Софийски университет „Св. Климент Охридски”

Факултет по математика и информатика

Катедра „Информационни технологии”

Дипломна работа

Използване на декларативния програмен модел при разработка на софтуерни приложения

Божидар Станоев

юни 2007

Ръководител:доц. Боян Бончев

Съдържание

1. Въведение 4
1.1 Увод.. 4
1.2 Цел и структуриране на работата.. 4
2.

Декларативно програмиране 6
3.

Декларативно програмиранеи модели 10
3.1 Windows Workflow Foundation.. 10
3.1.1 Въведение 10
3.1.2 Поток от активности 11
3.1.3 Активности. Начин на изпълнение 13
3.1.4 Машина на състоянието на потока 16
3.1.5 Динамична промяна на поток от активности 19
3.2 XAML.. 20
3.2.1 Въведение 20
3.2.2 Приложение.. 21
3.3 Business Process Execution Language..................................... 27
3.3.1 Дефиниция 27
3.3.2 Цели.. 27
3.3.3 Поддръжка на тясно програмиране в BPEL 28
3.3.4 Езикът BPEL 29
3.4 Сравнение между WF и BPEL 35
4. Приложимост на декларативния програмен модел за разработка на софтуерни приложения....................................37
4.1 Предимства на декларативния модел 37
4.2 Типове потоци 38
4.3 Приложимост на WF 42
5. Примерно приложение..45
5.1 Увод... 45
5.2 Последователен поток... 45
5.3 Машина на състоянието.. 49
5.4 Приложение хост 51
5.5 Дизайнер на потоци 55
6. Заключение 57
Използвани съкращения 59
Приложение 1...64
Приложение 2...68

Въведение

1.1. Увод

Програмирането е на път да измени своя облик. От чисто императивно в преобладаващата си част то преминава към декларативно. Появяват се редица инструменти и технологии за разработка, при които части от програмата се програмират императивно, а други се описват декларативно и така се комбинират преимуществата на двата модела.

Декларативните програмни модели дават възможност за описание на проблема, вместо за дефиниране на решение. Те предлагат по-добра изразителност и визуализация. Средата за изпълнение включва поддръжка на сложни проблеми като дълго продължаващи потоци или управление на състоянието. Динамичната промяна в модела е лесна за реализиране и това не води до проблеми със сигурността или поддръжката.

Тази работа представя декларативното програмиране, къде е уместно да се прилага и как то влияе върху процеса на разработка на приложението. При представянето са използвани Microsoft технологии – Windows Workflow Foundation (WF), част от .NET Framework 3.0 и XAML, декларативен XML-базиран език.

1.2. Цел и структуриране на работата

Целта на дипломната работа е да се разгледа и представи декларативния програмен модел и се оцени приложимостта му в разработката на софтуерни продукти. За реализиране на целта бе необходимо да се решат следните задачи:

- Да се разгледа същността на декларативния програмен модел.
- Да се представят примерни продукти и среди поддържащи програмния модел (Windows Workflow Foundation, BPEL).
- Да се оцени приложимостта на декларативния модел за разработката на софтуерни продукти.
- Да се разработи и оцени примерно приложение, на базата на разглеждания програмен модел.

Дипломната работа е организирана в шест глави. Във втора глава е направен обзор на декларативното програмиране и езиците за декларативно програмиране. В трета глава е наблегнато на декларативните програмни модели WF и BPEL. WF представлява програмен модел и инструменти за бързо изграждане на приложения чрез поток от активности, а BPEL е XML – базиран език за моделиране на бизнес процеси, който е изпълним. В четвърта глава е направен опит за оценка приложимостта на декларативния модел за разработка на софтуерни приложения. В глава пета са разгледани разработката на примерно приложение с WF и Visual Studio. Представени са предимствата и недостатъците на модела и възможности за бъдещо разширение.

2.Декларативно програмиране

Според дефиницията на декларативно програмиране [1], една програма е декларативна ако тя описва как изглежда даден проблем, вместо това как се решава той. HTML уеб страниците са декларативни, защото те описват как изглежда страницата (шрифт, текст, изображения), но не и как действително ще се покаже на екрана на устройството, с което се разглежда. От друга страна, императивните езици като C и Java, изискват да се посочи точен алгоритъм, който да бъде изпълнен.

Декларативното програмиране е известно от около тридесет години [2], но не е широко разпространено в съвременното приложно програмиране. В същото време обектно ориентираното програмиране доминира навсякъде. Въпреки ограниченията на някои езици като C++ (ясна семантика и автоматично управление на паметта), в обектно ориентирания подход има нещо, което интуитивно се харесва от програмистите. Под въздействие на това са направени множество опити да се включат обектно ориентирани характеристики в декларативното програмиране и някои езици като Prolog имат обектно ориентирани разширения.

Повечето декларативни програмни езици имат в основата си трудове в областта на изкуствения интелект и автоматизираното доказване на теореми, където нуждата от по-високо ниво на абстракция и ясен семантичен модел на програмата е очевиден. Основната характеристика на декларативния програмен език е, че програмата е теория в някаква подходяща логика. От гледна точка на програмиста, програмният модел е издигнат с още едно ниво на абстракция. От това по-високо ниво на абстракция програмистът може да се концентрира върху това какво да бъде изчислено, вместо това как да бъде изчислено. Ако програмата (алгоритъмът) е логика и управление, програмистът дава логиката, без да се съсредоточава върху управлението.

Според J. W. Lloyd декларативното програмиране може да бъде разделено на **силно** и **слабо** [3]. При силното декларативно програмиране програмистът дава само логиката на алгоритъма и цялото управление на информацията се предоставя на системата. При слабото декларативно програмиране освен логиката, трябва да се програмира и управляваща информация с цел да се постигне по-ефективна програма.

Декларативното програмиране има много предимства пред императивното. В декларативните езици програмистът не посочва последователност от операции, а само дефиниции или уравнения, специфициращи връзки. За разлика от императивното програмиране, логическите връзки в декларативното програмиране са независими от реда на изпълнение, без странични ефекти от изчисленията, семантично чисти и лесни за визуално представяне.

Фамилията от декларативни програмни езици има дълга история в академичното общество по компютърни науки и специализирани области на приложение като компилатори, експертни системи и бази данни. Декларативните езици оформят два основни клона. Логическите декларативни езици, като Prolog, са базирани на предикатно смятане от първи ред, което обобщава идеите на Аристотел за предикати със стойности „истина" и „лъжа". Другият клон се състои от функционални декларативни езици като Miranda, Haskell и SML. Функционалните декларативни езици са базирани на I-смятането, разработено от математика Alonzo Church през 30-те години на XX век. I-смятането формализира идеите за рекурсивното приложение на чисти функции за изчисляване на задачи. Въпреки че не е широко известно, последният писък в програмирането, XSLT (разширяем стилов език за трансформиране на XML), е също функционален декларативен език.

От изложеното дотук следва, че декларативните езици позволяват на програмиста да се концентрира върху логиката на алгоритъма (декларативните езици са ориентирани към целта, управлението не

засяга програмиста) [5], докато императивните езици изискват от програмиста да се фокусира както върху логиката, така и върху управлението на алгоритъма. За по-ясно различаване на императивния от декларативния подход е добре да се сравнят моделите им на изчисление. Техните основни характеристики са представени в **Таблица 1**.

Табл.1 Характеристики на декларативния и императивния подход.

	Императивен подход	**Декларативен подход**
Модел на изчисление	Модел на изчисление, базиран на последователни команди стъпка по стъпка	Модел на изчисление, базиран на система, където връзките са специфицирани директно с езика на входните данни
Дефиниция на връзките	Програмата определя точно какъв резултат ще бъде постигнат	Изградени от набор от дефиниции и уравнения, описващи връзките, които специфицират какво трябва да бъде изчислено, а не как точно да бъде изчислено
Присвояване	Деструктивно присвояване на променливи	Недеструктивно присвояване на променливи
Структури от данни	Структурите от данни се променят чрез последователно деструктивно присвояване	Явно представяне на използваните структури от данни
Ред на изпълнение	Редът на изпълнение е критичен, командите имат смисъл само в контекста на предишното изчисление поради странични ефекти	Редът на изпълнение няма значение (липсват странични ефекти)

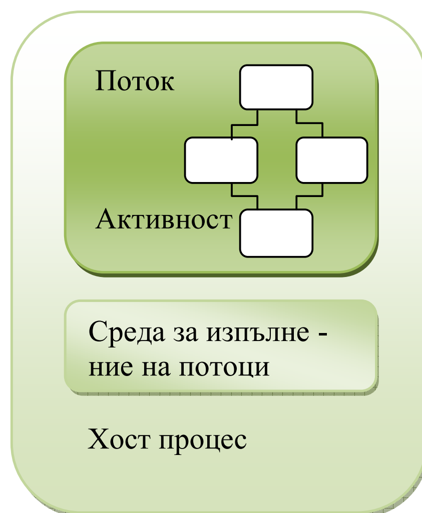
Изрази и дефиниции като стойност	Изразите/ дефинициите не могат да бъдат използвани като стойност	Изразите/ дефинициите могат да бъдат използвани като стойност
Управление	Управлението е отговорност на програмиста	Програмистът не е отговорен за управлението

3. Декларативни програмни модели

3.1. Windows Workflow Foundation (WF)

~~3.2.0.~~3.1.1. Въведение

Windows Workflow Foundation (WF) е програмен модел, изпълнима част и инструменти за бързо изграждане на приложения. Той е част от Microsoft .NET Framework 3.0 (**Фигура 1**).



Фиг.1 Windows Workflow Foundation

WF осезаемо ускорява способността за разработка и поддръжка на бизнес процеси. Поддръжката включва поток от активности между системи и/или хора в широк набор от сценарии: бизнес приложения, потребителски интерфейс, документно ориентирани потоци, сложни потоци за приложения като услуга (SOA). Активност е стъпка в потока и основна единица за изпълнение. Потокът от задачи е набор от активности (activities), съхранени като модел, който описва реален процес. Работата (задачите) минава през модела от началото до края и активностите могат да бъдат изпълнени от хора или системни функции. Потокът дава възможност за описване реда на изпълнение и

връзките между процеси или задачи. Въпреки че е възможно потока да се опише само с код, той се възприема най-добре визуално. След като един поток-модел е компилиран, може да бъде изпълнен в който и да е Windows процес, включително конзолно приложение, Windows Forms приложение, Windows Service, ASP.NET уеб сайт или уеб услуга.

3.1.2 Поток от активности

Блок-диаграмите са графичен модел, използван за да формализира структурата на програма [6]. Квадрати, ромбове и триъгълници представят различни стъпки в диаграмата и представляват активности или решения в алгоритъма. Въпреки че блок-диаграмата е отлично средство за обучение, тя не е директно представена в изпълнимата част на софтуера. По тази причина блок-диаграмите досега се използваха само за документация. След като принципите на блок-диаграмата са усвоени напълно, тя се забравя и програмата се пише директно в код на език за програмиране. Въпреки че писането директно в код е парадигма за разработка повече от 25 години, на която са написани милиони програми, съществуват много проблеми с този подход:

- Код, написан от един разработчик, трудно се възприема от друг. Въпреки че добре документираният код в програмата помага много за рабирането и, смисълът на кода е най-ясен за първия разработчик. Когато той се премести на нова работа или заеме нова роля, се повишава риска в проекта. Ако блок-диаграмата е директно свързана с кода, това със сигурност би подобрило прегледността и яснотата в дизайна на програмата.

