Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Отчет по преддипломной практике

Выполнил студент гр. 852001
Ярошевич Ю. А.

Руководитель практики от
предприятия:
начальник отдела
Ковалёв А. М.

Руководитель практики
от университета:
доцент кафедры информатики
Волосевич А. А.

Минск 2013

СОДЕРЖАНИЕ

1 ХАРАКТЕРИСТИКА МЕСТА ПРАКТИКИ

Частная компания ООО «Техартгруп» занимается предоставлением услуг по разработке программного обеспечения, консалтингом и внедрением корпоративных решений для многих европейских и североамериканских компаний. Компания была основана в 2003 году. Штаб-квартира компании расположена в городе Изелин (Iselin), штат Нью-Джерси, Соединенные Штаты Америки. Центры разработки компании расположены в Минске, Беларусь и Киеве, Украина. Среди партнеров компании можно выделить такие общеизвестные компании как Microsoft, Oracle, IBM, Adobe и другие.

Компания предоставляет следующие услуги своим заказчикам [**?**]:

- разработка программного обеспечения;
- интеграция решений в существующую инфраструктуру;
- разработка мобильных приложений;
- миграция больших объемов данных;
- проведение тестирования и контроля качества;
- поддержка существующих решений;
- консалтинг в сфере информационных технологий;
- администрирование баз данных;
- управление инфраструктурой;
- управление проектами.

Рабочая модель взаимодействия с заказчиками — оффшорный центр разработки. Данная модель представляет собой виртуальную команду разработчиков программного обеспечения. Команда создается в соответствии с требованиями клиента относительно проекта и специфики его бизнеса и выступает в качестве удаленного расширения внутреннего штата компании клиента.

В соответствии с требованиями клиентов компания предлагает различные модели взаимодействия. Разработка на стороне клиента — данный подход находит свое применение для наиболее сложных и больших проектов, когда нужно тесное взаимодействие заказчика и команд исполнителя, или когда разработка не может быть передана в другое место, например по причине законодательных ограничений, принятых в стране клиента. Оффшорная организация — вся работа по удовлетворению потребностей заказчика выполняется на стороне компании-исполнителя, данная модель является наиболее экономичной для заказчика. При смешанной организации основная работа выполняется на стороне «Техартгруп», но управление проектом и выработка бизнес требований выполняется представителем «Техартгруп»

на стороне клиента или представителем клиента на стороне компании-исполнителя. Данная модель часто используется для больших проектов, когда нужно соблюдать баланс между стоимостью проекта и эффективностью взаимодействия с клиентом [**?**]. В качестве примера компаний-клиентов «Техартгруп» можно привести следующие компании: Coca-Cola, Disney, FedEx, Gain Capital, 10gen и другие общеизвестные компании.

Для более удобного управления структурная организация компании представляет из себя множество отделов. В компании присутствуют отделы занимающиеся разработкой мобильных приложений для iOS и Android, разработкой приложений для Microsoft .NET, разработкой приложений для платформы Java, также есть отделы разработки, специализирующиеся на других технологиях. Управление компанией осуществляет административный отдел. Также присутствуют отделы материально-технического обеспечения, тестирования и контроля качества. Каждый отдел имеет своего руководителя с которым решаются многие вопросы, возникающие у сотрудников отдела.

Кроме деятельности направленной на зарабатывание денег, компания занимается обучением студентов. В компании с недавнего времени проходят тренинги по веб-разработке и разработке на платформе Microsoft .NET. По результатам курсов многим студентам предлагают работать в компании. Компания также сотрудничает и помогает различным университетам в Беларуси. Благодаря помощи компании был модернизирован студенческий читальный зал №1 в БГУИР.

В компании работает много молодых и зрелых специалистов. Компания хорошо относится к своим сотрудникам, созданы условия для отдыха и развлечения сотрудников. В офисе компании есть комната для отдыха и развлечений, созданная специально для сотрудников. В теплое время года компания часто организует активный отдых за городом для своих работников.

Прохождение преддипломной практики было в команде занимающейся поддержкой и развитием существующей инфраструктуры для компании GAIN Capital Holdings, Inc. Данная компания является пионером в области онлайн торговли иностранными валютами и является владельцем бренда FOREX.com. Результаты, полученные в ходе выполнения индивидуального задания, в данный момент используются компанией GAIN Capital для рассылки уведомлений для своего приложения FOREXTrader for iPhone™. Разработанное решение было гармонично вписано в существующую сервисно-ориентированную архитектуру инфраструктуры GAIN Capital.

