УНИВЕРЗИТЕТ „ГОЦЕ ДЕЛЧЕВ“ – ШТИП

Факултет за информатика – Штип

Софтверско инженерство

Васко Башоски

Современи објектно-ориентирани софтверски архитектури

магистерски труд

Штип, 2013 г.

Комисија за оценка и одбрана

Ментор: доц. д-р Сашо Коцески

Факултет за информатика

Член: проф. д-р Цвета Мартиновска-Банде

Факултет за информатика

Член: доц. д-р Александра Милева

Факултет за информатика

Современи објектно-ориентирани софтверски архитектури

Апстракт

Магистерскиот труд се занимава со принципите на дизајн на лесно проширливи и лесно одржливи апликации и како главна цел има да даде насоки за менување на постојните и општоприфатени начини на работа со примена на современи објектно-ориентирани софтверски архитектури кои можат да одговорат на предизвиците при развојот на софтверски решенија со кои се соочуваме на почетокот од втората деценија на 21 век.

Анализите на тековните состојби и принципите за развој на софтвер кои масовно се користат покажуваат дека постојат проблеми кои придонесуваат за намалување на продуктивноста при работа. Најгорлив е проблемот на непостоење на стандарден и функционален метод на неограничено употребување на функционалните компоненти кои веќе еднаш се изработени и тоа, главно, поради тоа што функционалните компоненти се цврсто поврзани со останатите функционални компоненти во рамки на апликациите. Дополнително и документирањето на функционалностите на софтверските апликации е отежнато како последица на отсуството на независност на функционалните компоненти, како и нивниот преголем опсег. Следствено како последица и одржувањето и надградувањето на апликациите во иднина станува сè потешко.

Современата објектно-ориентирана софтверска архитектура која е разработена директно ги тангира откриените слабости и проблеми кои влијаат на продуктивноста при развојот на софтверски апликации и со користење на комбинација од неколку веќе познати принципи и начела се обидува истите да ги разреши. Истата е во насока на зголемување на продуктивноста, но и во поттикнување на соработката помеѓу развивачите на софтверски решенија.

Клучни зборови: *композитни апликации, модул, олеснето одржување, олеснето надградување.*

Modern object-oriented software architectures

Abstract

The master thesis deals with the principles of design easily extensible and easily manageable applications, and has as a main objective to give directions for changing the existing and generally accepted ways of software development by using modern object-oriented software architectures that can meet the challenges of development software solutions that rise up in the beginning of the second decade of the twenty-first century.

The analysis of current conditions and principles for developing software that are in common use, shows that there are problems that extensively contribute to lowered productivity in the process of software development. The main problem is the lack of standardized and functional methods for unlimited usage of functional components that have already been developed and generally because the functional components are tightly coupled with other functional components within the application. Additionally, documentation of the functionalities of the software applications is difficult due to the lack of independence of the functional components as well as their too vide range. Consequently due to this the maintenance and upgrading of the software applications in the future becomes inflexible.

The modern object-oriented software architecture that is elaborated in the thesis directly targets the discovered weaknesses and problems that affect productivity in the development process of software applications by using a combination of several known principles and guidelines tries to resolve them. It is aimed to increase the productivity and in foster the cooperation between developers of software solutions.

Key words: *Composite application, module, maintainability, upgrading.*

Содржина

Современи објектно-ориентирани софтверски архитектури 3

Апстракт 3

Modern object-oriented software architectures..... 4

Abstract 4

1. Вовед (Introduction) 7

2. Анализа на постоечки решенија и проблеми (Analysis of existing problems and solutions) 8

2.1. Анализа на процесот на развој на објектно-ориентирана софтверска апликација 8

2.1.1. Слабо реискористување на елементарните функционалности 9

2.1.2. Квантификација на знаењето 9

2.2. Тенденции кај модерните електронски уреди 10

2.3. Постојни решенија 13

2.3.1. Вертикална модулација на проектите..... 13

2.3.2. Хоризонтална модулација на проектите..... 13

2.3.3. Поглед на MVVM шаблонот..... 15

2.3.4. Начело на единствена одговорност / Single responsibility principle..... 19