- Един път компилиран и изпълнен, кодът е непроменяем и нечетим. Резултатното асембли (assembly) или междинен език се изпълнява върху процесор или виртуална машина напълно

непрозрачно. Това има известни предимства при запазването на интелектуалната собственост, но и редица недостатъци. Когато програмите стават големи, прозрачността в тях става много важна за откриване на грешки по време на изпълнение. Ако модел на блок-диаграма, свързан с кода, е достъпен по време на изпълнението на програмата, това би осигурило прозрачност в изпълнението на програмата.

- За преместването на данни между процедури или обекти има два подхода. Първият е изпращане на съобщение от единия към другия, но той не е много ефективен ако се налага изпращане през верига от множество обекти. Вторият подход централизира идеята за споделено състояние на обект. Моделът блок-диаграма поддържа тази идея за споделено състояние на обект. Ако този модел за достъп до състоянието (извличане, промяна, запазване) на обекта е по-лесен, тогава и управлението на състоянието ще бъде по-ефективно.

WF се състои от следните части:

- **Модел с активности:** Активностите са съставните блокове на потока, те са част от работа, която трябва да бъде изпълнена. Активностите са лесни за създаване с писане на код или от други активности. Съществува базово множество от активности, които осигуряват паралелно изпълнение, условен оператор и извикване на уеб услуга.

- **Дизайнер на поток от активности:** Това е среда за графично проектиране на потоци чрез поставяне на активности в модела.

- **Изпълнима част:** Изпълнява активностите, които изграждат потока. Изпълнимата част може да се хоства (host) във всеки

.NET процес, позволявайки на разработчиците да използват потоците във всякакви приложения от Windows Forms до ASP .NET уеб сайтове или Windows Service.

- **Блок за правила:** Windows Workflow Foundation съдържа блок за правила, който позволява декларативна, базирана на правила разработка на потоци, която да се използва от което и да е .NET приложение.

3.1.3. Активности. Начин на изпълнение.

Потоците в WF са дефинирани като йерархично организирани дървета от компоненти, наричани активности. Активностите са основните единици за изграждане и изпълнение на един поток. Те могат да бъдат представени като стъпка в даден поток. Активностите са идеални за функционална композиция в потока.

3.1.3.1. Компонентен модел

На високо ниво активностите могат да бъдат разделени в две категории: основни и съставни. Основните активности имат предефинирано функционално поведение и свойства, събития и обработчици на събития за конфигуриране. Съставните активности имат същите характеристики, но те могат да служат като контейнери за други активности, основни или съставни. Като частен случай на това главният корен на потока е сам по себе си активност. WF има набор от стандартни, основни и съставни активности за поддръжка на бързата разработка на потоци от разработчиците. Той лесно може да бъде разширяван чрез компонентния модел от активности, който поддържа разработката на нови основни и съставни активности. Новите активности могат да бъдат използвани самостоятелно или

заедно със стандартните активности, за разширение на начина, по който потоците се моделират и изпълняват. Дизайнерът на активности може да бъде използван за създаване на съставни активности без използване на код. Всички активности могат да бъдат създадени и с код. Например следващият C# код дефинира активност:

```
using System;

using System.Workflow.ComponentModel;

// bind an XML namespace to our CLR namespace for XAML

[assembly: XmlnsDefinition( "http://schemas.example.org/MyStuff",
"MyStuff.Activities")]

namespace MyStuff.Activities

{

   public class NoOp : Activity {}

}
```

За да се използва тази активност, се написва прост поток в XAML, подобен на следващия код:

```
<my:NoOp xmlns:my="http://schemas.example.org/MyStuff" />
```

3.1.3.2. Компоненти активности

При създаване на потребителски активности се използва набор от свързани класове, изграждащи WF компонентния модел. Този модел осигурява компонентно-базирана архитектура, където всеки от тези компоненти е отговорен за капсулирането на специфична част от функционалността. Повечето от компонентите се идентифицират с

даден клас. Компонентите активности, част от модела, са следните:

- Activity Definition - задължителен
- Activity Executor
- Activity Validator
- Activity Designer
- Activity Toolbox item

Единственият задължителен клас за автора на активности е Activity Definition. Отделен Activity Executor ще бъде създаден само при създаване на потребителска съставна активност. **Таблица 2** представя накратко всеки компонент активност.

Табл. 2 Компоненти активности

Компонент	Дефиниция
Definition	Използван по време на дизайн и изпълнение за достъпване на характеристики и обработчици на активности.
Executor	Истанциран и управляван по време на изпълнение, за да се изпълни потока съдържащ активност.
Validator	Използван по време на дизайн и изпълнение, за да валидира стойностите на характеристиките и обработчиците на активност.
Designer	Използван в средата Visual Studio 2005 за да моделира активност.
Toolbox item	Използван в средата Visual Studio 2005, за да покаже активност в инструментите и инициализира компонента активност, когато той се влачи от инструментите върху повърхността за моделиране на средата.

3.1.3.3. Поведение на активност

На активностите може да се придаде различно поведение. Това поведение улеснява поддръжката на интелигентни и продължителни потоци и включва:

- Транзакции
- Компенсация
- Обработка на изключения
- Обработка на събития
- Синхронизация

Освен че са достъпни за потребителските активности, тези поведения вече съществуват в последователните потоци. След като някое от поведенията е придадено на потребителска активност в дизайнера на Visual Studio 2005, може да се моделира това поведение.

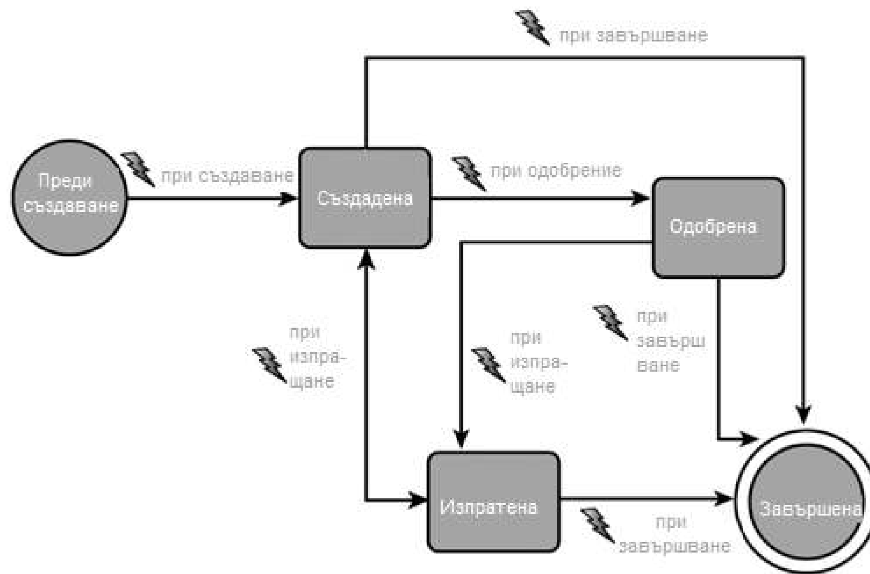
~~3.1.3.~~3.1.4. Машина на състоянието на потока

Бизнес процесите днес са вградени в бизнес приложенията. Такива бизнес приложения са големи монолитни блокове, които са трудни за промяна. В противоречие на това нуждите на бизнеса изискват постоянна промяна и създаването на приложения, които са гъвкави и готови да отговорят на нуждите. Машината на състоянието е подходящ дизайн шаблон за създаване на процеси, използвайки бизнес събития и действия. Той е шаблон, използван дълго време и добре разбиран. Състоянията в машината на състоянието отговарят на състояния в бизнес процесите. Преходите в машината на състоянието имат два компонента: събитието, което предизвиква машината на състоянието да реагира и действието, което се извършва като резултат от промяната на състоянието от текущото към следващото. Начинът, по който машината на състоянието е описана, може да помогне в решаването на редица проблеми. Например в статичният анализ може

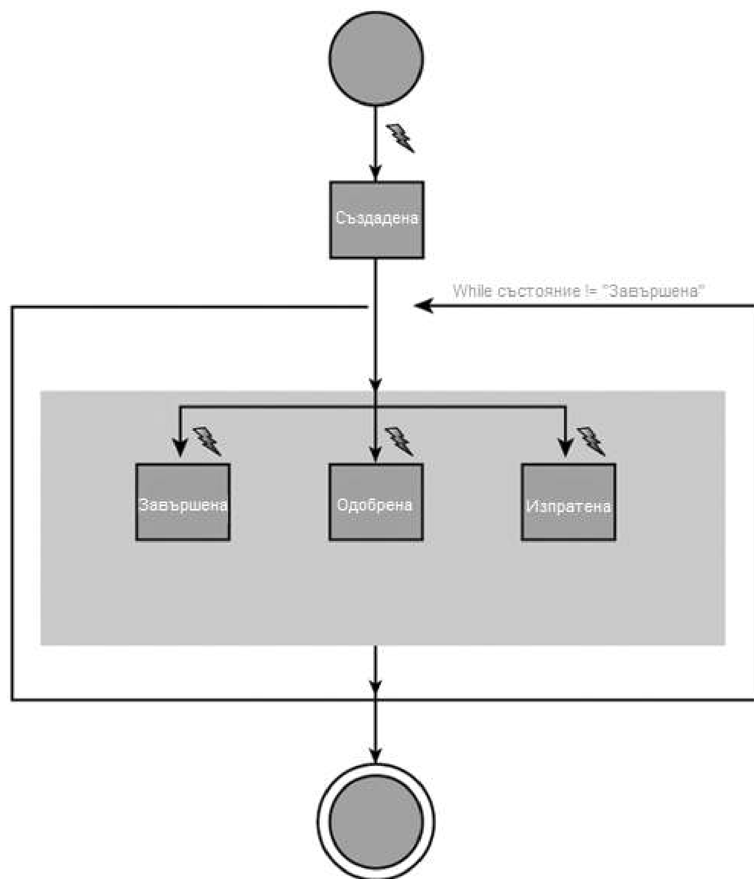
да бъде използвана машина на състоянието, която може да помогне за предсказване на възможните следващи състояния, към които може да се премине. Машината на състоянието е много полезен шаблон, защото осигурява естествен начин за наблюдение на бизнес процеси. Може да се добие бърза представа за процеса като се наблюдава машината на състоянието на процеса. Това е в контраст с последователния процес, където може да се добие информация само за текущата стъпка на изпълнение и информацията за състоянието трябва да се екстраполира от нея.

В много случаи е трудно да се моделира последователен процес. Това особено важи за жизнен цикъл на обекти, като жизнен цикъл на документи или на търговска поръчка. Тези процеси могат да съдържат произволни преходи от едно състояние към друго. Моделирането на тези преходи с последователен поток може да бъде предизвикателство. Конструкциите, които позволяват преходи в последователния дизайн, са циклите и условният оператор. Реализирането на произволни преходи с тези конструкции може да направи бизнеса напълно неразбираем и неуправляем.

Да вземем например жизненият цикъл на документ и да сравним последователната имплементация и такава с машина на състоянието (**Фигури 2 и 3**).



Фиг. 2 Жизнен цикъл на документ, имплементиран като машина на състоянието.



Фиг. 3 Жизнен цикъл на документ, имплементиран като последователен процес.

