как для преобразования RotateTransform.

Преобразование ScaleTransform

Преобразование ScaleTransform увеличивает или уменьшает элемент по горизонтали, по вертикали или в обоих направлениях. У него есть четыре свойства типа double:

- ScaleX - коэффициент изменения ширины элемента (по умолчанию 1)
- ScaleY - коэффициент изменения высоты элемента (по умолчанию 1)
- CenterX-начальная точка для масштабирования по горизонтали (по умолч. 0)
- CenterY - начальная точка для масштабирования по вертикали (по умолч. 0)

Если ScaleX равно 0.5, то ширина рисуемого элемента уменьшается вдвое, а если ScaleX равно 2, то вдвое увеличивается. Смысл свойств CenterX и CenterY такой же, как для преобразования RotateTransform.



Преобразование TranslateTransform

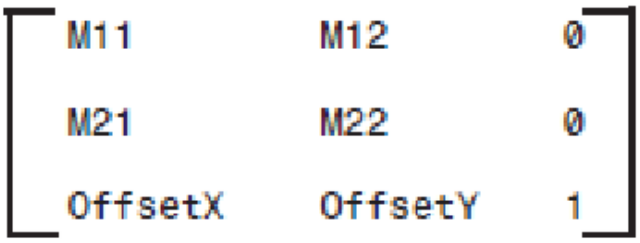
Преобразование TranslateTransform просто параллельно переносит элемент в соответствии со значениями двух свойств типа double:

• X - величина смещения по горизонтали (по умолчанию 0)
• Y- величина смещения по вертикали (по умолчанию 0)

TranslateTransform не дает никакого эффекта, когда применяется в режиме LayoutTransform, но применение его в режиме RenderTransform удобный способ «подвинуть» элементы. Чаще всего это делается динамически в ответ на действия пользователя.

Преобразование MatrixTransform

Преобразование MatrixTransform представляет собой низкоуровневый механизм описания произвольного двумерного преобразования. У него есть единственное свойство Matrix(типа System.Windows.Media.Matrix), представлении матрицу размером 3x3. Все рассмотренные выше преобразования (и их комбинации) также можно осуществить с помощью MatrixTransorm. Матрица устроена следующим образом:



Значения в последнем столбце фиксированы, а остальные шесть значений, но задать в виде свойств объекта Matrix или в конструкторе, который принимает шесть чисел в порядке следования строк матрицы.

Комбинирование преобразований

Комбинировать преобразования можно различными способами. Можно совместно применять преобразования в режимах LayoutTransform и RenderTransform. Или вычислить матрицу преобразования MatrixTransform. Однако, скорее всего, вы предпочтете воспользоваться классом TransformGroup. Его задача - скомбинировать несколько дочерних объектов типа Transform.

В процедурном коде объекты отдельных преобразований добавляются в коллекцию Children, в XAML это делается следующим образом:

```
<Button>
      <Button.RenderTransform>
      <TransformGroup>
            <RotateTransform Angle="45"/>
           <ScaleTransform ScaleX="5" ScaleY="1"/>
            <SkewTransform AngleX="30"/>
       </TransformGroup>
       </Button.RenderTransform>
  OK
</Button>
```

Компоновка с помощью панелей

Компоновка - важнейший фактор, обеспечивающий удобство работы с приложением на различных устройствах. Можно, конечно, размещать элементы пользовательского интерфейса статически, задавая координаты и размеры в пикселах, и где-то это даже будет работать, однако такое решение «посыпется» под воздействием самых разных факторов: разная разрешающая способность, заданные пользователем настройки, к примеру, размер шрифта, содержимое, изменяющееся непредсказуемым образом (например, после перевода текста на другие языки). Добавим еще, что приложение, не позволяющее пользователю изменять размеры своих частей многих раздражает.

Z-порядок по умолчанию (задающий, какие элементы располагаются «поверх» других) определяется порядком добавления дочерних элементов к родителю. В XAML это определяется порядком следования дочерних элементов в файле. Элементы, добавленные позже, располагаются поверх элементов, добавленных раньше.
Однако Z-порядок любого элемента можно задать явно, указав для него присоединенное свойство ZIndex, определенное в классе Panel (и наследуемое всеми панелями). ZIndex- это целое число, по умолчанию равное 0; оно может принимать любое целое значение (положительное или отрицательное). Элементы с большим значением ZIndex рисуются поверх элементов с меньшим значением.
Если для нескольких элементов задано одно и то же значение ZIndex, то их взаимное расположение определяется порядком следования в коллекции Children панели, как в случае по умолчанию.

```
<Canvas>
        <Button Canvas.ZIndex="1" Background="Red">On Top!</Button>
        <Button Background="Orange">On Bottom with a Default ZIndex=0</Button>
</Canvas>
```

В программе на C# это делается следующим образом:

```
Panel.SetZIndex(redButton, 0);
```

Панель Canvas

Canvas(холст) - самая простая панель. Canvas поддерживает только «классическое» позиционирование элементов путем явного задания координат; впрочем, координаты хотя бы задаются в независимых от устройства пикселах, в отличие от прежних систем конструирования пользовательских интерфейсов. Панель Canvas позволяет задавать координаты относительно любого, а не только левого верхнего угла.