2 ИСПОЛЬЗУЕМЫЕ ТЕХНОЛОГИИ

Выбор технологий является важным предварительным этапом разработки сложных информационных систем. Платформа и язык программирования, на котором будет реализована система, заслуживает большого внимания, так как исследования показали, что выбор языка программирования влияет на производительность труда программистов и качество создаваемого ими кода [**?** , с. 59].

Ниже перечислены некоторые факторы, повлиявшие на выбор технологий:

– Разрабатываемое ПО должно работать на операционной системе Windows 7 и более новых версиях системы.

– Среди различных платформ разработки имеющийся программист лучше всего знаком с разработкой на платформе Microsoft .NET.

– Дальнейшей поддержкой проекта, возможно, будут заниматься разработчики, не принимавшие участие в выпуске первой версии.

– Имеющийся разработчик имеет опыт работы с объекто-ориентированными и с функциональными языками программирования.

Основываясь на опыте работы имеющихся программистов разрабатывать ПО целесообразно на платформе Microsoft .NET. Приняв во внимание необходимость обеспечения доступности дальнейшей поддержки ПО, возможно, другой командой программистов, целесообразно не использовать малоизвестные и сложные языки программирования. С учетом этого фактора выбор языков программирования сужается до четырех официально поддерживаемых Microsoft и имеющих изначальную поддержку в Visual Studio 2012: Visual C++/CLI, C#, Visual Basic .NET и F#. Необходимость использования низкоуровневых возможностей Visual C++/CLI в разрабатываемом ПО отсутствует, следовательно данный язык можно исключить из списка кандидатов. Visual Basic .NET уступает по удобству использования двум другим кандидатам из нашего списка. Оставшиеся два языка программирования C# и F# являются первостепенным, элегантными, мультипарадигменными языками программирования для платформы Microsoft .NET. Таким образом, с учетом вышеперечисленных факторов, целесообразно остановить выбор на следующих технологиях:

– операционная система Windows 7;

– платформа разработки Microsoft .NET;

– языки программирования C# и F#.

Для реализации поставленной задачи нет необходимости в использовании

каких-либо прикладных библиотек для создания настольных или веб-приложений, достаточно использовать стандартные библиотеки указанных выше языков. Поддержка платформой Microsoft .NET различных языков программирования позволяет использовать язык, который наиболее просто и «красиво» позволяет решить возникающую задачу. Разрабатываемое программное обеспечение в некоторой степени использует данное преимущество платформы. Язык C# больше подходит для создания высокоуровнего дизайна проложения (иерархия классов и интерфейсов, организация пространств имен и публичного программного интерфейса), язык F# — для реализации логики приложения, функций и методов [**?**], прототипирования различных идей. В разрабатываемом программном продукте C# используется для предоставления удобного программного интерфейса, F# — для прототипирования и реализации вычислительной логики. Далее проводится характеристика используемых технологий и языков программирования более подробно.

2.1 Программная платформа Microsoft .NET

Программная платформа Microsoft .NET является одной из реализаций стандарта ECMA-335 [**?**] и является современным инструментом создания клиентских и серверных приложений для операционной системы Windows. Первая общедоступная версия .NET Framework вышла в феврале 2002 года. С тех пор платформа активно эволюционировала и на данный момент было выпущено шесть версии данного продукта. На данный момент номер последней версии .NET Framework — 4.5. Платформа Microsoft .NET была призвана решить некоторые наболевшие проблемы, скопившиеся на момент её выхода, в средствах разработки приложений под Windows. Ниже перечислены некоторые из них [**?** , с. XIV – XVII]:

– сложность создания надежных приложений;

– сложность развертывания и управления версиями приложений и библиотек;

– сложность создания переносимого ПО;

– отсутствие единой целевой платформы для создателей компиляторов;

– проблемы с безопасным исполнением непроверенного кода;

– великое множество различных технологий и языков программирования, которые не совместимы между собой.

Многие из этих проблем были решены. Далее более подробно рас-

сматривается внутреннее устройство Microsoft .NET.