Може добре да се види лекотата в имплементацията на процеса чрез машина на състоянието. Последователната имплементация е възможна, но трудна за реализиране. Първо концепцията за състоянието е напълно изгубена. То трябва да бъде управлявано от автора на потока. Моделът е труден за поддръжка и текущото състояние трябва да бъде дешифрирано на базата на потребителски променливи.

~~3.1.4.~~3.1.5. Динамична промяна на поток от активности

Динамична промяна е способността за промяна на логиката на потока по време на изпълнение. Поток, който може да бъде модифициран по време на изпълнение, се нарича адаптивен поток. Той може да се адаптира през определено време към променящите се изисквания без необходимост да се изгражда наново. Въпреки че е възможно модифицирането отвън на произволен поток, който не е проектиран да се променя, това не винаги е добра идея. Механизмът за това може да бъде прекалено сложен, ако потока не е разработен с мисълта за възможна промяна. При изграждане на поток, който трябва да бъде гъвкав и отворен за разширение, обикновено е ясно още при проектирането в кои точки ще се наложи разширение. В терминологията на потоците тези точки се наричат „отворени точки” и е добра практика създаването на потоци, които притежават най-доброто от двете – устойчив дизайн и гъвкавост по време на изпълнение.

3.2. XAML

3.2.1. Въведение

Extensible Application Markup Language (XAML) е декларативен XML-базиран език, използван да дефинира обектите и техните свойства, връзките и взаимодействието между тях [1]. Той се използва изключително широко в .NET Framework 3.0 на Microsoft и Windows Workflow Foundation, където служи за дефиниране на потоците от активности. XAML осигурява лесно разширяване и отделяне от логиката на приложението [7], подобно на обектно-ориентираната техника за разработване на n-слойни приложения по MVC (Модел-Изглед-Контролер) архитектура. Тъй като XAML е XML – базиран, кодът, който се пише на него, трябва да е правилен (well-formed) XML. Всеки XAML таг съответства директно на .NET Framework клас, чийто характеристики са контролирани чрез употребата на XML атрибути. Тъй като XAML елементите представляват CLR (Common Language Runtime) обекти, всичко, което може да се напише на XAML, може да се напише и с процедурен код. Има някои части от кода, които се постигат чрез програмно манипулиране на обектния модел и не могат да станат достъпни през XAML. Свойствата, които са само за четене, не се виждат през XAML. Само тези, които са публични и имат get и set методи, са достъпни за разработчиците на XAML.

Събитията и обработчиците на събития могат също да бъдат специфицирани чрез XAML атрибути, а необходимият код зад обработчиците (codebehind) може да бъде написан на който и да е от текущо поддържаните .NET езици (C#, VB.NET). Този код може да бъде вграден в XAML файла или разположен в codebehind файл, подобно на кода в ASP.NET приложение. Ако процедурният код е вграден в XAML страницата, приложението трябва да се компилира преди да може да се изпълни.

За XAML е присъщо да бъде обектно-ориентиран, тъй като неговите елементи представляват CLR класове. Това означава, че елемент, който произхожда от друг XAML елемент, наследява атрибутите на своя родител. Някои XAML елементи изискват деца или атрибути да бъдат от специфичен тип, обикновено някой от базовите класове. Обектно-ориентираното програмиране позволява атрибут да бъде широко дефиниран чрез базов клас и оставя на разработчика възможността да избере специфичен подклас, който да използва. XAML предлага сходни характеристики на другите XML-базирани езици, но той предлага и някои преимущества:

- Отделя интерфейса от логиката и предоставя начин, по който те могат ефективно да си взаимодействат.
- Улесненото моделиране на интерфейси и потоци го прави достъпен за неинженерни кадри, като освобождава ресурс при .NET разработчиците и им позволява да се фокусират върху логиката на приложението.
- XAML е податлив за работа с най-различни инструменти.
- XAML е разширяем, което е част и от неговото име. Той позволява на разработчиците да създадат потребителски контроли и активности.

3.2.2. Приложение

Тук е представена примерна активност, която демонстрира как се използва XAML:

```
namespace MyStuff.Activities
{
   public class WriteLine : Activity
   {
```

```
        string text;
        public string Text
        {
            get { return text; }
            set { text = value; }
        }

        protected override ActivityExecutionStatus  Execute(
            ActivityExecutionContext aec)
        {
            Console.WriteLine(this.Text);
            return ActivityExecutionStatus.Closed;
        }
    }
}
```

Тази активност се отличава с две особености, които трябва да се отбележат. Първо тя дефинира публично свойство с име Text, което може да бъде инициализирано с XAML–базиран поток. Освен това тя предефинира вуртуалния метод Execute и отговаря за отпечатването на свойството Text в конзолата. След като тази активност е дефинирана, можем да напишем поток на XAML:

```
<my:WriteLine Text="Hello, world from WinFX."
    xmlns:my="http://schemas.example.org/MyStuff"
/>
```

Когато този поток се изпълни, програмата ще отпечата низа „Hello, world from WinFX”. С помощта на SequenceActivity активността, можем да изградим съставни активности. В XAML активностите деца се изразяват като елементи на съставната активност. Например:

```
<SequenceActivity
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow"
xmlns:my="http://schemas.example.org/MyStuff"
>
   <my:WriteLine Text="One"/>
   <my:WriteLine Text="Two"/>
   <my:WriteLine Text="Three"/>
   <my:WriteLine Text="Four"/>
</SequenceActivity>
```

Когато се изпълни, този код отпечатва в конзолата:

```
One
Two
Three
Four
```

SequenceActivity активността изпълнява всяка активност-дете последователно. За поддръжка на условно изпълнение, WF предлага активност наречена IfElseActivity. Тази активност съдържа една или повече вложени активности, за всяка от които се оценява булев израз и активността се изпълнява само ако булевият израз се оцени на верен. Например:

```
<!-- MyWorkflow.xaml -->
<SequenceActivity
x:Class="MyNamespace.MyWorkflow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:my="http://schemas.example.org/MyStuff"
>
  <IfElseActivity>
    <IfElseBranchActivity>
      <IfElseBranchActivity.Condition>
        <CodeCondition Condition="Is05"/>
      </IfElseBranchActivity.Condition>
      <my:WriteLine Text="Circa-Whidbey"/>
    </IfElseBranchActivity>
    <IfElseBranchActivity>
      <IfElseBranchActivity.Condition>
        <CodeCondition Condition="Is06"/>
      </IfElseBranchActivity.Condition>
      <my:WriteLine Text="Circa-Vista"/>
    </IfElseBranchActivity>
    <IfElseBranchActivity>
      <my:WriteLine Text="Unknown Era"/>
    </IfElseBranchActivity>
  </IfElseActivity>
</SequenceActivity>
```

Codebehind за потока MyWorkflow:

```
using System;

using System.Workflow.ComponentModel;

using System.Workflow.Activities;


namespace MyNamespace

{

   public partial class MyWorkflow : SequenceActivity

   {

      void Is05(object s, ConditionalEventArgs e)

      {

         e.Result = (DateTime.Now.Year == 2005);

      }


      void Is06(object s, ConditionalEventArgs e)

      {

         e.Result = (DateTime.Now.Year == 2006);

      }

   }

}
```

В този пример са дефинирани два условни израза (Is05 и Is06), към които има обръщение от дефиницията на потока, чрез стандартна XAML нотация. Семантиката на IfElseActivity е проста. Всеки подчинен IfElseBranchActivity има условен израз, които се оценява, за да се определи кои клон да се изпълни. Първият IfElseBranchActivity, който се оцени с истина, се избира за изпълнение. Ако никой от условните изрази не се оцени с истина, последния клон ще бъде избран за изпълнение, ако той няма условен израз, какъвто е случаят.

Разгледаният поток е еквивалентен на следния код, написан на езика C#:

```
if (DateTime.Now.Year == 2005) Console.WriteLine("Circa-Whidbey");
else if (DateTime.Now.Year == 2006) Console.WriteLine("Circa-Vista");
else Console.WriteLine("Unknown era");
```

Въпреки че версията на програмата, базирана на потоци, е с повече код, тя е също и по-прозрачна и лесна за промяна (особено за не C# програмисти) и може да се възползва от услугите, които предлагат потоците като суспенсия (suspension), дехидратация (dehydration) и компенсация (compensation).
Примера за поток с IfElseActivity използва условие, написано с код, за своите булеви изрази. Изразите могат също така да бъдат написани на чист XAML, без да се разчита на отделен codebehind файл. Използвайки XAML–базирани изрази, позволява условията в потока да бъдат изразени в декларативен формат, което е добре дошло за визуалните дизайнери в сравнение с отделния редактор за текстов код, който е специфичен за използвания програмен език.

3.3. Business Process Execution Language (BPEL)

3.3.1. Дефиниция

BPEL е език за моделиране на бизнес процеси, базирани на уеб услуги. Той е изпълним и позволява т.н. програмиране в широко (Programming in the large). Програмиране в широко най-общо представлява взаимодействията при преход между състоянията на дълъг процес, означаван от BPEL като абстрактен процес. Абстрактният процес включва информация като кога се очакват съобщения, кога се изпращат и кога трябва да се компенсира неуспешна транзакция. Програмиране в тясно (Programming in small), от друга страна, се отнася до сравнително късите периоди от изпълнението на една програма: единична транзакция, достъп до локални ресурси като файлове, бази данни и др. BPEL е изграден с идеята за програмиране в широко, а за програмиране в тясно се изискват друг тип езици.

BPEL е език за оркестрация. Оркестрацията е модел, чийто обхват се фокусира тясно върху всяка участваща уеб услуга (модел равен с равен).

3.3.2. Цели

Основните цели при проектирането на BPEL са следните:

- Дефинира бизнес процеси, които взаимодействат с външни обекти през уеб услуги.

- Дефинира бизнес процеси, базирани на XML. Не дефинира графично представяне или методология за проектиране на процеси.

- Осигурява функции за манипулация на данните, необходими за дефиниране на данните за процеса и управляващия поток.

- Поддържа механизъм за идентификация на инстанции на процес, което позволява дефиниране на идентификатори на инстанции на приложно ниво.

- Поддържа неявно създаване и терминиране на инстанции на процес като част от жизнения цикъл.

- Дефинира продължителния транзакционен модел, базиран на добре доказани техники като компенсация и определяне на обхвата за поддържане възстановяване на части от неуспели, продължителни във времето бизнес процеси.

- Използва уеб услуги като модел за асемблиране и декомпозиция на процеси.

3.3.3. Поддръжка на тясно програмиране в BPEL

BPEL е ориентиран към програмиране в широко, което поддържа логиката на бизнес процесите. Тези бизнес процеси са самостоятелни приложения, които използват уеб услуги като активности, които имплементират бизнес функции. BPEL не се опитва да бъде многоцелеви програмен език. Вместо това той се комбинира с други езици, които се използват, за да имплементират бизнес функции (Програмиране в тясно).

BPELJ позволява Java и BPEL да се обединят, позволявайки секции от Java код, наричани Java отрязъци, за да бъдат включени в BPEL дефиниции на процеси. Отрязъците са изрази или малки блокове Java код, които да бъдат използвани за:

- Цикли
- Разклонения
- Инициализация на променливи
- Приготвяне на съобщението за уеб услуги
- Логика на бизнес функциите

3.3.4. Езикът BPEL

3.3.4.1. Увод

Уеб услугите са самостоятелни, модулни бизнес приложения, които са базирани на индустриални стандарти като WSDL (за описание), UDDI (за публикуване), и SOAP (за комуникация) [10]. Те позволяват на потребителите да свързват различни компоненти от различни организации без значение от платформата и използвания език. Свързването на уеб услугите и специфицирането как набор от уеб услуги заедно реализират по-сложна функционалност представлява бизнес процес.