Позиционирование элемента на холсте осуществляется с помощью присоединенных свойств: Left, Top, Right и Bottom. Задавая значение Left или Right, вы определяете, что ближайшая сторона элемента должна всегда отстоять на фиксированное расстояние от соответствующей стороны холста. То же самое относится к свойствам Top и Bottom. По сути дела, вы указываете угол, к которому «примыкает» каждый элемент, а значения присоединенных свойств выступают в роли полей (к которым добавляются значения самого свойства Margin элемента). Если для некоторого элемента не задано ни одно присоединенное свойство (то есть все они имеют значение по умолчанию Double.NaN), то он помещается в левый верхний угол (что эквивалентно установке для Left и Top значения 0).

Для элемента нельзя задавать более двух присоединенных свойств Canvas! При попытке одновременно установить свойства Canvas.Left и Canvas.Right последнее будет проигнорировано. А при попытке одновременно установить свойства Canvas.Top и Canvas.Bottom будет проигнорировано Canvas.Bottom. Таким образом, невозможно пристыковать элемент более чем к одному углу холста.

Хотя панель Canvas слишком примитивная для создания гибких пользовательских интерфейсов, она работает быстрее всех других панелей.

Панель StackPanel

Панель StackPanel популярна из-за своей простоты и удобства. Как следует из названия, она последовательно размещает своих потомков в виде стопки.

В отсутствие присоединенных свойств единственный способ организова дочерние элементы - воспользоваться свойством панели Orientation, которое может принимать значение Horizontal или Vertical. По умолчанию подразумевается ориентация Vertical.

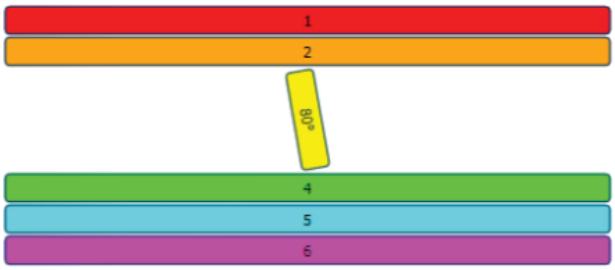
В случае ориентации Vertical элементы располагаются один под другим

В случае ориентации Horizontal элементы располагаются слева направо

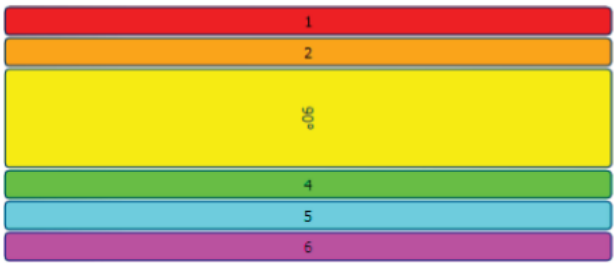
выравнивание HorizontalAlignment и VerticalAlignment игнорируется в направлении сборки стопки (так как дочерним элементам отводится ровно столько места, сколько им необходимо). Если Orientation="Vertical" ,то игнорируется значение VerticalAlignment. Если Orientation="Horizontal", то игнорируется значение HorizontalAlignment.

При комбинировании компоновки Stretch преобразованием RotateTransform или SkewTransform, применяемым в режиме LayoutTransform, растяжение происходит, только если угол кратен 90. Это поведение не является особенностью StackPanel, а присутствует всюду, где элемент растягивается только в одном направлении. Эта странность проявляется, только когда преобразование применяется в режиме LayoutTransform; RenderTransform она не относится.

При повороте на 80° растяжения нет



При повороте на 90° растяжение есть



Панель WrapPanel

Панель WrapPanel похожа на StackPanel. Но помимо организации дочерних элементов в стопку она создает новые строки или столбцы, когда для одной стопки не хватает места. Это полезно для отображения заранее неизвестного числа элементов, когда компоновка должна отличаться от простого списка,- как, например, в Проводнике Windows.

В классе WrapPanel определены три свойства, контролирующие его поведение:

- Orientation- аналогично одноименному свойству StackPanel с тем отличием, что по умолчанию подразумевается значение Horizontal. Панель с горизонтальной ориентацией выглядит, как вид Эскизы страниц в Проводнике Windows: элементы располагаются один за другим слева направо, а когда место кончается, переходят на следующую строку. Панель с вертикальной ориентацией выглядит, как вид Список в Проводнике Windows: элементы располагаются один под другим, а когда место кончается, начинается новый столбец.
- ItemHeight- единая высота для всех дочерних элементов. Каким образом каждый потомок распоряжается этой высотой, зависит от значений его свойств VerticalAlignment, Height и пр. Элементы, ширина которых превышает ItemHeight, отсекаются.
- ItemWidth- единая ширина для всех дочерних элементов. Каким образом каждый потомок распоряжается этой шириной, зависит от значений его свойств HorizontalAlignment, Width и пр. Элементы, высота которых превышает ItemWidth, отсекаются.

По умолчанию свойства ItemHeight и ItemWidth не установлены (точнее, имеют значение Double.NaN).В этом случае панель WrapPanel с вертикальной ориентацией

отводит каждому столбцу ширину, равную ширине самого широкого элемента в нем, а панель с горизонтальной ориентацией отводит каждой строке высоту, равную высоте самого высокого элемента в ней. Поэтому по умолчанию ни в строках, ни в столбцах отсечение не производится.

Можно заставить панель WrapPanel располагать элементы в одну строку или в один столбец. Для этого следует присвоить свойству Width (в случае горизонтальной ориентации) или свойству Height(в случае вертикальной ориентации) значение Double.MaxValue либо Double.PositiveInfinity. В XAML это достигается с помощью расширения разметки x: Static, поскольку ни то ни другое значение не поддерживается конвертером типа System. Double.