Основными составляющими компонентами Microsoft .NET являются общая языковая исполняющая среда (Common Language Runtime) и стандартная библиотека классов (Framework Class Library). CLR представляет из себя виртуальную машину и набор сервисов обслуживающих исполнение программ написанных для Microsoft .NET. Ниже приводится перечень задач, возлагаемых на CLR [**?**]:

– загрузка и исполнение управляемого кода;
– управление памятью при размещении объектов;
– изоляция памяти приложений;
– проверка безопасности кода;
– преобразование промежуточного языка в машинный код;
– доступ к расширенной информации от типах — метаданным;
– обработка исключений, включая межъязыковые исключения;
– взаимодействие между управляемым и неуправляемым кодом (в том числе и COM-объектами);
– поддержка сервисов для разработки (профилирование, отладка и т. д.).

Программы написанные для Microsoft .NET представляют из себя набор типов взаимодействующих между собой. Microsoft .NET имеет общую систему типов (Common Type System, CTS). Данная спецификация описывает определения и поведение типов создаваемых для Microsoft .NET [**?**]. В частности в данной спецификации описаны возможные члены типов, механизмы сокрытия реализации, правила наследования, типы-значения и ссылочные типы, особенности параметрического полиморфизма и другие возможности предоставляемые CLI. Общая языковая спецификация (Common Language Specification, CLS) — подмножество общей системы типов. Это набор конструкций и ограничений, которые являются руководством для создателей библиотек и компиляторов в среде .NET Framework. Библиотеки, построенные в соответствии с CLS, могут быть использованы из любого языка программирования, поддерживающего CLS. Языки, соответствующие CLS (к их числу относятся языки C#, Visual Basic .NET, Visual C++/CLI), могут интегрироваться друг с другом. CLS — это основа межъязыкового взаимодействия в рамках платформы Microsoft .NET [**?**].

Некоторые из возможностей, предоставляемых Microsoft .NET: верификация кода, расширенная информация о типах во время исполнения, сборка мусора, безопасность типов, — невозможны без наличия подробных метаданных о типах из которых состоит исполняемая программа. Подроб-

ные метаданные о типах генерируются компиляторами и сохраняются в результирующих сборках. Сборка — это логическая группировка одного или нескольких управляемых модулей или файлов ресурсов, является минимальной единицей с точки зрения повторного использования, безопасности и управлениями версиями [**?** , с. 6].

Одной из особенностей Microsoft .NET, обеспечивающей переносимость программ без необходимости повторной компиляции, является представление исполняемого кода приложений на общем промежуточном языке (Common Intermediate Language, CIL). Промежуточный язык является бестиповым, стековым, объекто-ориентированным ассемблером [**?** , с. 16 – 17]. Данный язык очень удобен в качестве целевого языка для создателей компиляторов и средств автоматической проверки кода для платформы Microsoft .NET, также язык довольно удобен для чтения людьми. Наличие промежуточного языка и необходимость создания производительных программ подразумевают наличие преобразования промежуточного кода в машинный код во время исполнения программы. Одним из компонентов общей языковой исполняющей среды, выполняющим данное преобразование, является компилятор времени исполнения (Just-in-time compiler) транслирующий промежуточный язык в машинные инструкции, специфические для архитектуры компьютера на котором исполняется программа.

Ручное управление памятью всегда являлось очень кропотливой и подверженной ошибкам работой. Ошибки в управлении памятью являются одними из наиболее сложных в устранении типами программных ошибок, также эти ошибки обычно приводят к непредсказуемому поведению программы, поэтому в Microsoft .NET управление памятью происходит автоматически [**?** , с. 505 – 506]. Автоматическое управление памятью является механизмом поддержания иллюзии бесконечности памяти. Когда объект данных перестает быть нужным, занятая под него память автоматически освобождается и используется для построения новых объектов данных. Имеются различные методы реализации такого автоматического распределения памяти [**?** , с. 489]. В Microsoft .NET для автоматического управления памятью используется механизм сборки мусора (garbage collection). Существуют различные алгоритмы сборки мусора со своими достоинствами и недостатками. В Microsoft .NET используется алгоритм пометок (mark and sweep) в сочетании с различными оптимизациями, такими как, например, разбиение всех объектов по поколениям и использование различных куч для больших и малых объектов.

Ниже перечислены, без приведения подробностей, некоторые важные

функции исполняемые общей языковой исполняющей средой:

– обеспечение многопоточного исполнения программы;
– поддержание модели памяти, принятой в CLR;
– поддержка двоичной сериализации;
– управление вводом и выводом;
– структурная обработка исключений;
– возможность размещения исполняющей среды внутри других процессов.