Бизнес процесът специфицира потенциалния ред на изпълнение на операциите от колекция от уеб услуги, данните, споделени между тези уеб услуги, и как те се включват в бизнес процесите. Това позволява описанието на продължителни транзакции между уеб услуги, увеличавайки сигурността и надеждността на приложенията, използващи уеб услуги.

Business Process Execution Language For Web Services (BPEl4WS или накратко BPEL) позволява описанието на бизнес процеси и как те се отнасят към уеб услугите. Бизнес процесите, специфицирани с BPEL, са изпълними и напълно преносими между BPEL–съвместими среди.

3.3.4.2. Партньори

Бизнес процесите, включващи уеб услуги, често взаимодействат с различни партньори. Партньорите са свързани с процес чрез връзка на услугата (service link type). Връзката специфицира две колекции от уеб услуги, които взаимно се използват и изискват от два свързани партьора. Тези колекции от уеб услуги се наричат роли. Следващият код съдържа пример за дефиниция на връзка на услугата:

```
<serviceLinkType name="buyerLink">
   <role name="ticketRequester">
     <portType name="itineraryPT"/>
   </role>
   <role name="ticketService">
     <portType name="ticketOrderPT"/>
   </role>
 </serviceLinkType>
```

Връзката на услугата **buyerLink** се състои от две роли. Ролята **tickerRequester** осигурява порт от тип itineraryPT, а ролята ticketService осигурява порт от тип ticketOrderPT. Типовете портове са дефинирани отделно.

При дефиниране на партньор в бизнес процес се прави указател към връзка на услугата, намираща се в основата на двупосочната

връзка между процес и партньор. Примерна дефиниция на партньори е дадена в следващия код:

```
<partners>
	<partner name="customer"
		serviceLinkType="agentLink"
		myRole="agentService"/>
	<partner name="airline"
		serviceLinkType="buyerLink"
		myRole="ticketRequester"
		partnerRole="ticketService"/>
</partners
```

Дефиницията на партньора специфицира още коя роля от изброените във връзката на услугата приема процесът и коя роля трябва да бъде приета от партньор. Приемането на роля задължава осигуряването на съответните уеб услуги, т.е. осигуряване на имплементация на типовете портове за съответната роля. Уеб услугите, които очаква процесът от партньора, са маркирани с атрибута **partnerRole**, а уеб услугите, които партньорът очаква от процеса, са маркирани с **myRole**.

3.3.4.3. Контейнери. Характеристики.

Бизнес процесите, специфицирани от BPEL, описват размяната на съобщения между уеб услуги. Тези съобщения са WSDL съобщения за операциите и типовете портове, включени в ролите на сервизните връзки, осъществени между процеса и партньорите му. Част от разменяните съобщения могат да бъдат включени в бизнес контекста на бизнес процеса. Този контекст представлява набор от WSDL съобщения, наричани контейнери, които са важни за правилното

изпълнение на бизнес процеса, като определянето на маршрута или конструирането на съобщенията за изпращане.

При размяна на съобщения между партньори трябва да има допълнителни данни, по които да се свърже дадено съобщение с подходящия бизнес процес. Такива свързани данни се наричат характеристика в BPEL. Много често една и съща характеристика се използва с различни съобщения, като данни използвани за връзка. За тази цел BPEL поддържа дефиниция на характеристики като отделни същности. Следва дефиниция на характеристиката **orderNumber**.

```
<property name="orderNumber" type="xsd:int"/
```

3.3.4.4. Активности

Активностите са действия, които се изпълняват в един бизнес процес. Важно действие в бизнес процеса е изчакване да бъде получено съобщение от партньор. Това действие се специфицира с активността **<receive>**. Тя специфицира партньор, от когото да бъде получено съобщението, заедно с порт и операция, осигурена от процеса и използвана от партньора за предаване на съобщението.

Друг по-мощен механизъм за предаване на съобщения е активността **<pick>**. Тази активност специфицира набор от съобщения, които могат да бъдат получени от един или няколко партньора. Когато едно от специфицираните съобщения е получено, <pick> активността е завършена и обработката на бизнес процеса продължава. Допълнително може да се дефинира, че обработката трябва да продължи, ако не е получено съобщение за определен период от време.

Първоначалните активности на бизнес процес трябва да бъдат <receive> или <pick>. Атрибутът **createInstance=yes** показва, че трябва да бъде създадена инстанция на дадения бизнес процес, ако

още не съществува. В повечето случаи след получаване на съобщение трябва да се върне отговор. Следващият код показва **<reply>** активността, използвана за да специфицира синхронен отговор на съответната заявка чрез <receive> активността.

```
<receive partner="customer",
        portType="itineraryPT",
        operation="sendItinerary",
        container="itinerary"
        createInstance="yes"/>
<reply   partner="customer",
        portType="travelPT",
        operation="sendTickets",
        container="tickets"/>
```

Кодът по-горе представлява входно-изходна операция. Входното съобщение се приема от <receive> активността, а изходното съобщение се създава от <replay> активността.

Ако отговорът на заявката трябва да бъде асинхронен, той се изпраща чрез извикване на уеб услуга от страната на заявителя. Оттук и **<invoke>** активността се използва в процес за създаване на асинхронен отговор. Заявителят ще използва <receive>, за да приеме отговора, върнат от <invoke> активността.

Всички споменати дотук активности са прости активности. Това означава, че те не могат да съдържат други активности. Съществуват и структурирани активности, една от които е **<flow>** активността. Тя дефинира набор от активности, които са свързани чрез връзки, с възможност за паралелно изпълнение на части от активността. Всяка връзка може да бъде асоциирана с условие за преход, което е булев израз, използващ променливи от различни контейнери на процеса. Когато процесът се изпълнява, дадена връзка се следва, когато асоциираното условие за преход се оцени на вярно. Други структурни

активности са **<scope>**, която изгражда група от активности и **<sequence>**, която задължава заградените активности да бъдат изпълнявани в реда, в който са изброени.

3.3.4.5. Приложения базирани на процес

Приложенията, създадени с BPEL, са процесно-базирани приложения. Тази структура на приложението го разделя в два отделни слоя: горен слой, написан на BPEL и представляващ управлението на потока на приложението, и долен слой – уеб услугите – представляващ функционалната логика на приложението.

Тази структура има няколко предимства пред конвенционалните подходи. На първо място бизнес процесът в основата, както и извикваните уеб услуги, могат да бъдат променяни без това да се отрази на останалите уеб услуги в приложението или на уеб услугите, които бизнес процесът представлява. Освен това приложението може да бъде тествано в два отделни етапа: бизнес процесите се разработват и тестват отделно от разработката и теста на индивидуалните уеб услуги.

Друго предимство на приложенията, написани на BPEL, е приспособяването на готовото приложение към нуждите и особеностите на дадена среда без промени по самото приложение. Това се постига чрез разделяне на дефиницията на партньорите, с които бизнес процесът взаимодейства, от характеристиките на действително включените партньори. В BPEL се описват само типовете портове и операциите, които различните партньори се очаква да поддържат. Когато даден бизнес процес се изпълнява, информацията за действителните портове или уеб услуги, които конкретен партньор поддържа, трябва да бъде достъпна. Информацията за уеб услуги или портове е включена в указател към услуги. Обикновено указателят на услуги се попълва при инсталиране на процес.

Приложенията, базирани на BPEL, са преносими между средите поддържащи BPEL и уеб услуги. BPEL процесите се изпълняват от BPEL изпълнима среда и по време на изпълнението средата взаимодейства с уеб услугите, които са открити на базата на информация, записана по време на инсталация.

3.4. Сравнение между WF и BPEL

Потоците са много подходящи за грануларната природа на услугите. Уеб услугите изпълняват много малка част от работата, за да бъдат използвани отново в много различни сценарии, което води до възможност за създаване на много различни потоци. Обобщено, потокът представлява серия от услуги, свързани заедно и изпълняващи конкретен бизнес процес. При бизнес процесите изградени от уеб услуги най-често използван и широко поддържан е BPEL. BPEL притежава семантика и сложност да оркестрира различни сценарии с уеб услуги, но той е тясно ориентиран към услугите.

Съществуват редица реални ситуации, където бизнесът изисква интеграция с остарели приложения и неавтоматизирани задачи изпълнявани от човек. Тези потоци излизат извън обхвата на BPEL или други техники досега прилагани в SOA. Тук на преден план излиза WF. Основната предпоставка на WF е, че потокът се простира върху различни слоеве или софтуерни компоненти. Потокът може да се комбинира с настолни приложения, уеб услуги или остарели системи. Това е много универсална абстракция, позволяваща частите на потока да не са услуги, обвързани с SOAP/XML, изисквани от подходи като BPEL.

WF потоците се изпълняват от WF изпълнима среда, аналогична на BPEL изпълнимата среда. За разлика от BEPL средата, която е част от сървърната инсталация, архитектурата зад WF изпълнимата среда и позволява тя да бъде инсталирана не само в класическия контекст

като уеб услуги от страната на сървъра, но и като част от настолно приложение или всяко приложение за .NET FrameWork.

За дефиниране на потоци WF позволява използване на всеки един от текущо използваните езици за .NET платформата, както и предпочитания XAML. Това е много съществено преимущество пред редица платформи за създаване на потоци, които изискват специфични знания и език.

4. Приложимост на декларативния модел за разработка на софтуерни продукти

4.1. Предимства на декларативния модел

За разглеждане на предимствата на декларативния програмен модел е използван WF. Този декларативен програмен модел притежава най-добри възможности за интеграция със съществуващи продукти и езици и универсален начин за изграждане на потоци. Голяма част от разгледаните предимства се отнасят и за другите програмни модели.

Декларативното програмиране чрез потоци и активности ни предлага редица преимущества [8], които могат да бъдат обобщени както следва:

- **Визуализация**. Полезна за разработчика по време на разработка и поддръжка. Може да се използва и от потребителя на потока, който иска да знае какво ще се прави, или от информационния отдел искат да проверят защо приложението се държи странно.

- **Изразителност**. Потокът е език, специфичен за областта (Domain Specific Language), специализиран в описанието на определени проблеми. Пример за това е процесът на рецензия, в който три от пет гласа са необходими, за да може документа да бъде одобрен – всички незавършени рецензии се анулират. Това е досадно да се напише с код, но с помощта на потоци е интуитивно, защото съдържат готови конструкции за справяне с проблема.

- **Изпълнение**. WF оптимизира работата на потока и премахва необходимостта от решаване на подобни задачи повече от един

път. Той включва поддръжка на сложни проблеми като дълго продължаващи потоци, управление на състоянието и компенсация, контролирани от прости и лесни за моделиране елементи.

- **Наблюдение**. Наличието на модел позволява по-лесно наблюдение върху процесите. WF позволява да се вдигат събития към приложението от данни за състоянието на потока.

- **Трансформация**. Моделите пораждат модели. Когато доставчик на софтуер пусне в продажба поток, той е настройваем посредством променливи. Този поток се настройва при клиента, а след това същият поток се трансформира в нова базова версия. Наличието на споделен, добре дефиниран модел на потока прави трансформацията по-гъвкава.

- **Композиция**. Ако е изграден поток и работи, неговите съставни елементи (активности) и самият поток могат да се използват отново в други потоци.