Если свойство FlowDirection равно RightToLeft то для панели WrapPanel с вертикальной ориентацией новый столбец создается слева от заполненного, а для панели с горизонтальной ориентацией заполнение строки производится справа налево.

Выравнивание HorizontalAlignment и VerticalAlignment можно задавать в направлении, противоположном направлению роста стопки, как и в случае StackPanel. Но выравнивание может быть полезно и в направлении роста стопки, если значение ItemHeight или ItemWidth таково, что в элементе имеется дополнительное пространство для выравнивания

Панель DockPanel

Панель DockPanel дает простой способ пристыковки элемента к одной из сторон, растягивая его на всю имеющуюся ширину или высоту. (Отличие от Canvas заключается в том, что элементы пристыковываются не к одному углу, а ко всей стороне.) Кроме того, DockPanel позволяет расположить один элемент» так чтобы он занял все место, свободное от пристыкованных элементов.

Порядок добавления кнопок на панель обозначен числом (и цветом).

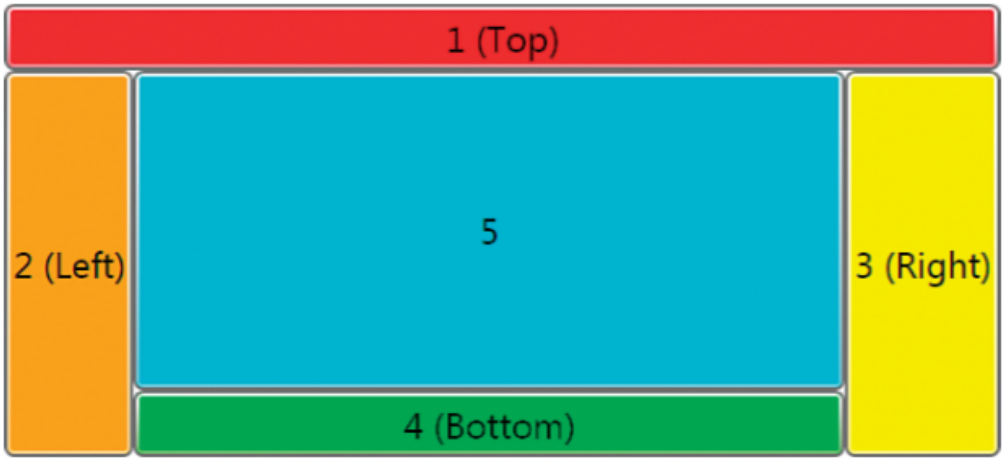


Рис. 5.6. Кнопки на панели DockPanel

В классе DockPanel определено присоединенное свойство Dock(типа System.Windows.Controls.Dock), с помощью которого дочерние элементы могут управлять своим положением. Оно может принимать четыре значения: Left, Top, Right и Bottom.

Отметим, что у свойства Dock нет значения Fill, означающего, что нужно заполнить оставшееся место. Вместо этого действует соглашение о том, что все оставшееся место отдается последнему дочернему элементу, добавленному в DockPanel, если только свойство LastChildFill не равно false. Если LastChildFill равно true(по умолчанию), то значение свойства Dock, заданное для последнего добавленного элемента, игнорируется. Если же оно равно false, то последний элемент можно пристыковать к любой стороне. Если к одной стороне пристыковано несколько элементов, то они просто организуются в стопку соответствующего направления.

Как и в случае StackPanel, растяжение элементов определяется подразумеваемым по умолчанию значением свойства HorizontalAlignment или VerticalAlignment. Если элемент не будет занимать все пространство, выделенное ему панелью DockPanel, то можно задать другое выравнивание. На рис. 5.7 показано, как будет выглядеть панель, когда у всех кнопок, кроме одной, явно задано значение HorizontalAlignment или VerticalAlignment:

Рис. 5.7. Кнопки на панели DockPanel не занимают все выделенное им место

Панель DockPanel полезна для организации верхнего уровня интерфейса внутри элемента Window или Page, когда пристыкованные элементы по большей части представляют собой другие панели, где и находится все самое важное. Так, обычно к верхней стороне пристыковывается меню (Menu), справа и слева находятся какие-то панели, а снизу - строка состояния (StatusBar). Центральную же часть занимают основные данные приложения.

Панель Grid Grid

(сетка) - самая гибкая из всех панелей и, пожалуй, наиболее употребительная. (В проектах VisualStudio и ExpressionBlend панель Grid используется по умолчанию.) Она позволяет расположить дочерние элементы в несколько строк и несколько столбцов, не полагаясь на режим автоматического переноса (как в WrapPanel). Кроме того, предоставляется ряд интересных способов управления строками и столбцами. Работа с панелью Grid очень напоминает использование элемента TABLE в HTML.

Определяем количество строк и столбцов, помещая нужное число элементов RowDefinition и ColumnDefinition внутрь элементов, соответствующих свойствам сетки RowDefinitions и ColumnDefinitions. Затем позиционируем дочерние элементы с помощью присоединенных свойств Row и Column, принимающих целочисленные значения, начиная с 0.

```
<Grid Background="LightBlue">
 <!—Определение четыре строки: -->
 <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition/>
 </Grid.RowDefinitions>
 <!—Определяем два столбца: -->
 <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
 </Grid.ColumnDefinitions>

<Label Grid.Row="0" Grid.Column="0" Background="Blue" Foreground="White" HorizontalContentAlignment="Center">Start Page</Label>
. . .

</Grid>
```

Ячейки сетки можно оставлять пустыми, и, наоборот, в одной ячейке может находиться несколько элементов. В таком случае они просто располагаются один над другим в соответствии с заданным Z-порядком.