2.3.5. Раслојување на одговорностите за одредени типови задачи 20

2.3.6. Модуларен дизајн 24

2.3.7. Модуларно програмирање..... 25

2.3.8. Спарување (врзување) / Coupling 27

2.3.9. Попречни задачи (одговорности) / Cross - cutting concerns 33

2.3.10. Аспект - ориентирано програмирање..... 34

2.3.11. QML..... 35

2.3.12. Постојни имплементации на модуларни решенија 35

3. Предлог-иновативно решение за модуларна архитектура (Innovative modular architecture solution)..... 37

3.1. Опис на Prism библиотеката [17]..... 37

3.1.1. Вовед во Prism 37

3.1.2. Користење на Prism 50

3.1.3. Иницијализација на Prism апликациите..... 55

3.1.4. Управување / менаџирање со зависностите помеѓу компонентите 56

3.1.5. Комуникација помеѓу лабаво врзани компоненти ...70
3.1.6. Составувањето на корисничкиот интерфејс / Composing the User Interface ...73
3.1.7. Развој на модуларни апликации ...74
3.1.8. Prism - Заклучоци ...106
3.2. Windows Phone Архитектура во корелација со Prism ...107
3.2.1. Prism за Windows Phone 7 ...107
3.2.2. Цели во развојот на Windows 8 и Windows Phone 8 ...108
3.3. Библиотеки наменети за мултиоколинско таргетирање / Portable Class Libraries 110
4. Пример за примена на предложената модуларна архитектура(Show case Application) 111
4.1. Корелација на апликацијата со предлог-архитектурата ...111
4.2. Предлог-решение ...112
4.3. Опис на апликацијата ...112
4.3.1. Размена на модули преку xml ...112
5. Заклучок (Concluding remarks) ...115
5.1. Заклучоци за архитектурните предности и недостатоци ...115
5.1.1. Unity наспроти MEF ...115
5.1.2. MEF наспроти Prism ...115
5.1.3. Клучни точки во споредба на Unity, MEF и Prism ...115
5.1.4. Оптичко процесирање и паралелно програмирање ...116
5.2. Значење на тезата ...117
5.3. Предности и недостатоци ...120
5.4. Придобивки од развивањето на модуларни апликации ...122
5.5. Иден развој ...123
6. Користена литература (Bibliography): ...124
6.1. Прилози ...126

1. Вовед (Introduction)

Потребата за развој на софтверски апликации кои треба да опфаќаат значителна бизнис логика, да овозможуваат подобрена корисничка и податочна визуализација, комуникација со мноштво податочни центри, размена на хетерогени податоци и интеракција со различен профил на корисници, а притоа нивното одржување и надградливост да бидат обезбедени на подолг период, наметнува потреба од користење на принципите за развој на нови современи објектно-ориентирани софтверски архитектури познати како принцип за развој на софтвер „дизајниран да трае“ и „дизајниран за лесна надградливост“.

Дополнително кон овие концепти се придружува и концептот за модуларност на апликациите каде што секоја функционалност на апликацијата се дели на елементарни градивни модули, со цел апликациите да може динамички да се надградуваат и менуваат само на одреден дел од апликацијата без притоа да се загрози нејзиното функционирање во целост. Сето тоа е во прилог на основните парадигми во софтверското инженерство и значително би ја олеснило работата во фазите на програмирање и тестирање на модулите кои се ограничени на елементарни функционалности. Дополнителен мотив за користење на овие принципи на дизајн секако е можноста за реискористување на готови модули за најразлични апликации.