Как уже упоминалось выше, большую ценностью для Microsoft .NET представляет библиотека стандартных классов — соответствующая CLS-спецификации объектно-ориентированная библиотека классов, интерфейсов и системы типов (типов-значений), которые включаются в состав платформы Microsoft .NET. Эта библиотека обеспечивает доступ к функциональным возможностям системы и предназначена служить основой при разработке .NET-приложений, компонент, элементов управления [**?**].

2.2 Язык программирования C#

C# — объектно-ориентированный, типо-безопасный язык программирования общего назначения. Язык создавался с целью повысить продуктивность программистов. Для достижения этой цели в языке гармонично сочетаются простота, выразительность и производительность промежуточного кода, получаемого после компиляции. Главным архитектором и идеологом языка с первой версии является Андрес Хейлсберг (создатель Turbo Pascal и архитектор Delphi). Язык C# является платформенно нейтральным, но создавался для хорошей работы с Microsoft .NET [**?**]. Этот язык сочетает простой синтаксис, похожий на синтаксис языков C++ и Java, и полную поддержку всех современных объектно-ориентированных концепций и подходов. В качестве ориентира при разработке языка было выбрано безопасное программирование, нацеленное на создание надежного и простого в сопровождении кода [**?**].

Язык имеет богатую поддержку парадигмы объекто-ориентированного программирования, включающую поддержку инкапсуляции, наследования и полиморфизма. Отличительными чертами C# с точки зрения ОО парадигмы являются:

– Унифицированная система типов. В C# сущность, содержащая данные и методы их обработки, называется типом. В C# все типы, являются ли они пользовательскими типами, или примитивами, такими как число, про-

изводны от одного базового класса.

– Классы и интерфейсы. В классической объекто-ориентированной парадигме существуют только классы. В C# дополнительно существуют и другие типы, например, интерфейсы. Интерфейс — это сущность напоминающая классы, но содержащая только определения членов. Конкретная реализация указанных членов интерфейса происходит в типах, реализующих данный интерфейс. В частности интерфейсы могут быть использованы при необходимости проведения множественного наследования (в отличие от языков C++ и Eiffel, C# не поддерживает множественное наследование классов).

– Свойства, методы и события. В чистой объекто-ориентированной парадигме все функции являются методами. В C# методы являются лишь одной из возможных разновидностей членов типа, в C# типы также могут содержать свойства, события и другие члены. Свойство — это такая разновидность функций, которая инкапсулирует часть состояния объекта. Событие — это разновидность функций, которые реагируют на изменение состояния объекта [**?**].

В большинстве случаев C# обеспечивает безопасность типов в том смысле, что компилятор контролирует чтобы взаимодействие с экземпляром типа происходило согласно контракту, который он определяют. Например, компилятор C# не скомпилирует код который обращается со строками, как если бы они были целыми числами. Говоря более точно, C# поддерживает статическую типизацию, в том смысле что большинство ошибок типов обнаруживаются на стадии компиляции. За соблюдение более строгих правил безопасности типов следит исполняющая среда. Статическая типизация позволяет избавиться от широкого круга ошибок, возникающих из-за ошибок типов. Она делает написание и изменение программ более предсказуемыми и надежными, кроме того, статическая типизация позволяет существовать таким средствам как автоматическое дополнение кода и его предсказуемый статический анализ. Еще одним аспектом типизации в C# является её строгость. Строгая типизация означает, что правила типизации в языке очень «сильные». Например, язык не позволяет совершать вызов метода, принимающего целые числа, передавая в него вещественное число [**?**]. Такие требования спасают от некоторых ошибок.

C# полагается на автоматическое управление памятью со стороны исполняющей среды, предоставляя совсем немного средств для управления жизненным циклом объектов. Не смотря на это, в языке все же присутствует поддержка работы с указателями. Данная возможность предусмотрена

для случаев, когда критически важна производительность приложения или необходимо обеспечить взаимодействие с неуправляемым кодом [**?**].

Как уже упоминалось C# не является платформенно зависимым языком. Благодаря усилиям компании Xamarin возможно писать программы на языке C# не только для операционных систем Microsoft, но и ряда других ОС. Существуют инструменты создания приложений на C# для серверных и мобильных платформ, например: iOS, Android, Linux и других.

Создатели языка C# не являются противниками привнесения в язык новых идей и возможностей, в отличии от создателей одного из конкурирующих языков. Каждая новая версия компилятора языка привносит различные полезные возможности, которые отчаются требованиям индустрии. Далее приводится краткий обзор развития языка.