Свойства RowSpan и ColumnSpan указывают на количество строк столбцов, занимаемых данной ячейкой. По умолчанию они равны 1. (Если значение больше общего количества строк или столбцов, то ячейка занимает столько строк или столбцов сколько возможно.)

например добавим в элемент Label- атрибут

```
Grid.ColumnSpan=‘2‘;
```

и надпись займет две верхние ячейки и расположится посередине.

Автоматический выбор размера достигается присваивания свойствам Height и Width соответственно в элементах RowDefinition и ColumnDefinition специального значения Auto, нечувствительного к регистру букв.

Задание размеров строк и столбцов

В отличие от всех прочих свойств Height и Width в WPF, в Grid они имеют тип System.Windows.GridLength, а не double. Поэтому панель Grid поддерживает три способа задания размера в элементах RowDefinition и ColumnDefinition:

- Абсолютный размер - числовое значение Height илиWidth означает, что размер задан в независимых от устройства. В отличие от других способов задания размера, абсолютные значения не позволяют строкам и столбцам увеличиваться или сжиматься при изменении размера самой сетки Grid или находящихся внутри нее элементов.
- Автоматический выбор размера – если Height или Width равно, то дочерним элементам выделяется столько места, сколько необходимо, но не больше. Для строки эта величина равна высоте самого высокого элемента, а для столбца - ширине самого широкого элемента.
- Пропорциональное изменение размера - (иногда называется размером «звездочка») предусмотрен специальный синтаксис задания свойств Height и Width, позволяющий распределить имеющееся пространство поровну или в соответствии с заданными пропорциями. Если задано пропорциональное изменение размера, строка и столбец увеличиваются или сжимаются при изменении размера сетки.

С абсолютным и автоматическим заданием размера все понятно, но вот пропорциональное измерение требует пояснений. Звездочка работает следующим образом:

- Если высота строки или ширина столбца равна *, то соответствующему структурному элементу выделяется все оставшееся место.
- Если размер * задан для нескольких строк или столбцов, то все оставшееся место делится между ними поровну.
- Перед символом * можно указывать коэффициент (например, 2* или 5.5*), тогда соответствующей строке или столбцу будет выделено пропорционально больше места, чем остальным строкам или столбцам, в размере которых присутствует символ *. Столбец шириной 2* всегда в два раза шире столбца шириной * (это означает в точности то же самое, что 1*) в той же самой сетке. Столбец шириной 5.5* в два раза шире столбца шириной 2.75* в той же самой сетке.

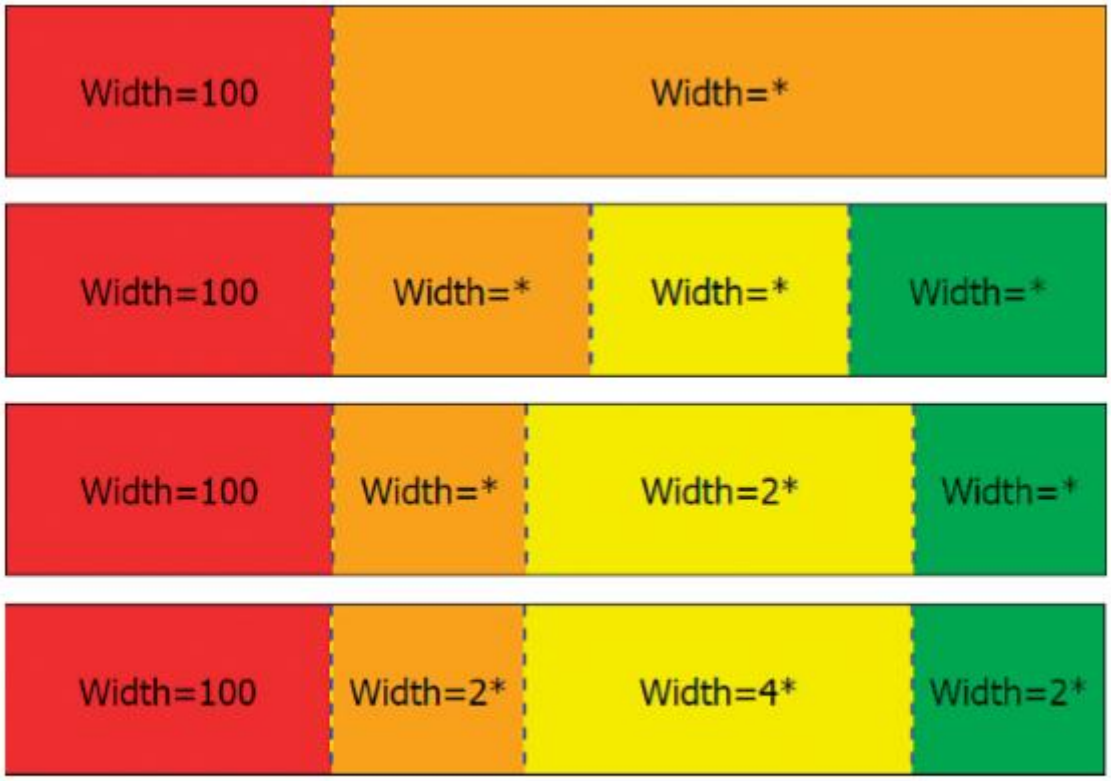


Рис. 5.14. Пропорционально измеренные столбцы сетки

Интерактивное задание размера с помощью GridSplitter

Еще одна привлекательная особенность панели Grid— поддержка интерактивного изменения размера строк и столбцов мышью или клавишами (или стилусом, или пальцем — в зависимости от имеющегося оборудования).