- **Манипулация**. Твърдо често има изисквания за създаване и модифициране на потоци в движение. Ако това означава промяна в кода, то би довело до проблеми със сигурността, дори промяната да е възможна за изпълнение. Използването на модел прави динамичната промяна контролируема и разбираема. WF поддържа динамична промяна на типа и инстанцията на потока.

4.2. Типове потоци

WF поддържа три основни типа потоци: последователен, машина на състоянието и поток от данни. Интересен е въпросът в кой случай кой поток да предпочетем за решаване на конкретен проблем.

Като пример нека разгледаме документ, който трябва да се напише от А, да се рецензира от Б и да се изпрати на клиент В. Това е един последователен поток. За да се реализира, се създава проект с последователен поток и се добавят последователно изброени активности. Последователният поток се характеризира с това, че управлението е в основата на потока. Когато А, Б и В си вършат работата, потокът разбира какво е свършено и решава какво ще последва. Последователният стил не означава винаги просто линейно изпълнение. Може да има условни разклонения, цикли и т.н. Последователният стил е класически стил за поток, имплементиран от много продукти през годините.

Тук е представен като пример още един често срещащ се проблем – тестване на току-що разработен софтуер. Когато се намери бъг, той се насочва към отговорния разработчик. Той го оправя, а тестерът одобрява поправката. Всичко изглежда просто и познато, като рецензирането на документ разгледано по-горе. Но това е само на пръв поглед. Тестерът намира бъг и го насочва към А, но отговорен за бъга всъщност е Б и А го насочва към Б. Б разбира, че това не е бъг и го анулира или иска още информация и т.н. Всеки участник взема определено решение на определен етап от процеса. Ако се направи опит да се реши проблемът в последователен стил, ще се получи смесица от цикли и преходи и мъглява представа дали са изчерпани всички възможни пътища и дали проблемът е решен напълно. Много по-лесно може да се реши проблемът, използвайки машина на състоянието. Примерен модел с псевдо код е представен на **Фигура 4**.

Фиг. 4 Модел с псевдо код за тестване на софтуер

(Т-тестер, D, E, F-разработчици)

Този подход е доста по-ясен и разбираем. Добавянето на повече характеристики няма да усложни модела – просто ще се прибавят повече състояния и действия. Критерият за ползване на машина на състоянието е въпросът: правят ли се важните избори извън потока? Участва ли потребителят във вземането на решение? Ако е така, идеята на последователния поток излиза от употреба. От друга страна машината на състоянието очаква изборът за това какво да прави извън потока. Машината на състоянието поема управлението, когато се попадне в едно от възможните и състояния. Последователният поток, по своята природа, включва всички възможни последователности на поведение в своята структура. Но в този случай не се интересуваме от това. Необходимо е да се познава текущото състояние и следващото. Така че ако се прави опит да се опишат всички възможни преходи с

последователния модел, ще се изгуби много време и почти сигурно напразно.

Остава да се изясни кога се използва потокът от данни. Нека този път примерът е недостиг в склад. Асемблират се компютри и не стигат десет броя от дадена част. Отговорникът на склада се опитва да ги извади от склада, но има недостиг на дадената част. Има няколко решения на този проблем. Отдел доставки може да поръча частта от друг доставчик като заплати повече пари за по-бърза доставка. Счетоводителят може да отиде при клиента и отложи доставката или я раздели на части, вземайки предвид допълнителните разходи за доставка. Продуктовият мениджър може да вземе необходимите части от друга поръчка и да ги пренасочи към тази. Отговорникът на склада може да потърси в склада и да намери липсващите части.

Потокът в този случай е сътрудничество между всички тези дейности, ограничени от съответните роли. Всяко действие може да се повтори много пъти. Очевидно ограничението е дотогава, докато се попълнят липсващите запаси от комбинация от изброените действия.
Това не е последователен поток, защото всички решения се вземат извън потока. Това обаче не е и машина на състоянието, защото наборът от действия в определен момент зависи от много фактори и броят на възможните състояния би бил прекалено голям. Налице е набор от активности и за всяка се изчислява булев израз, за да се определи дали може да се приложи. Освен това необходимо е да се знае кога всичко е приключило и са набавени необходимите запаси. За това се грижи булев израз, който приема стойност истина, когато са набавени запасите. Този управляван от данните поток се имплементира в WF по два начина: чрез активностите Constrained Activity Group и Policy. И двете обикновено се използват в последователен поток, в който има области, където се налага управлявано от данни изпълнение.

Всички останали потоци могат да се имплементират чрез поток, управляван от данни. Това обаче почти никога не е оптимално. За

сравнение може да се вземе последователният поток, при който малкият брой възможни последователности може лесно да се тества на достъпна цена. Така че потокът трябва да се избира много внимателно според структурата на проблема, който се решава. Всяко отклонение би довело до допълнителни разходи. Едно реално приложение почти никога не използва само един от стиловете потоци. Повечето приложения са ефективно изградени от композиция от стилове.

4.3. Приложимост на WF

WF е отворен модел. Той предоставя последователен поток и машина на състоянието, но също така може да се разширява с различни стилове според желанието на потребителя. Активностите деца се изпълняват последователно, но ако се наложи изпълнение в произволен ред – може да се имплементира. Могат да се изграждат нови активности на база на съществуващи такива или като композиция от активности. Или да се направят нови активности от самото начало. Активностите се оприличат на контроли в Windows Forms програмирането, а потоците са съпоставими с формите. Всички приложни интерфейси (API), използвани в WF, са публично достъпни и добре документирани. WF е рамка (framework), която може да се разширява и настройва, докато не отговори на желанието на клиента.

Въпреки че WF има много възможности, той не е подходящ за всеки проект. От работата с него след първото му представяне са се оформили няколко области на широко приложение [9].

- **Изграждане на процесен сървър**

Най-очевидната употреба на WF в проектите е за изграждане на процесен сървър (Business Process Management сървър). Разработчиците, които изграждат процесен сървър, знаят всичко за изпълнението на процесите, колко далече са те от писането на процедурен код и всички фактори на този подход. WF предоставя

на готово поддръжка на управлението на потоци и оставя ресурси на доставчиците на софтуер да се съсредоточат върху характеристиките от по-високо ниво. Но процесните сървъри са сравнително малък процент от пазара. Така че остава въпросът за какво се използва WF в софтуерните проекти?

- **Дълго продължаваща бизнес логика**

WF е добре проектирана да поддържа продължителни във времето бизнес процеси. Част от софтуера не се нуждае от тази характеристика, но повечето големи стопански (enterprise) проекти го изискват. Голяма част от процесите там са продължителни и се прекарва много време в дискутиране със собствениците какво точно трябва да прави софтуера. Разработчиците трябва да преведат бизнес изискванията в процедурен код. Процедурният код не може просто да спре, изчаквайки отговор при дългите процеси, освен ако не е лошо проектиран. Остава задължение на разработчика да наблюдава състоянието, връзката между процесите и кога дадено събитие е настъпило. WF решава този проблем чрез моделиране на процеса на по-високо ниво, отколкото директно в код. Код ще трябва да се пише, но управлението на състоянието, съхраняването и връзката между съобщенията са предоставени от WF.

- **Регулярно променящи се бизнес изисквания или правила**

WF решава друг типичен за софтуера проблем, където приложението е написано и изпратено до много крайни клиенти. Всеки краен клиент има свои изисквания за имплементиране на бизнес логиката и това води до дълъг период на настройка на приложението, докато реално започне да се използва. WF предоставя модел, където набор от активности може да бъде имплементиран с код и за всеки отделен клиент редът на изпълнение на тези активности и характеристиките им да се настройват лесно през дизайнера на WF. Софтуерът ще бъде

изграден предимно в код, но настройката на модела за нов клиент или просто изискване на клиент за промяна става много по-лесно.

- **Прозрачност в изпълнението на бизнес логиката или модела**

След изграждане на бизнес логиката като поток, могат да се използват всички услуги на WF. Това включва (ACID) транзакции, компенсация, управление на изключенията и логване. Услугата за логване осигурява нагледност върху инстанциите на изпълнявания поток и подобно на други услуги може да бъде приложена към потока без писане на допълнителен код. Много често за прибавяне на логване е необходимо да се пише код след като основният софтуер е завършен. Нагледността на потока е осигурена от изгледа на модела в дизайнера на WF, който може да се отпечата или приложи към функционалната спецификация. Това може да бъде особено полезно за представяне на софтуера пред непрофесионалисти в областта или за целите на документацията, когато дизайнът има нужда да бъде учен или модифициран по-късно.

5. Примерно приложение

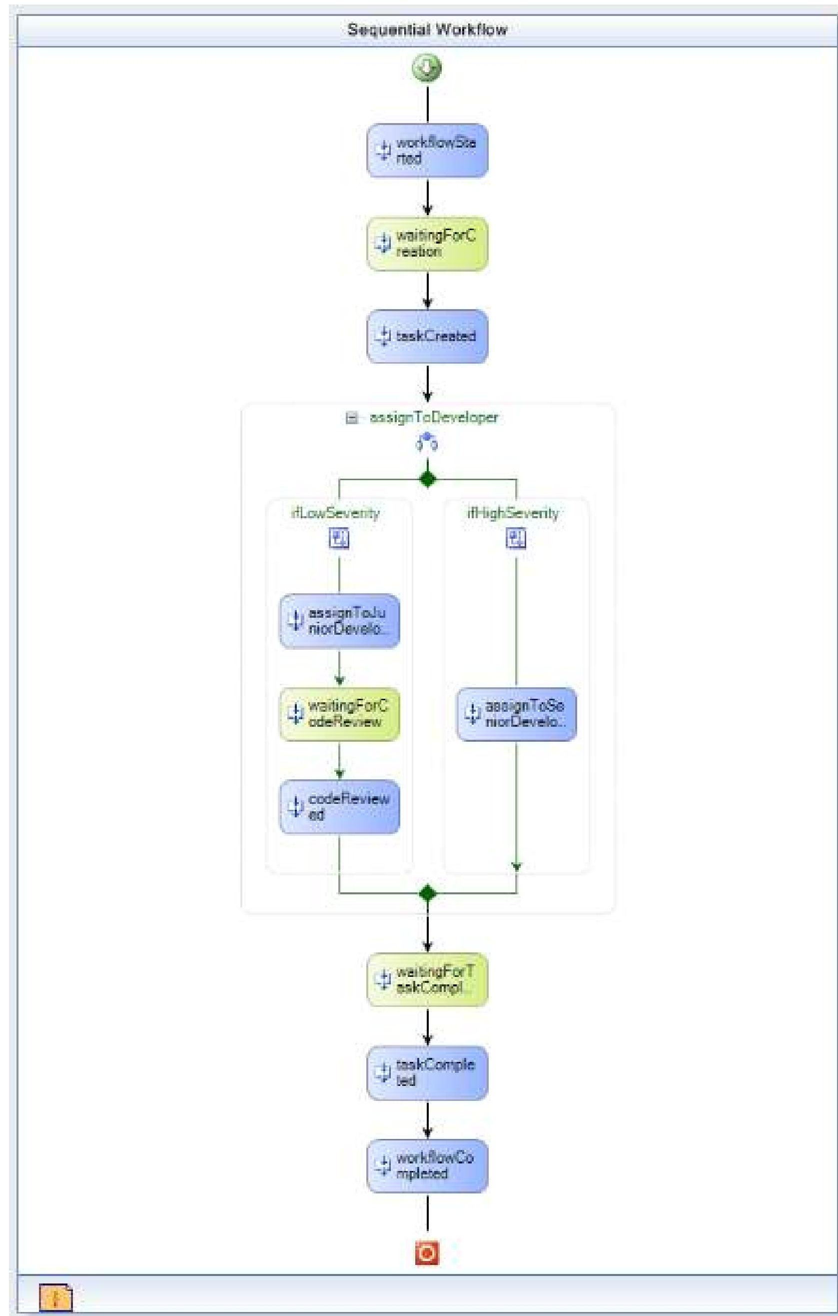
~~5.0.~~5.1. Увод

За представяне на декларативния програмен модел и Windows Workflow Foundation, в **настоящата дипломна работа бе разработено примерно приложение за създаване на задачи**.