Первая версия C# была похожа по своим возможностям на Java 1.4, несколько их расширяя: так, в C# имелись свойства (выглядящие в коде как поля объекта, но на деле вызывающие при обращении к ним методы класса), индексаторы (подобные свойствам, но принимающие параметр как индекс массива), события, делегаты, циклы `foreach`, структуры, передаваемые по значению, автоматическое преобразование встроенных типов в объекты при необходимости (boxing), атрибуты, встроенные средства взаимодействия с неуправляемым кодом (DLL, COM) и прочее [**?**].

Версия Microsoft .NET 2.0 привнесла много новых возможностей в сравнении с предыдущей версией, что отразилось и на языках под эту платформу. Проект спецификации C# 2.0 впервые был опубликован Microsoft в октябре 2003 года; в 2004 году выходили бета-версии (проект с кодовым названием Whidbey), C# 2.0 окончательно вышел 7 ноября 2005 года вместе с Visual Studio 2005 и Microsoft .NET 2.0. Ниже перечислены новые возможности в версии 2.0

– Частичные типы (разделение реализации класса более чем на один файл).

– Обобщённые, или параметризованные типы (generics). В отличие от шаблонов C++, они поддерживают некоторые дополнительные возможности и работают на уровне виртуальной машины. Вместе с тем, параметрами обобщённого типа не могут быть выражения, они не могут быть полностью или частично специализированы, не поддерживают шаблонных параметров по умолчанию, от шаблонного параметра нельзя наследоваться.

– Новая форма итератора, позволяющая создавать сопрограммы с помощью ключевого слова `yield`, подобно Python и Ruby.

– Анонимные методы, обеспечивающие функциональность замыка-

ний.

– Оператор ??: `return obj1 ?? obj2;` означает (в нотации C# 1.0) `return obj1!=null ? obj1 : obj2;`.

– Обнуляемые (nullable) типы-значения (обозначаемые вопросительным знаком, например, `int? i = null;`), представляющие собой те же самые типы-значения, способные принимать также значение null. Такие типы позволяют улучшить взаимодействие с базами данных через язык SQL.

– Поддержка 64-разрядных вычислений позволяет увеличить адресное пространство и использовать 64-разрядные примитивные типы данных [**?**].

Третья версия языка имела одно большое нововведение — Language Integrated Query (LINQ), для реализации которого в языке дополнительно появилось множество дополнительных возможностей. Ниже приведены некоторые из них:

– Ключевые слова `select`, `from`, `where`, позволяющие делать запросы из SQL, XML, коллекций и т. п.

– Инициализацию объекта вместе с его свойствами:

```
Customer c = new Customer(); c.Name = "James"; c.Age=30;
```

можно записать как

```
Customer c = new Customer { Name = "James", Age = 30 };
```

– Лямбда-выражения:

```
listOfFoo.Where(delegate(Foo x) { return x.size > 10; });
```

теперь можно записать как

```
listOfFoo.Where(x => x.size > 10);
```

– Деревья выражений — лямбда-выражения теперь могут быть представлены в виде структуры данных, доступной для обхода во время выполнения, тем самым позволяя транслировать строго типизированные C#-выражения в другие домены (например, выражения SQL).

– Вывод типов локальной переменной: `var x = "hello";` вместо `string x = "hello";`

– Безымянные типы: `var x = new { Name = "James"};`

– Методы-расширения — добавление метода в существующий класс с помощью ключевого слова `this` при первом параметре статической функции.

– Автоматические свойства: компилятор сгенерирует закрытое поле и соответствующие аксессор и мутатор для кода вида

```
public string Name { get; private set; }
```

C# 3.0 совместим с C# 2.0 по генерируемому MSIL-коду; улучшения в языке — чисто синтаксические и реализуются на этапе компиляции [**?**].

Visual Basic .NET 10.0 и C# 4.0 были выпущены в апреле 2010 года, одновременно с выпуском Visual Studio 2010. Новые возможности в версии 4.0:

- Возможность использования позднего связывания.
- Именованные и опциональные параметры.
- Новые возможности COM interop.
- Ковариантность и контрвариантность интерфейсов и делегатов.
- Контракты в коде (Code Contracts) [**?**].