В сетку Grid можно добавить произвольное число дочерних элементов GridSplitter, указав для них присоединенные свойства Grid.Row, Grid.Column, Grid.RowSpan и/или Grid.ColumnSpan.

Буксировка GridSplitter изменяет размер по меньшей мере одной ячейки. Что происходит с остальными - изменение размера или просто перемещение - зависит от заданного способа изменения размера: пропорционально или как-то иначе.

Хотя GridSplitter, по умолчанию располагается в одной ячейке, его действие всегда распространяется на весь столбец (при буксировке по горизонтали) или на всю строку (при буксировке по вертикали). Поэтому лучше задавать для него свойство ColumSpan или RowSpan, так чтобы он пересекал всю сетку.

В классе GridSplitter свойство HorizontalAlignment по умолчанию равно Right, а свойство VerticalAlignment - Stretch, поэтому по умолчанию он примыкает к правой стороне указанной ячейки.

Чтобы элемент GridSplitter был виден и доступен для использования, его ширина Width(или высота Height - в зависимости от ориентации) должна быть задана явно.

В классах RowDefinitions и ColumnDefinitions имеется свойство SharedSizeGroup позволяющее задать режим, при котором линейные размеры нескольких строк и/или столбцов будут оставаться одинаковыми даже в случае, когда размер любой из них изменяется в процессе выполнения программы. Свойству SharedSizeGroup можно присвоить произвольное строковое значение (чувствительное к регистру); оно интерпреруется как имя группы.