Задачи могат да се създават с последователен поток и машина на състоянието. При създаване с последователен поток, задачата трябва да премине през точно определен набор от стъпки, които потокът очаква, за да бъде завършена. При създаване на задачата с машина на състоянието се дефинират краен брой състояния, в които може да се намира задачата. Тя може да преминава между състоянията многократно, като извършва само разрешени преходи. Задачата завършва, когато премине в състояние приключена. То е състояние само с един вход и от него не са разрешени преходи към други състояния.

~~5.1.~~5.2. Последователен поток

С помощта на дизайнера на Visual Studio бе създаден последователен поток за създаване на задача (**Фигура 5**).



Фиг. 5 Последователен поток за създаване на задача

Той се състои от HandleExternalEventActivity активности, с който предаваме информация към потока и CallExternalMethodActivity активности, с които предаваме информация от потока. Потокът се

изпълнява последователно от началото към края по посока на стрелките. При срещане на HandleExternalEventActivity активност изпълнението спира и се очаква настъпването на външно събитие. Такава активност е waitingForCreation, която очаква създаването на задача. След настъпване на събитието изпълнението на потока продължава.

Събитието трябва да бъде дефинирано в интерфейс, който да се имплементира от даден клас.

```
//Интерфейс, който дефинира договора за комуникация
//между локалната услуга и потока
[ExternalDataExchange]
public interface ITaskService
{
	event EventHandler<TaskEventArgs> TaskCreated;
	event EventHandler<TaskEventArgs> TaskRunning;
	event EventHandler<TaskEventArgs> TaskQA;
	event EventHandler<TaskEventArgs> TaskTerminated;
	event EventHandler<TaskEventArgs> TaskCompleted;
	event EventHandler<TaskEventArgs> CodeReviewed;

	void SendMessage(string message);

}
```

При възникване на събитието се предава съобщение (TaskEventArgs) към потока. Това съобщение трябва да бъде сериализируемо и да наследява ExternalDataEventArgs. Потокът ще използва допълнителните свойства на този клас при обработката на събитието. Съобщението предава потребителската информация към потока.

```
//Клас, който дефинира съобщението, предавано между
//локалната услуга и потока
[Serializable]
public class TaskEventArgs : ExternalDataEventArgs
{
private string message;
```

```
public TaskEventArgs(Guid instanceId, string message): base(instanceId)
{
	this.message = message;
}

public string Message
{
	get { return message; }
	set { message = value; }
}

}
```

ifElseActivity Изпълнява съдържащите се активности на базата на някакво правило. Това определя посоката на изпълнение на потока. Правилата могат да бъдат зададени декларативно и императивно (с код). assignToDeveloper е ifElseActivity активност в последователния поток. Тя определя на кой разработчик ще се присвои задачата в зависимост от степента на трудност. Ако задачата е с ниска трудност (low severity), тя се присвоява на младши програмист, като след нея се назначава код ревю. Ако задачата е с висока трудност (high severity), тя се присвоява на старши програмист. За да приключи потокът, е необходимо приключване на задачата. След графичното проектиране на потока, дизайнерът на Visual Studio е генерирал XAML, който е представен в **Приложение 1**.

В приложението декларативно е описан потокът с всички съставящи го активности. Поток може да се създаде и с помощта на текстов редактор и набиране на съответния XAML. Дизайнерът на Visual Studio обаче предоставя редица предимства и е за предпочитане. Правилата са отделени в самостоятелен файл със следното съдържание:

```
<RuleDefinitions
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
	<RuleDefinitions.Conditions>
		<RuleExpressionCondition Name="claculateTaskSeverity">
```

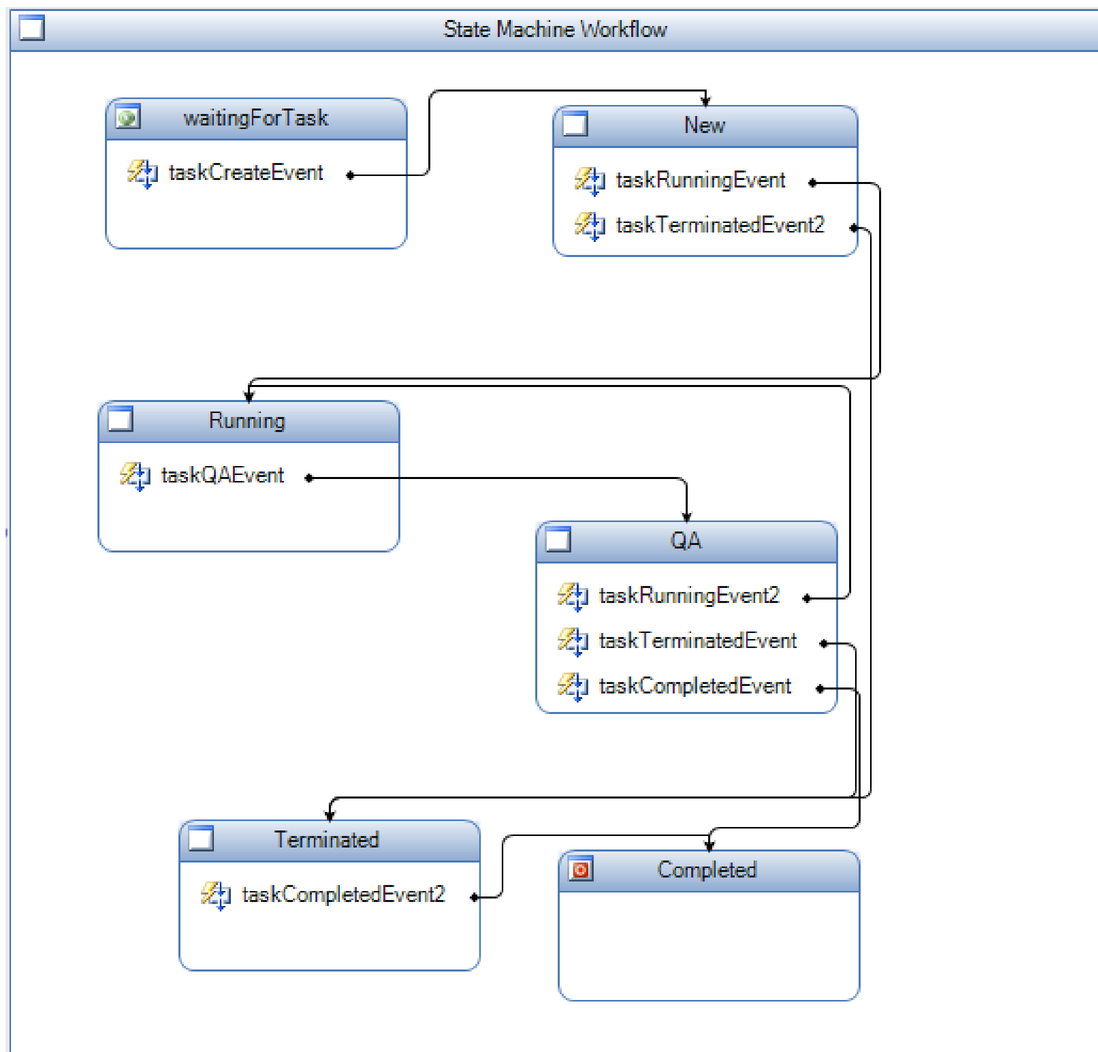
```
                        <RuleExpressionCondition.Expression>
                                <ns0:CodeBinaryOperatorExpression
Operator="ValueEquality" xmlns:ns0="clr-
namespace:System.CodeDom;Assembly=System, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089">
                                        <ns0:CodeBinaryOperatorExpression.Left>
                                                <ns0:CodeFieldReferenceExpression
FieldName="taskSeverity">

        <ns0:CodeFieldReferenceExpression.TargetObject>

        <ns0:CodeThisReferenceExpression />

        </ns0:CodeFieldReferenceExpression.TargetObject>
                                                </ns0:CodeFieldReferenceExpression>
                                        </ns0:CodeBinaryOperatorExpression.Left>
                                        <ns0:CodeBinaryOperatorExpression.Right>
                                                <ns0:CodePrimitiveExpression>

        <ns0:CodePrimitiveExpression.Value>
                                                                <ns1:String
xmlns:ns1="clr-namespace:System;Assembly=mscorlib, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089">Low</ns1:String>

        </ns0:CodePrimitiveExpression.Value>
                                                </ns0:CodePrimitiveExpression>
                                        </ns0:CodeBinaryOperatorExpression.Right>
                                </ns0:CodeBinaryOperatorExpression>
                        </RuleExpressionCondition.Expression>
                </RuleExpressionCondition>
        </RuleDefinitions.Conditions>
</RuleDefinitions>
```

Тук правилото се казва calculateTaskSeverity. Операцията е ValueEquality и се сравнява полето на потока taskSeverity със стринга Low, за да се определи посоката на изпълнение на потока.

~~5.2.~~5.3. Машина на състоянието

Машината на състоянието се проектира по различен начин от последователния поток. Първо бяха определени възможните състояния на задачата. Задължително едно от състоянията е начално, а друго крайно. Състоянията са: Преди създаване (waitingForTask), нова (new), в изпълнение (running), за тестване (qa), завършена (completed) и терминирана (terminated). След това за всяко състояние бяха определени възможните събития. При настъпване на събитие

задачата преминава от едно състояние в друго и се изпълнява някакъв код (**Фигура 6**).