В C# 5.0 было немного нововведений, но они носят большую практическую ценность. В новой версии появилась упрощенная поддержка выполнения асинхронных функций с помощью двух новых слов — `async` и `await`. Ключевым словом `async` помечаются методы и лямбда-выражения, которые внутри содержат ожидание выполнения асинхронных операций с помощью оператора `await`, который отвечает за преобразования кода метода во время компиляции.

2.3 Язык программирования F#

F# — это мультипарадигменный язык программирования, разработанный в подразделении Microsot Research и предназначенный для исполнения на платформе Microsoft .NET. Он сочетает в себе выразительность функциональных языков, таких как OCaml и Haskell с возможностями и объектной моделью Microsoft .NET. История F# началась в 2002 году, когда команда разработчиков из Microsoft Research под руководством Дона Сайма решила, что языки семейства ML вполне подходят для реализации функциональной парадигмы на платформе Microsoft .NET. Идея разработки нового языка появилась во время работы над реализацией обобщённого программирования для Common Language Runtime. Известно, что одно время в качестве прототипа нового языка рассматривался Haskell, но из-за функциональной чистоты и более сложной системы типов потенциальный Haskell.NET не мог бы предоставить разработчикам простого механизма работы с библиотекой классов .NET Framework, а значит, не давал бы каких-то дополнительных преимуществ. Как бы то ни было, за основу был взят OCaml, язык из семейства ML, который не является чисто функциональным и предоставляет возможности для императивного и объектно-ориентированного программирования. Однако Haskell хоть и не стал непосредственно родителем нового

языка, тем не менее, оказал на него некоторое влияние. Например, концепция вычислительных выражений (computation expressions или workflows), играющих важную роль для асинхронного программирования и реализации DSL на F#, позаимствована из монад Haskell [**?**].

Следует также отметить, что на данный момент F# является, пожалуй, единственным функциональным языком программирования, который продвигается одним из ведущих производителей в области разработки программного обеспечения. Он позволяет использовать множество уже существующих библиотек, писать приложения для самых разных платформ и что не менее важно — делать всё это в современной IDE [**?**].

Далее рассматриваются некоторые из возможностей предоставляемых F#.

2.3.1 Функциональная парадигма. Будучи наследником славных традиций семейства языков ML, предоставляет полный набор инструментов функционального программирования: здесь есть алгебраические типы данных и функции высшего порядка, средства для композиции функций и неизменяемые структуры данных, а также частичное применение на пару с каррированием. Все функциональные возможности F# реализованы в конечном итоге поверх общей системы типов .NET Framework. Однако этот факт не обеспечивает удобства использования таких конструкций из других языков платформы. При разработке собственных библиотек на F# следует предусмотреть создание объектно-ориентированных обёрток, которые будет проще использовать из C# или Visual Basic .NET [**?**]. Рекомендации по проектированию таких библиотек приведены в [**?**].

2.3.2 Императивное программирование. В случаях, когда богатых функциональных возможностей не хватает, F# предоставляет разработчику возможность использовать в коде прелести изменяемого состояния. Это как непосредственно изменяемые переменные, поддержка полей и свойств объектов стандартной библиотеки, так и явные циклы, а также изменяемые коллекции и типы данных.

2.3.3 Объектно-ориентированная парадигма. Объектно-ориентированные возможности F#, как и многое другое в этом языке, обусловлены необходимостью предоставить разработчикам возможность использовать стандартную библиотеку классов .NET Framework. С поставленной задачей язык вполне справляется: можно как использовать библиотеки классов, реализованные на других .NET языках, так и разрабатывать свои соб-

ственные. Следует отметить, однако, что некоторые возможности ОО языков реализованы не самым привычным образом [**?**].

2.3.4 Система типов. Каждая переменная, выражение или функция в F# имеет тип. Можно считать, что тип — это некий контракт, поддерживаемый всеми объектами данного типа. К примеру, тип переменной однозначно определяет диапазон значений этой переменной; тип функции говорит о том, какие параметры она принимает и значение какого типа она возвращает; тип объекта определяет то, как этот объект может быть использован [**?**].

F# — статически типизированный язык. Это означает, что тип каждого выражения известен на этапе компиляции, и позволяет отслеживать ошибки, связанные с неправильным использованием объектов определенного типа, до запуска программы. Помимо этого, F# — язык программирования со строгой типизацией, а значит, в выражениях отсутствует неявное приведение типов. Попытка использовать целое число там, где компилятор ожидает увидеть число с плавающей точкой, приведёт к ошибке компиляции [**?**].