За да може една архитектура да се здобие со епитетот современа објектно-ориентирана архитектура за развој на софтвер во моментот (2013 година), во научната заедница се прифатени некои општи карактеристики кои таа треба да ги поседува и тоа:

- Апликациите градени врз основа на архитектурата да може да се развиваат од независни функционални градивни компоненти за кои мора да постои можност да бидат развивани од страна на независни тимови.
- Архитектурата треба да промовира намалување на меѓутимските зависности и зголемување на можностите за интеракција и интеграција меѓу тимови кои се специјализирани за различни области, како што се дизајн кориснички интерфејс (UI), имплементација на бизнис логиката итн.
- Архитектура треба да промовира реискористување на функционалностите (функционалните градивни компоненти/модули на апликациите), зголемување на квалитетот на апликациите со црпење на заеднички ресурси кои се достапни за сите тимови и да овозможува растечко интегрирање на нови функционалности на апликациите.
- Развиените функционалности да се достапни на поголем дијапазон на уреди, со можност за постоење на механизми за лесна размена на функционалностите не само помеѓу апликации, туку и помеѓу разните уреди.

2. Анализа на постоечки решенија и проблеми (Analysis of existing problems and solutions)

2.1. Анализа на процесот на развој на објектно-ориентирана софтверска апликација

Со цел полесно да ги воочиме проблемите со кои се судираат програмерите во процесот на развој на објектно ориентирани софтверски апликации ќе наведеме две имагинарни имплементации. Првата имплементација на објектно ориентирана апликација за пресметка на вредноста на портфолио на инвестициски фонд, а втората за управување со работењето на медицинска установа. Во првата пример имплементација под претпоставка е развиено софтверско решение апликација со која се пресметува вкупната вредност на портфолиото на еден инвестициски фонд на даден ден. Портфолиото на инвестицискиот фонд се состои од разни примарни инструменти, како обврзници, депозити, акции во компании, како и од изведени инструменти како swap, future итн. Дополнително секој од наведените типови на инструменти може да има некои специфични карактеристики во смисла на начин на исплата, стандардизација за пресметување на приносот и сл. Практично бесконечно е множеството од типови на финансиски инструменти и начини на пресметка на нивната вредност. Во моментот кога иницијално било развиено софтверското решение постоел еден сет од финансиски инструменти и начини на пресметка, но со текот на времето како таа апликација се користи секојдневно и неминовно се појавува потребата од додавање на нови функционални елементи на првичната апликација, со кои би биле опфатени и новопроизлезените финансиски инструменти и начини на пресметка. Ова е првиот проблем кој јасно може да се воочи. Вториот проблем при развојот и одржувањето на ваквата апликација е потребата од динамичко додавање на новите функционалности, не е продуктивно да се стопира и враќа од почеток вреднувањето на целото портфолио од финансиски инструменти само поради тоа што некој нов и неподдржан финансиски инструмент е лоциран во портфолиото. Втората анализа на решение за управување со работењето на медицинска установа каде што постои централна единица на која се чуваат сите податоци за работењето, кон која вработените во установата се поврзуваат преку различен тип на уреди (статични, преносни). Третиот проблем тука е веќе воочлив, а тоа е што различните типови на уреди имаат дијаметрално различни визуелни можности за приказ на податоците поврзани со работењето, примарно со пациентите, на преносен уред може да се прикаже визуелно поограничена форма на податоците, но не и содржински поограничена, сепак на медицинскиот персонал треба да им се достапни податоците во целост. Дополнително секој тип на медицински персонал нема исти потреби за пристап до податоци, ако на докторот кој треба да врши операција му треба целата историја на лекување на пациентот, моменталната дијагноза, подготвеност за операција, на докторот кој врши посета по соби и ја евидентира состојбата на пациентите преку преносен уред потребата му е сосема друга. Останатиот медицински персонал пр.

болничари и сестри би користеле преносен уред на кој би добиле приказ што треба да извршат за одреден пациент. Во однос на сите корисници, а делумно зависно од нивната возраст не се исти потребите за полесно користење на апликациите, во однос на големина на фонт кој се користи, јасност на приказите и сл. Оттука произлегува потребата да развивачите на софтвер развиваат решенија за секој различен тип на корисник индивидуално, а дополнително како проблем се јавува како би се управувал процесот на одржување на таквите апликации.