Например:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" SharedSizeGroup="myGroup"/> <ColumnDefinition/>
   <ColumnDefinition SharedSizeGroup="myGroup"/>
 </Grid.ColumnDefinitions>
```

Примитивные панели

Панель TabPanel TabPanel очень похожа наWrapPanel, но она поддерживает только горизонтальную ориентацию стопки и вертикальное оборачивание.

Панель ToolBarPanel, по умолчанию используемая в стиле элемента ToolBar, напоминает StackPanel. Однако она работает совместно с панелью переполнения и организует элементы, не умещающиеся в ее пределах

Панель ToolBarOverflowPanel- это упрощенный вариант WrapPanel, поддерживающий только горизонтальную ориентацию стопки и вертикальное оборачивание.

Панель ToolBarTray поддерживает только потомков типа ToolBar. Она компонует элементы ToolBar последовательно (по умолчанию горизонтально) и позволяет перетаскивать их, организуя дополнитеные строки, или сворачивать и разворачивать соседние элементы.

Панель UniformGrid. Это упрощенный вариант сетки Grid, в которой все строки и столбцы имеют размер *. И есть два простых свойства, задающих количество строк и столбцов. Кроме того, в нем нет никаких присоединенных свойств; дочерние элементы добавляются по строкам, и в каждой ячейке может быть только один потомок.

Наконец, если количество строк и столбцов явно не задано, то UniformGrid автоматически выбирает подходящие значения.

Панель SelectiveScrollingGrid - подкласс Grid. В дополнение к функциональности Grid он позволяет «замораживать» некоторые ячейки, не препятствуя прокрутке остальных. Этим поведением управляет свойство SelectiweScrollingOrientation, принимающее следующие значения:

- None - ячейки не могут прокручиваться ни в каком направлении
- Horizontal — ячейки могут прокручиваться только по горизонтали
- Vertical — ячейки могут прокручиваться только по вертикали
- Both - ячейки могут прокручиваться в любом направлении.

Обработка переполнения содержимого

Иногда панели вынуждены выделять потомкам меньше места, чем требуется, и бывает, что потомки отказываются полностью рисовать себя, когда места недостаточно. .В таких случаях возникает проблема переполнения содержимого.

Для ее разрешения можно применять различные стратегии:

- отсечение
- прокрутку
- масштабирование
- оборачивание
- обрезку

Отсечение

Отсечение дочерних элементов - это тот способ, который панели применяют по умолчанию, когда потомков становится слишком много. Отсечение может происходить на краях панели или внутри нее (например, на краях ячейки сетки или в заполняемой области DockPanel).

Во всех классах, производных от UIElement, есть булевское свойство ClipToBounds, которое управляет тем, можно ли рисовать дочерние элементы вне границ родителя. Однако, если внешний край элемента совпадает с внешним краем Window или Page,отсечение все равно производится.

Несмотря на то, что все панели наследуют свойство ClipToBounds,большая их часть автоматически отсекает потомков вне зависимости от значения этого свойства. Но панели Canvas и UniformGrid по умолчанию не отсекают свои дочерние элементы, и обе поддерживают установку для свойства ClipToBounds значения true, чтобы принудительно включить режим отсечения.

Такое поведение означает, что в случае, когда ClipToBounds не равно true, размер Canvas неважен; можно задать Height и Width равными 0, а содержимое равно будет рисоваться, как будто Canvas занимает весь экран!

Отсечение производится до применения преобразований в режиме RenderTransform. Отсечение — часть процесса компоновки и к моменту применения RenderTransform оно уже полностью определено.

Прокрутка

Во многих приложениях критически важна возможность прокручивать содержимое, которое из-за его размера нельзя увидеть целиком. С WPF это просто стоит поместить элемент внутрь элемента управления ScrollViewer, как он сразу же становится прокручиваемым. ScrollViewer применяет элементы управления ScrollBar, которые автоматически присоединяются к содержимому, когда в этом возникает необходимость.

В классе ScrollViewer имеется свойство Content, значением которого может быть один-единственный элемент, обычно в этом качестве выступает некая панель.

Элементы ScrollBar отвечают на различные события ввода, в том числе на нажатия клавиш со стрелками для перемещения на небольшое расстояние, клавиш PageUp и PageDown - на большее расстояние и сочетаний клавиш Ctrl+Home и Ctrl+End для перемещения в начало и конец соответственно.

В классе ScrollViewer имеется еще ряд свойств и методов для манипулирования из программы, но самыми важными являются свойства VerticalScrolBarVisibility и HorizontalScrollBarVisibility. Оба они имеют тип перечисления ScrollBarVisibility, которое определяет четыре состояния полосы прокрутки:

- Visible - полоса прокрутки всегда присутствует, даже если она не нужна. Если необходимости в ней нет, то она выглядит неактивной и не реагирует на события ввода.
- Auto - полоса прокрутки видна, если содержимое нуждается в прокрутке в данном направлении. В противном случае полоса прокрутки отсутствует.
- Hidden- полоса прокрутки всегда невидима, но логически существует, есть содержимое и его можно прокручивать клавишами со стрелками. Поэтому содержимое полностью доступно в данном направлении.
- Disabled- полоса прокрутки не только невидима, но и вообще не существует, то есть прокрутка невозможна ни с помощью клавиатуры, ни посредством мыши.

Масштабирование

В некоторых случаях больше подходит динамическое увеличение или уменьшение содержимого с целью подогнать его под размер имеющейся области. Представьте, к примеру, что вы пишете программу для игры в карты. Очень вероятно, что вы захотите пропорционально масштабировать карты с учетом размера окна.

Класс System.Windows.Controls.Viewbox предлагает простой механизм масштабирования произвольного содержимого в указанном пространстве. Он относится к так называемым декораторам, то есть панелеподобным классам, у которых может быть только один дочерний элемент.

По умолчанию Viewbox растягивается в обоих направлениях, занимая все выделенное ему место. Но у него также имеется свойство Stretch, позволяющее указать, как в занимаемой области должен масштабироваться единственный дочерний элемент.

- None - масштабирование не производится вовсе.

- Fill - размеры дочернего элемента устанавливаются такими же, как размеры самого Viewbox. Поэтому отношение сторон дочернего элемента может не сохраняться.
- Uniform - дочерний элемент масштабируется так, чтобы он целиком поместился внутри Viewbox с сохранением отношения сторон.
- UnitormToFill - дочерний элемент масштабируется так, чтобы он целиком заполнял Viewbox с сохранением отношения сторон. Поэтому, если отношения сторон Viewbox и дочернего элемента не совпадают, то в одном направлении содержимое будет отсечено.

Второе свойство Viewbox StretchDirection позволяет указать, какие операции с содержимым:

- UpOnly - увеличивает содержимое, если необходимо. Если содержимое уже слишком велико, то Viewbox оставляет текущий размер без изменения.
- DownOnly - уменьшает содержимое, если необходимо. Если содержимое уже достаточно мало, то Viewbox оставляет текущий размер без изменения.
- Both - увеличивает или уменьшает содержимое в соответствии с заданным значением описанного выше свойства Stretch. Этот вариант подразумевается по умолчанию.

Обрезка - это более интеллектуальная форма отсечения. Она поддерживается только для текста элементами TextBlock и AccessText, в которых есть свойство TextTrimming, принимающее значения None, CharacterEllipsis или WordEllipsis. В последних двух случаях отброшенный текст заменяется многоточием (...), а не просто прерывается в произвольном месте.

Элементы управления

Обзор элементов управления и их свойств

Чтобы как-то взаимодействовать с пользователем, получать от пользователя ввод с клавиатуры или мыши и использовать введенные данные в программе, нам нужны элементы управления. WPF предлагает нам богатый стандартный набор элементов управления

Все элементы управления могут быть условно разделены на несколько подгрупп:

- Элементы управления содержимым, например кнопки (Button), метки (Label)
- Специальные контейнеры, которые содержат другие элементы, но в отличие от элементов Grid или Canvas не являются контейнерами компоновки - ScrollViewer,GroupBox
- Декораторы, чье предназначение создание определенного фона вокруг вложенных элементов, например, Border или Viewbox.
- Элементы управления списками, например, ListBox, ComboBox.
- Текстовые элементы управления, например, TextBox, RichTextBox.
- Элементы, основанные на диапазонах значений, например, ProgressBar, Slider.
- Элементы для работ с датами, например, DatePicker и Calendar.
- Остальные элементы управления, которые не вошли в предыдущие подгруппы, например, Image.

Все элементы управления наследуются от общего класса System.Window.Controls.Control и имеют ряд общих свойств.

Рассмотрим некоторые из основных свойств, которые наследуются элементами управления.

Name

Наверное важнейшее свойство. По установленному имени впоследствии можно будет обращаться к элементу, как в коде, так и в xaml разметке. Например, в xaml-коде у нас определена следующая кнопка:

```
<Button x:Name="button1" Width="60" Height="30" Content="Текст" Click="button1_Click" />
```

Здесь у нас задан атрибут Click с названием метода обработчика `button1_Click`, который будет определен в файле кода C# и будет вызываться по нажатию кнопки. Тогда в связанном файле кода C# мы можем обратиться к этой кнопке:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    button1.Content = "Привет!";
}
```

Поскольку свойство Name имеет значение `button1`, то через это значение мы можем обратиться к кнопке в коде.