2.3.5 Вывод типов. В отличие от большинства других промышленных языков программирования, F# не требует явно указывать типы всех значений. Механизм вывода типов позволяет определить недостающие типы значений, глядя на их использование. При этом некоторые значения должны иметь заранее известный тип. В роли таких значений, к примеру, могут выступать числовые литералы, так как их тип однозначно определяется суффиксом [**?**]. В листинге 2.1 приведен простой пример:

Листинг 2.1 – Пример автоматического вывода типа функции

```
> let add a b = a + b;;
val add : int -> int -> int
> add 3 5;;
val it : int = 8
```

Функция `add` возвращает сумму своих параметров и имеет тип `int -> int -> int`. Если не смотреть на сигнатуру функции, то можно подумать, что она складывает значения любых типов, но это не так. Попытка вызвать её для аргументов типа `float` или `decimal` приведёт к ошибке компиляции. Причина такого поведения кроется в механизме вывода типов. Поскольку оператор `+` может работать с разными типами данных, а никакой дополнительной информации о том, как эта функция будет использоваться, нет, компилятор по умолчанию приводит её к типу `int -> int -> int` [**?**].

В большинстве случаев автоматический вывод типов является очень удобным и способствует написанию полиморфный функций. Алгоритм, используемый компилятором F#, использует глобальный вывод типов и позволяет справляться даже с очень сложными типами. Для демонстрации возможностей вывода типа для полиморфных функций рассмотрим классический пример — комбинаторный базис SKI [**?** , с. 21]:

$$I = \lambda x.x \tag{2.1}$$

$$K = \lambda x\ y.x \tag{2.2}$$

$$S = \lambda f\ g\ x.(f\ x)(g\ x)\,. \tag{2.3}$$

Пример вывода интерактивного окружения F# приведен в листинге 2.2. Как можно заметить типы полученных функций довольно сложные, но компилятор смог их вывести.

Листинг 2.2 – Пример определения комбинаторного базиса SKI

```
> let I x = x;;
val I : x:'a -> 'a

> let K x y = x;;
val K : x:'a -> y:'b -> 'a

> let S f g x = (f x) (g x);;
val S : f:('a -> 'b -> 'c) -> g:('a -> 'b) -> x:'a -> 'c
```

Обычно при программировании на F# в функциональном стиле нет необходимости указывать типы явно, транслятор сам назначит выражению наиболее общий тип. Однако, иногда бывает полезно ограничить вывод типа. Подобная мера не заставит работать код, который до этого не работал, но может использоваться как документация для понимания его предназначения; также возможно использовать более короткие синонимы для сложных типов. Ограничение типа может быть задано в F# путём добавления аннотации типа после некоторого выражения. Аннотации типов состоят из двоеточия, за которым указан тип. Обычно расположение аннотаций не имеет значения; если они есть, то они заставляют компилятор использовать соответствующие ограничения [**?** , с. 59].

2.3.6 Стратегии вычислений. Обычно выражения в F# вычисляются «энергично». Это означает, что значение выражения будет вычислено независимо от того, используется оно где-либо или нет. В противоположность жадному подходу существует стратегия ленивых вычислений, которая

позволяет вычислять значение выражения только тогда, когда оно становится необходимо. Преимуществами такого подхода являются:

– производительность, поскольку неиспользуемые значения просто не вычисляются;

– возможность работать с бесконечными или очень большими последовательностями, так как они никогда не загружаются в память полностью;

– декларативность кода. Использование ленивых вычислений избавляет программиста от необходимости следить за порядком вычислений, что делает код проще.

Главный недостаток ленивых вычислений — плохая предсказуемость. В отличие от энергичных вычислений, где очень просто определить пространственную и временную сложность алгоритма, с ленивыми вычислениями всё может быть куда менее очевидно. F# позволяет программисту самостоятельно решать, что именно должно вычисляться лениво, а что нет. Это в значительной степени устраняет проблему плохой предсказуемости, так как ленивые вычисления применяются только там, где это действительно необходимо, позволяя сочетать лучшее из обоих миров [**?**].