Фиг. 6 Поток машина на състоянието за създаване на задача

Тъй като възможните преходи (събития) са предварително известни, машината на състоянието е много удобна за управление на потребителския интерфейс, като се активират само контролите, които са уместни за даденото състояние. Ако се опитаме да преминем в непозволено състояние, ще възникне изключение. Преход може да се извършва и към текущото състояние. За поток машина на състоянието

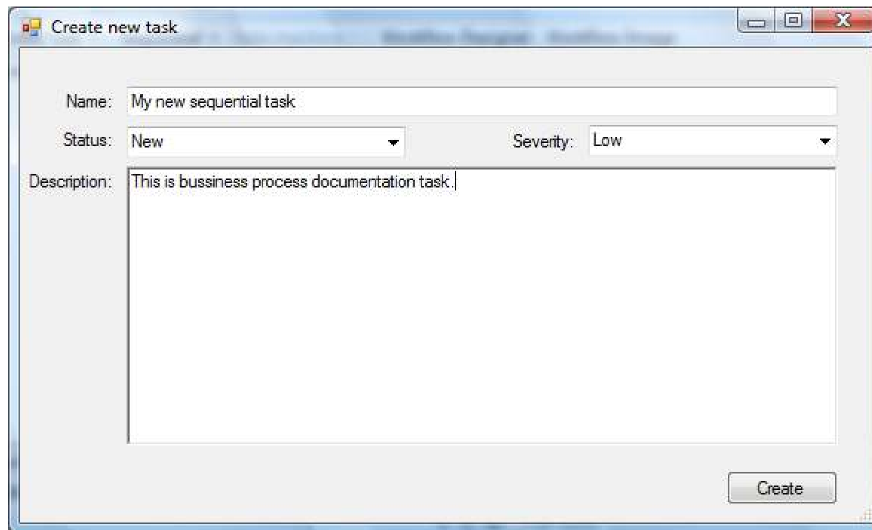
дизайнерът на Visual Studio също генерира XAML, представен в **Приложение 2.**

5.4. Приложение хост

След като е създаден поток, той трябва да се изпълни в приложение хост (host). Такова приложение може да бъде почти всяко Windows приложение (Windows Forms, ASP.NET, Windows Service...), което много ще помогне за широкото разпространение на Windows Workflow Foundation. За хост в настоящата дипломна работа бе използвано Windows Forms приложение (**Фигура 7**):

Фиг. 7 Windows Forms приложение хост

С него може да се създадат задачи, които се изпълняват последователно или като машина на състоянието (**Фигура 8**). Може да се преминава между различните състояния на задачата и да се наблюдава как се изпълнява потокът. На всяка стъпка от изпълнението на потока може да се прегледа графично изображение на потока с подчертана текущо изпълняваната активност.



Фиг. 8 Създаване на нова задача

При създаването на задача се стартира съответен поток (workflow). Преди него обаче трябва да се стартира WorkflowRuntime. WorkflowRuntime е клас, който осигурява среда за изпълнение на потоци. Той вдига събития, когато потокът стартира, терминира или завършва. Стартира се WorkflowRuntime по следния начин:

```
WorkflowRuntime workflowRuntime = new WorkflowRuntime();

ExternalDataExchangeService dataService = new
ExternalDataExchangeService();
workflowRuntime.AddService(dataService);
dataService.AddService(this);

StateMachineTrackingService stateMachineTrackingService = new
StateMachineTrackingService(workflowRuntime);

workflowRuntime.WorkflowCreated += new
EventHandler<WorkflowEventArgs>(workflowRuntime_WorkflowCreated);

workflowRuntime.WorkflowStarted += new
EventHandler<WorkflowEventArgs>(workflowRuntime_WorkflowStarted);
```

```
workflowRuntime.WorkflowCompleted += new
EventHandler<WorkflowCompletedEventArgs>(workflowRuntime_Workflo
wCompleted);

workflowRuntime.StartRuntime();
```

Към WorkflowRuntime се добавя инстанция на ExternalDataExchangeService, която управлява комуникацията между хост приложението и потока, а към нея се прибавя инстанция на клас, който имплементира нашия интерфейс (ITaskService). В случая това е хост приложението, затова се добавя инстанция към него (this). StateMachineTrackingService позволява да се регистрират инстанции на поток машина на състоянието и след това да се регистрира възникване на събитие или промяна на състоянието на потока. След като е създаден и настроен WorkflowRuntime с негова помощ може лесно да се създаде поток:

```
workflowInstance = workflowRuntime.CreateWorkflow(workflowType);
workflowInstance.Start();
```

За да се предаде съобщение към потока, се вдига събитие:

```
public event EventHandler<TaskEventArgs> TaskCompleted;
if (TaskCompleted != null)
{
   TaskCompleted(null, new TaskEventArgs(WFInstance.InstanceId,
"Task completed"));
}
```

Когато се получава информация от потока, обикновено се изпълнява метод на интерфейса (ITaskService). За да се обнови графичния интерфейс, първо трябва да се провери дали извикващият метод е от друга нишка, различна от тази, която е създала контролата. Ако е така, се създава делегат на метода, който прави обновяването и го се извиква чрез метода Invoke на обновяваната контрола. В хост

приложението такъв метод е UpdateListViewItem, който обновява списъка с текущото състояние на потока.

```
private delegate void UpdateListViewItemStatusDelegate(Guid
workflowInstanceID, TaskStatus taskStatus);

private void UpdateListViewItem(Guid workflowInstanceID, TaskStatus
taskStatus)
{
// Ако извикващият е от друга нишка
// различна от тази, създала контролата
if (this.taskStatusListView.InvokeRequired)
{
	// Създаваме делегат на метода и го извикваме
	// чрез Invoke на контролата
	UpdateListViewItemStatusDelegate updateListViewItemDelegate =
	new UpdateListViewItemStatusDelegate(this.UpdateListViewItem);

	object[] args = new object[2] { workflowInstanceID, taskStatus };

	this.taskStatusListView.Invoke(updateListViewItemDelegate,
	args);
}
else
{
	// взема ListViewItem за дадената инстанция
	ListViewItem listViewItem =
	taskStatusListView.Items[workflowInstanceID.ToString()];

	if (listViewItem == null)
	{
		return;
	}

// Обновява колоната за състоянието на потока
listViewItem.SubItems[2].Text = taskStatus.ToString();
}
}
```

При поток машина на състоянието във всяко състояние на потока са разрешени различни събития (преходи). За да се поддържа потребителският интерфейс консистентен и да се позволят само разрешени преходи, трябва да са известни възможните преходи на

всяко състояние и да се активират потребителските контроли, които са уместни за тези преходи.

```
DisableCommands();

StateMachineWorkflowInstance instance = new
    StateMachineWorkflowInstance(WFRuntime, WFInstance.InstanceId);

if (instance.CurrentState == null)
    return;

ReadOnlyCollection<string> transitions =
    instance.PossibleStateTransitions;

if (transitions.Contains("Running"))
    runningToolStripMenuItem.Enabled = true;
if (transitions.Contains("QA"))
    qAToolStripMenuItem.Enabled = true;
if (transitions.Contains("Completed"))
    completedToolStripMenuItem.Enabled = true;
if (transitions.Contains("Terminated"))
    terminatedToolStripMenuItem.Enabled = true;
```

StateMachineWorkflowInstance класът позволява да се изследва поток машина на състоянието. Характеристиката PossibleTransitions на инстанцията на този клас връща колекция от стрингове с всички възможни преходи от текущото състояние.

~~5.4.~~5.5. Дизайнер на потоци

Windows Workflow Foundation осигурява контроли за дизайн на потребителски потоци, активности и правила. Те са подходящи за вграждане в потребителски и бизнес приложения, които използват Windows Workflow Foundation. Workflow Designer Control е контрола дизайнер на потоци. Тя има следните характеристики:

- Визуализира поток в Windows приложение
- Интерактивно проектиране на поток извън средата на Visual

Studio

- Отваряне, съхраняване, компилиране и стартиране на потоци

В примерното приложение Workflow Designer Control се използва за визуализиране на потока във всеки момент от неговото изпълнение, като е подчертана текущо изпълняваната активност.

6. Заключение

Дипломната работа представя декларативния програмен модел и неговото приложение в разработката на програмни продукти.

Windows Workflow Foundation е технология, дефинираща, изпълняваща и управляваща потоци. Тя е част от .NET Framework 3.0 и е едно от най-важните нововъведения в мидълуеъра (middleware) на Windows платформата след COM+.

XAML е декларативен XML-базиран език, използван да дефинира обектите и техните свойства, връзките и взаимодействието между тях. Той осигурява лесно разширяване и отделяне от логиката на приложението.

Business Process Execution Language е XML – базиран език за моделиране на бизнес процеси, който е изпълним.

Предимствата на декларативния програмен модел са визуализация, лесно проектиране и промяна, съсредоточаване върху логиката, а не върху управлението на програмата. Благодарение на тези качества този модел намира все по-широко приложение в съвременните приложения и услуги.

Използването на декларативното програмиране не трябва да е самоцел. Умелото съчетаване на декларативното с императивното програмиране води до по-бързо и по-пълно изпълнение на бизнес целите, вземайки максимума от двата модела.

Настоящата дипломна работа може да се развие, като изследва по-детайлно взаимодействието между декларативната и императивната част на една програма. Друга интересна насока за развитие е предаването на информация между средата за изпълнение на потоци и хост приложението. Това ще отговори на въпроса защо изпълнимата среда може да се прикачва към почти всички видове приложения. Трасирането и проследяването са използвани в примерното приложение за проследяване на статуса на машината на състоянието. Интересно би било да се разгледа в бъдеще как те

позволяват спирането за неопределен период от време на потока и продължаването му след това от същия или друг компютър. По този начин се освобождават ресурси за други приложения, изпълнявани на същия компютър, докато се изчаква настъпването на определено събитие.

Използвани съкращения

ACID – (Atomicity, Consistency, Isolation, Durability) набор от характеристики, които гарантират сигурна обработка на транзакциите в база данни.

Activity – стъпка в потока от активности и единица за изпълнение, повторна употреба и композиция в поток.

API – Application Programming Interface. Интерфейс, който компютърна система или библиотека осигурява за поддръжка на заявки за изграждане на услуги от дадено приложение.

ASP.NET – платформа за създаване на уеб приложения и XML уеб услуги от Microsoft.

Assembly – частично компилирана библиотека с код, използвана при инсталиране, смяна на версиите и обезпечаване на сигурността.

BPEL – Business Process Execution Language. XML – базиран език за моделиране на бизнес процеси, който е изпълним.

Business Process Management – дейности, изпълнявани от бизнеса, за оптимизиране и адаптиране на процесите.

Codebehind – код, който уеб форма използва, разположен във файл, различен от този, който уеб браузъра заявява. Използва се за разделяне на HTML от кода на страницата.

Common Language Runtime – компонент от Microsoft .NET, който дефинира среда за изпълнение на програмен код.

Domain Specific Language – програмен език, проектиран, за да бъде полезен за специфичен набор от задачи.

Host – програма, която приема и обработва заявки и изпраща отговори.

Microsoft .NET Framework – софтуерен компонент, който може да бъде включен или инсталиран допълнително в операционната система Microsoft Windows. Той поддържа изграждането и изпълнението на приложения от следващо поколение и XML уеб услуги.

Middleware – софтуер, който свързва компоненти или приложения. Най-често се използва в разработката на сложни разпределени приложения.

SOA – Service Oriented Architecture. архитектура на приложения, в които всички функции или услуги са дефинирани посредством описателен език и имат интерфейси, които се извикват, за да се изпълнят бизнес процеси.

Visual Studio 2005 – среда за разработка на приложения от Microsoft.

Well-formed XML – XML документа трябва да бъде правилно форматиран, за да може да се прочете. За да бъде правилно форматиран, той трябва да отговаря на повече от сто правила като: документът трябва да съдържа само един основен елемент, таговете не трябва да се застъпват,

имената на елементите трябва да съдържат един или повече символи и да започват с буква и т.н.

Windows Forms – програмен модел за графичен потребителски интерфейс, част от Microsoft .NET Framework, осигуряващ достъп до елементите на интерфейса на Windows, обвивайки съществуващия програмен интерфейс с управляем код.

Windows Service – приложение, което стартира със зареждането на операционната система (ОС) и работи във фонов режим, докато работи самата ОС.

Windows Workflow Foundation - програмен модел, изпълнима част и инструменти за бързо изграждане на приложения.

Workflow – поток от активности, модел на процес на човек или система, които е дефиниран като карта от активности.

WSDL – Web Services Description Language. XML базиран език, който предоставя модел за описание на уеб услуги. Дефинира услугите като набор от мрежови крайни точки и портове.