FieldModifier

Свойство `FieldModifier` задает модификатор доступа к объекту:

```
<StackPanel>
    <Button x:FieldModifier="private" x:Name="button1" Content="Hello World" />
    <Button x:FieldModifier="internal" x:Name="button2" Content="Hello WPF" />
</StackPanel>
```

В качестве значения используются стандартные модификатора доступа языка C#: private, protected, internal, protected internal и public. В данном случае объявление кнопок с модификаторами будет равноценно следующему их определению в коде:

```
private Button button1;
internal Button button2;
```

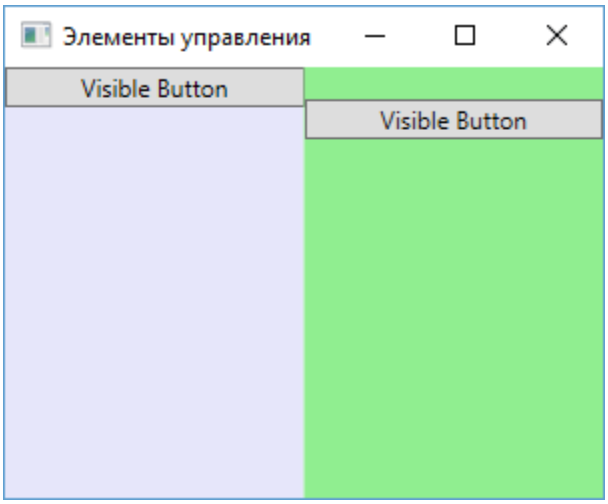
Если для элемента не определен атрибут `x:FieldModifier`, то по умолчанию он равен "protected internal".

Visibility

Это свойство устанавливает параметры видимости элемента и может принимать одно из трех значений:

- Visible - элемент виден и участвует в компоновке.
- Collapsed - элемент не виден и не участвует в компоновке.
- Hidden - элемент не виден, но при этом участвует в компоновке.

Различия между Collapsed и Hidden можно продемонстрировать на примере:



Свойства настройки шрифтов

- FontFamily - определяет семейство шрифта (например, Arial, Verdana и т.д.)
- FontSize - определяет высоту шрифта
- FontStyle - определяет наклон шрифта, принимает одно из трех значений - Normal, Italic,Oblique.
- FontWeight - определяет толщину шрифта и принимает ряд значений, как Black,Bold и др.
- FontStretch - определяет, как будет растягивать или сжимать текст, например, значение Condensed сжимает текст, а Expanded - расстягивает.

Например:

```
<Button Content="Hello World!" FontFamily="Verdana" FontSize="13" FontStretch="Expanded"
/>
```

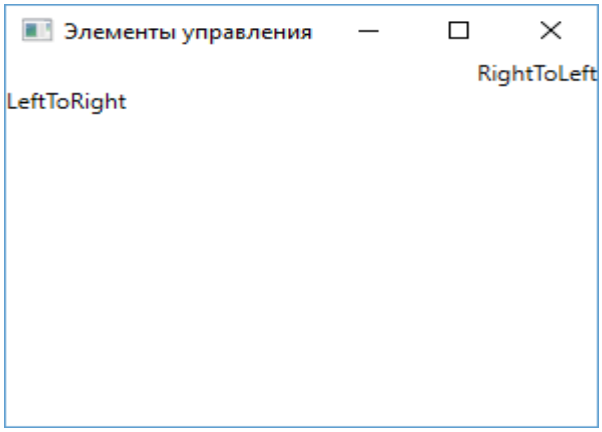
Cursor

Это свойство позволяет нам получить или установить курсор для элемента управления в одно из значений, например, Hand, Arrow, Wait и др. Например, установка курсора в коде c#: `button1.Cursor=Cursors.Hand;`

FlowDirection

Данное свойство задает направление текста. Если оно равно RightToLeft, то текст начинается с правого края, если - LeftToRight, то с левого.

```
<StackPanel>
    <TextBlock FlowDirection="RightToLeft">RightToLeft</TextBlock>
    <TextBlock FlowDirection="LeftToRight">LeftToRight</TextBlock>
</StackPanel>
```



Цвета фона и шрифта

Свойства Background и Foreground задают соответственно цвет фона и текста элемента управления.

Простейший способ задания цвета в коде xaml: `Background="#ffffff"`. В качестве значения свойство Background (Foreground) может принимать запись в виде шестнадцатеричного значения в формате `#rrggbb`, где `rr` - красная составляющая, `gg` - зеленая составляющая, а `bb` - синяя. Также можно задать цвет в формате `#aarrggbb`.