2.3.7 Сопоставление с образцом. Образец — это описание «формы» ожидаемой структуры данных: образец сам по себе похож на литерал структуры данных (т. е. он состоит из конструкторов алгебраических типов и литералов примитивных типов: целых, строковых и т. п.), однако может содержать метапеременные — «дырки», обозначающие: «значение, которое встретится в данном месте, назовем данным именем» [**?**]. Сопоставление с образцом — основной способ работы со структурами данных в F#. Эта языковая конструкция состоит из ключевого слова `match`, анализируемого выражения, ключевого слова `with` и набора правил. Каждое правило — это пара образец-результат. Всё выражение сопоставления с образцом принимает значение того правила, образец которого соответствует анализируемому выражению. Все правила сопоставления с образцом должны возвращать значения одного и того же типа. В простейшем случае в качестве образцов могут выступать константы:

```
> let xor x y =
      match x, y with
      | true, true -> false
      | false, false -> false
      | true, false -> true
      | false, true -> true ;;
val xor : bool -> bool -> bool
```

В правилах сопоставления с образцом можно использовать символ подчеркивания, если конкретное значение неважно. Если набор правил сопоставления не покрывает всевозможные значения образца, компилятор выдаёт предупреждение. На этапе исполнения, если ни одно правило не будет соответствовать образцу, будет сгенерировано исключение [**?**]. Сопоставление с образцом очень мощный механизм, но иногда и его выразительности недостаточно для описание идеи. Язык F# вводит понятие активных шаблонов, когда шаблон может представлять из себя пользовательскую функцию, которая может содержать дополнительную логику обработки.

2.3.8 Вычислительные выражения. Среди нововведений F# можно особо выделить так называемые вычислительные выражения (computation expressions или workflows). Они являются обобщением генераторов последовательности и, в частности, позволяют встраивать в F# такие вычислительные структуры, как монады и моноиды. Также они могут быть применены для асинхронного программирования и создания DSL [**?**].

Вычислительное выражение имеет форму блока, содержащего некоторый код на F# в фигурных скобках. Этому блоку должен предшествовать специальный объект, который называется еще построителем (builder). Общая форма следующая: `builder { comp-expr }`. Построитель определяет способ интерпретации того кода, который указан в фигурных скобках. Сам код вычисления внешне почти не отличается от обычного кода на F#, кроме того, что в нём нельзя определять новые типы, а также нельзя использовать изменяемые значения. Вместо таких значений можно использовать ссылки, но делать это следует с большой осторожностью, поскольку вычислительные выражения обычно задают некие отложенные вычисления, а последние не очень любят побочные эффекты [**?**].

2.3.9 Асинхронные потоки операций. Асинхронные потоки операций — это один из самых интересных примеров практического использования вычислительных выражений. Код, выполняющий какие-либо неблокирующие операции ввода-вывода, как правило сложен для понимания, поскольку представляет из себя множество методов обратного вызова, каждый из которых обрабатывает какой-то промежуточный результат и возможно начинает новую асинхронную операцию. Асинхронные потоки операций позволяют писать асинхронный код последовательно, не определяя методы обратного вызова явно. Для создания асинхронного потока операций используется построитель `async` [**?**]

3 ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ

Индивидуальное задание состоит из нескольких пунктов:

– Ознакомиться с условиями работы на предприятии, внутренним распорядком, техникой безопасности.

– Изучить новейшие возможности .NET Framework 4.5, углубить познания ASP.NET MVC 4.0, ознакомиться с новыми возможностями C# 5 и F# 3.

– Разработать сервис отправки уведомлений на мобильные устройства под управлением iOS. Разработать систему эмуляции.

– Подготовить отчет по преддипломной практике и подготоить главу пояснительной записки дипломного проекта.

4 ЭТАПЫ ВЫПОЛНЕНИЯ ЗАДАНИЯ

а) Подготовительный этап

1) разработка формальных требований;

2) разработка архитектуры приложения;

3) изучение документации по Apple Push Notification Service.

б) Этап реализации

1) разработка библиотеки отправки уведомлений, язык C#;

2) разработка сервиса, предоставляющего функции отсылки уведомлений, язык C#;

3) разработка эмулятора Apple Push Notification Service, язык F#.

в) Этап тестирования

1) тестирования корректности работы сервиса;

2) тестирование производительности сервиса при работе с эмулятором и с реальным APNS-сервисом;

3) тестирование клиентов, работающих с разработанным сервисом;

4) интеграционное тестирование цепочки «серверное приложение» — «разработанный сервис» — «Apple Push Notification Service» — «FOREXTrader for iPhone™».

г) Этап развертывания

1) развертывание сервиса в среде заказчика.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