XAML - Extensible Application Markup Language. Декларативен XML-базиран език използван да дефинира обектите и техните свойства, връзките и взаимодействието между тях.

Литературни източници

[1] Wikipedia. Declarative Programming
http://en.wikipedia.org/wiki/Declarative_programming

[2] A Note on Declarative Programming Paradigms and the Future of Definitional Programming.
http://www.cs.chalmers.se/~oloft/Papers/wm96/wm96.html

[3] Lloyd, J. W. Practical advantages of declarative programming. In Joint Conference on Declarative Programming, GULP-PRODE'94

[4] Furman, R. Declarative Strategies for Solving Software Problems.
http://www.sstnet.com/articles_declarative.html

[5] Coenen, Frans. Declarative languages
http://www.csc.liv.ac.uk/~frans/OldLectures/2CS24/declarative.html

[6] Andrew, Paul. Conrad, James. Woodfate, Scott. Flanders ,Jon. Hatoun, George. Hilerio, Israel. Indurkar, Pravin. Pilarions, Dennis. Willis, Jurgen. 2005. Presenting Windows Workflow Foundation. Beta Edition. Sams.

[7] MacVittie, Lori A. 2006. XAML in a Nutshell. O'Reilly.

[8] Green, Dave. What is Workflow, and why bother.
http://blogs.msdn.com/davegreen/archive/2005/09/17/470704.aspx

[9] Andrew, Paul. What to use Windows Workflow Foundation for?
http://blogs.msdn.com/pandrew/archive/2007/02/01/what-to-use-windows-workflow-foundation-for.aspx

[10] Leymann, Frank. Roller, Dieter. Business processes in a Web services world.
http://www-128.ibm.com/developerworks/webservices/library/ws-bpelwp/

[11] Rubio, Daniel. Windows Workflow Foundation for Web services.
http://searchwebservices.techtarget.com/tip/0,289483,sid26_gci1204115,00.html?bucket=ETA&topic=298926

Приложение 1

XAML код за последователен поток

```
<SequentialWorkflowActivity
x:Class="WorkflowLibrary.SequentialWorkflow"
x:Name="SequentialWorkflow"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
	<CallExternalMethodActivity x:Name="workflowStarted"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}"
MethodName="SendMessage">
		<CallExternalMethodActivity.ParameterBindings>
			<WorkflowParameterBinding
ParameterName="message">
				<WorkflowParameterBinding.Value>
					<ns0:String xmlns:ns0="clr-
namespace:System;Assembly=mscorlib, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089">Sequential
workflow started. (Waiting for task creation)</ns0:String>
				</WorkflowParameterBinding.Value>
			</WorkflowParameterBinding>
		</CallExternalMethodActivity.ParameterBindings>
	</CallExternalMethodActivity>
	<HandleExternalEventActivity Invoked="waitForCreation_Invoked"
x:Name="waitingForCreation" EventName="TaskCreated"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}">
		<HandleExternalEventActivity.ParameterBindings>
			<WorkflowParameterBinding ParameterName="sender">
				<WorkflowParameterBinding.Value>
					<ActivityBind Name="SequentialWorkflow"
Path="TaskSender" />
				</WorkflowParameterBinding.Value>
			</WorkflowParameterBinding>
			<WorkflowParameterBinding ParameterName="e">
				<WorkflowParameterBinding.Value>
					<ActivityBind Name="SequentialWorkflow"
Path="TaskEvtArgs" />
				</WorkflowParameterBinding.Value>
			</WorkflowParameterBinding>
		</HandleExternalEventActivity.ParameterBindings>
	</HandleExternalEventActivity>
	<CallExternalMethodActivity x:Name="taskCreated"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}"
MethodName="SendMessage">
		<CallExternalMethodActivity.ParameterBindings>
```

```
            <WorkflowParameterBinding
ParameterName="message">
                <WorkflowParameterBinding.Value>
                    <ns0:String xmlns:ns0="clr-
namespace:System;Assembly=mscorlib, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089">A task was
created. (Waiting for task assigning)</ns0:String>
                </WorkflowParameterBinding.Value>
            </WorkflowParameterBinding>
        </CallExternalMethodActivity.ParameterBindings>
    </CallExternalMethodActivity>
    <IfElseActivity x:Name="assignToDeveloper">
        <IfElseBranchActivity x:Name="ifLowSeverity">
            <IfElseBranchActivity.Condition>
                <RuleConditionReference
ConditionName="claculateTaskSeverity" />
            </IfElseBranchActivity.Condition>
            <CallExternalMethodActivity
x:Name="assignToJuniorDeveloper" InterfaceType="{x:Type
WorkflowLibrary.ITaskService}" MethodName="SendMessage">
                <CallExternalMethodActivity.ParameterBindings>
                    <WorkflowParameterBinding
ParameterName="message">
                        <WorkflowParameterBinding.Value>
                            <ns0:String xmlns:ns0="clr-
namespace:System;Assembly=mscorlib, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089">Task assigned to
junior developer by workflow. (Waiting for code review)</ns0:String>
                        </WorkflowParameterBinding.Value>
                    </WorkflowParameterBinding>
                </CallExternalMethodActivity.ParameterBindings>
            </CallExternalMethodActivity>
            <HandleExternalEventActivity
x:Name="waitingForCodeReview" EventName="CodeReviewed"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}" />
            <CallExternalMethodActivity x:Name="codeReviewed"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}"
MethodName="SendMessage">
                <CallExternalMethodActivity.ParameterBindings>
                    <WorkflowParameterBinding
ParameterName="message">
                        <WorkflowParameterBinding.Value>
                            <ns0:String xmlns:ns0="clr-
namespace:System;Assembly=mscorlib, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089">Task is code
reviewed. (Waiting for task completion)</ns0:String>
                        </WorkflowParameterBinding.Value>
                    </WorkflowParameterBinding>
```

```
                        </CallExternalMethodActivity.ParameterBindings>
                    </CallExternalMethodActivity>
                </IfElseBranchActivity>
                <IfElseBranchActivity x:Name="ifHighSeverity">
                    <CallExternalMethodActivity
x:Name="assignToSeniorDeveloper" InterfaceType="{x:Type
WorkflowLibrary.ITaskService}" MethodName="SendMessage">
                        <CallExternalMethodActivity.ParameterBindings>
                            <WorkflowParameterBinding
ParameterName="message">
                                <WorkflowParameterBinding.Value>
                                    <ns0:String xmlns:ns0="clr-
namespace:System;Assembly=mscorlib, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089">Task assigned to
senior developer by workflow. (Waiting for task completion)</ns0:String>
                                </WorkflowParameterBinding.Value>
                            </WorkflowParameterBinding>
                        </CallExternalMethodActivity.ParameterBindings>
                    </CallExternalMethodActivity>
                </IfElseBranchActivity>
            </IfElseActivity>
            <HandleExternalEventActivity x:Name="waitingForTaskCompleted"
EventName="TaskCompleted" InterfaceType="{x:Type
WorkflowLibrary.ITaskService}" />
            <CallExternalMethodActivity x:Name="taskCompleted"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}"
MethodName="SendMessage">
                <CallExternalMethodActivity.ParameterBindings>
                    <WorkflowParameterBinding
ParameterName="message">
                        <WorkflowParameterBinding.Value>
                            <ns0:String xmlns:ns0="clr-
namespace:System;Assembly=mscorlib, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089">Task
completed.</ns0:String>
                        </WorkflowParameterBinding.Value>
                    </WorkflowParameterBinding>
                </CallExternalMethodActivity.ParameterBindings>
            </CallExternalMethodActivity>
            <CallExternalMethodActivity x:Name="workflowCompleted"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}"
MethodName="SendMessage">
                <CallExternalMethodActivity.ParameterBindings>
                    <WorkflowParameterBinding
ParameterName="message">
                        <WorkflowParameterBinding.Value>
                            <ns0:String xmlns:ns0="clr-
namespace:System;Assembly=mscorlib, Version=2.0.0.0,
```

```
Culture=neutral, PublicKeyToken=b77a5c561934e089">Sequential
workflow completed.</ns0:String>
                        </WorkflowParameterBinding.Value>
                    </WorkflowParameterBinding>
                </CallExternalMethodActivity.ParameterBindings>
        </CallExternalMethodActivity>
</SequentialWorkflowActivity>
```

Приложение 2

XAML код за поток машина на състоянието

```
<StateMachineWorkflowActivity
x:Class="WorkflowLibrary.StateMachineWorkflow"
InitialStateName="waitingForTask" x:Name="StateMachineWorkflow"
DynamicUpdateCondition="{x:Null}" CompletedStateName="Completed"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
	<StateActivity x:Name="waitingForTask">
		<EventDrivenActivity x:Name="taskCreateEvent">
			<HandleExternalEventActivity
x:Name="waitingForCreateTask" EventName="TaskCreated"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}" />
			<SetStateActivity x:Name="setNewState"
TargetStateName="New" />
		</EventDrivenActivity>
	</StateActivity>
	<StateActivity x:Name="Running">
		<EventDrivenActivity x:Name="taskQAEvent">
			<HandleExternalEventActivity
x:Name="handleTaskQAEvent" EventName="TaskQA"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}" />
			<SetStateActivity x:Name="setStateActivity1"
TargetStateName="QA" />
		</EventDrivenActivity>
	</StateActivity>
	<StateActivity x:Name="QA">
		<EventDrivenActivity x:Name="taskRunningEvent2">
			<HandleExternalEventActivity
x:Name="handleTaskRunningEvent" EventName="TaskRunning"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}" />
			<SetStateActivity x:Name="setStateActivity2"
TargetStateName="Running" />
		</EventDrivenActivity>
		<EventDrivenActivity x:Name="taskTerminatedEvent">
			<HandleExternalEventActivity
x:Name="handleTaskTerminatedEvent2" EventName="TaskTerminated"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}" />
			<SetStateActivity x:Name="setStateActivity4"
TargetStateName="Terminated" />
		</EventDrivenActivity>
		<EventDrivenActivity x:Name="taskCompletedEvent">
			<HandleExternalEventActivity
x:Name="handleTaskCompletedState" EventName="TaskCompleted"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}" />
```

```xml
                        <SetStateActivity x:Name="setStateActivity5"
TargetStateName="Completed" />
                </EventDrivenActivity>
        </StateActivity>
        <StateActivity x:Name="Completed" />
        <StateActivity x:Name="Terminated">
                <EventDrivenActivity x:Name="taskCompletedEvent2">
                        <HandleExternalEventActivity
x:Name="handleTaskCompletedEvent2" EventName="TaskCompleted"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}" />
                        <SetStateActivity x:Name="setStateActivity3"
TargetStateName="Completed" />
                </EventDrivenActivity>
        </StateActivity>
        <StateActivity x:Name="New">
                <EventDrivenActivity x:Name="taskRunningEvent">
                        <HandleExternalEventActivity
x:Name="waitingForRunning" EventName="TaskRunning"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}" />
                        <SetStateActivity x:Name="setRunningState"
TargetStateName="Running" />
                </EventDrivenActivity>
                <EventDrivenActivity x:Name="taskTerminatedEvent2">
                        <HandleExternalEventActivity
x:Name="handleTaskTerminatedEvent" EventName="TaskTerminated"
InterfaceType="{x:Type WorkflowLibrary.ITaskService}" />
                        <SetStateActivity x:Name="setStateActivity6"
TargetStateName="Terminated" />
                </EventDrivenActivity>
        </StateActivity>
</StateMachineWorkflowActivity>
```