Либо можно использовать названия цветов напрямую:

```
<Button Width="60" Height="30" Background="LightGray" Foreground="DarkRed" Content="Цвет" />
```

Однако при компиляции будет создаваться объект `SolidColorBrush`, который и будет задавать цвет элемента. То есть определение кнопки выше фактически будет равноценно следующему:

```
<Button Width="60" Height="30" Content="Цвет">
    <Button.Background>
        <SolidColorBrush Color="LightGray" />
    </Button.Background>
    <Button.Foreground>
        <SolidColorBrush Color="DarkRed" />
```

```
6     </Button.Foreground>
7</Button>
```

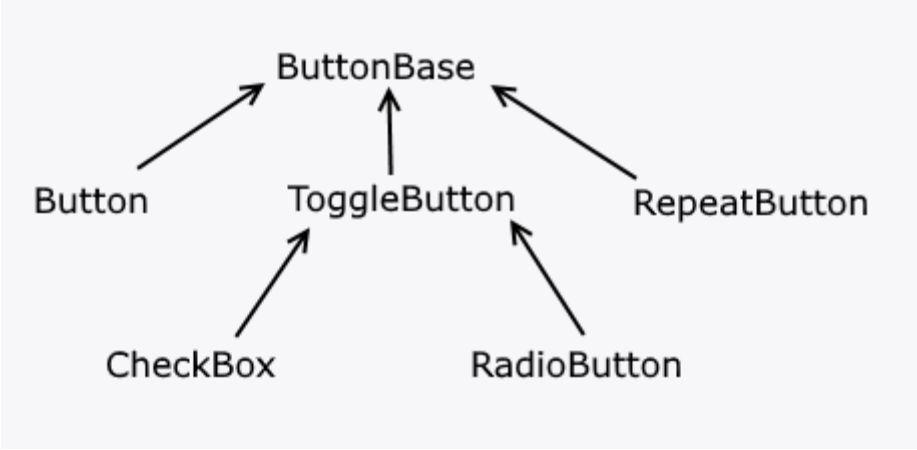
SolidColorBrush представляет собой кисть, покрывающую элемент одним цветом. Позже мы подробнее поговорим о цветах. А пока надо знать, что эти записи эквивалентны, кроме того, вторая форма определения цвета позволяет задать другие кисти - например, градиент.

Это надо также учитывать при установке или получении цвета элемента в коде c#:

```
1button1.Background = new SolidColorBrush(Colors.Red);
2button1.Foreground = new SolidColorBrush(Color.FromRgb(0,255, 0));
```

Класс `Colors` предлагает ряд встроенный цветовых констант, которыми мы можем воспользоваться. А если мы захотим конкретизировать настройки цвета с помощью значений RGB, то можно использовать метод Color.FromRgb.

В WPF кнопки представлены целым рядом классов, которые наследуются от базового класса ButtonBase:



Button

Элемент Button представляет обычную кнопку:

```
1<Button x:Name="button1" Width="60" Height="30" Background="LightGray" />
```

От класса ButtonBase кнопка наследует ряд событий, например, Click, которые позволяют обрабатывать пользовательский ввод.

Чтобы связать кнопку с обработчиком события нажатия, нам надо определить в самой кнопке атрибут `Click`. А значением этого атрибута будет название обработчика в коде C#. А затем в самом коде C# определить этот обработчик.

Например, код xaml:

```
1<Button x:Name="button1" Width="60" Height="30" Content="Нажать" Click="Button_Click" />
```

И обработчик в коде C#:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Кнопка нажата");
}
```

Либо можно не задавать обработчик через атрибут, а стандартным образом для C# прописать в коде: `button1.Click+=Button_Click;`

К унаследованным свойствам кнопка имеет такие свойства как IsDefault и IsCancel, которые принимают значения true и false.

Если свойство IsDefault установлено в true, то при нажатии клавиши Enter будет вызываться обработчик нажатия этой кнопки.

Аналогично если свойство IsCancel будет установлено в true, то при нажатии на клавишу Esc будет вызываться обработчик нажатия этой кнопки.

Например, определим код xaml:

```
<Window x:Class="ControlsApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:ControlsApp"
        mc:Ignorable="d"
        Title="Элементы управления" Height="250" Width="300">
    <StackPanel>
        <Button x:Name="acceptButton" Content="OK" IsDefault="True"
Click="acceptButton_Click" />
        <Button x:Name="escButton" Content="Выход" IsCancel="True"
Click="escButton_Click" />
    </StackPanel>
</Window>
```

А в коде MainWindow.xaml.cs определим следующий код C#:

```
using System.Windows;

namespace ControlsApp
{
```

```
5       public partial class MainWindow : Window
6       {
7           public MainWindow()
8           {
9               InitializeComponent();
10          }
11
12          private void acceptButton_Click(object sender, RoutedEventArgs e)
13          {
14              MessageBox.Show("Действие выполнено");
15          }
16
17          private void escButton_Click(object sender, RoutedEventArgs e)
18          {
19              this.Close(); // закрытие окна
20          }
21      }
22
```

Теперь при нажатии на клавишу Enter будет отображаться сообщение, а при нажатии на Esc будет происходить выход из приложения и закрытие окна.

RepeatButton

Отличительная особенность элемента RepeatButton - непрерывная генерация события `Click`, пока нажата кнопка. Интервал генерации события корректируется свойствами Delay и Interval.

Сам по себе элемент RepeatButton редко используется, однако он может служить основой для создания ползунка в элементах ScrollBar и ScrollViewer, в которых нажатие на ползунок инициирует постоянную прокрутку.

ToggleButton

Представляет элементарный переключатель. Может находиться в трех состояниях - true, false и "нулевом" (неотмеченном) состоянии, а его значение представляет значение типа `bool?` в языке C#. Состояние можно установить или получить с помощью свойства IsChecked. Также добавляет три события - Checked (переход в отмеченное состояние), Unchecked (снятие отметки) и Intermediate (если значение равно null). Чтобы отрабатывать все три события, надо установить свойство IsThreeState="True"

ToggleButton, как правило, сам по себе тоже редко используется, однако при этом он служит основой для создания других более функциональных элементов, таких как checkbox и radiobutton.