Проблемите наведени во двете пример апликации треба да бидат таргетирани од објектно-ориентираната архитектура која произлегува од овој труд и треба да овозможува лесна проширливост, надградливост, динамичка заменливост на делови од апликацијата (функционални и визуелни).

2.1.1. Слабо реискористување на елементарните функционалности

Во моментот може да се констатира дека реискористувањето на елементарните функционалности на апликациите е на многу ниско ниво, како и во рамки на една компанија, но уште подрастично на ниво на соработка и размена меѓу компаниите кои се занимаваат со развој на софтвер. Од друга страна, компаниите се соочуваат со проблемот да ја преработуваат цели апликации за имплементација на минорни промени во дел од функционалностите кога апликацијата се продава на нов купувач.

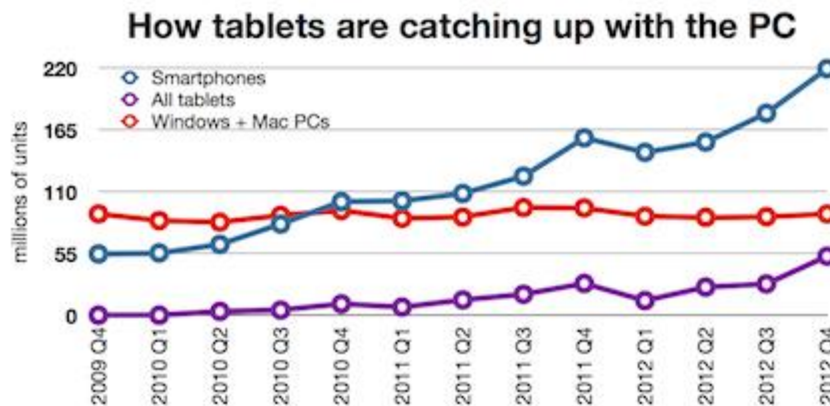
2.1.2. Квантификација на знаењето

Кај комплексни проекти каде што е потребно да се имплементира (квантифицира) соодветно знаење од некоја област најтешкиот дел од проектот е квантификацијата на знаењето. Решението е во доведување на проектите во множество минорни елементарни минипроекти, каде што знаењето лесно ќе биде екстрахирано, квантифицирано и имплементирано во иден софтверски „интелигентен“ инженерски систем.

2.2. Тенденции кај модерните електронски уреди

Модерните софтверски платформи (оперативни системи) како главна задача си го задаваат намалувањето на потрошувачката на енергија, а во исто време да обезбедат нормално функционирање на уредите, нормално време на одговор на акции (проценето на од 0.8 до 1.7 секунди).

Како што е прикажано на слика 1, статистички по години во 2010 година [1], 55% од продадените уреди се преносни, во 2011 година 69%, а во 2012 година 77%.



Слика 1. Приказ на продажба на Smartphone, песонални компјутери и таблети по квартали од четврти квартал 2009. Извор IDC [2]

Figure 1. Smartphone, PC and tablet shipments by quarter from 4Q 2009. Source: IDC [2]

Она што може да се забележи од статистичките податоци прикажани на слика 2 е значителниот пораст на продажба на Pad преносни компјутери од фантастични 275% и Smart phone уреди од околу 60% (2011 година во однос на 2010 година), но тоа не е воопшто на сметка на продажбата на персонални не преносни компјутери. Оттука сосема разбирлива тенденцијата кај развивачите на системски софтвер и производителите на електронските уреди да создаваат уреди кои на крајните корисници ќе им понудат поголем период на активно и пасивно користење.

Worldwide smart phone and client PC shipments Shipments and growth rates by category, Q4 2011 and full year 2011				
Category	Q4 2011 shipments (millions)	Growth Q4'11/Q4'10	Full year 2011 shipments (millions)	Growth 2011/2010
Smart phones	158.5	56.6%	487.7	62.7%
Total client PCs	120.2	16.3%	414.6	14.8%
- Pads	26.5	186.2%	63.2	274.2%
- Netbooks	6.7	-32.4%	29.4	-25.3%
- Notebooks	57.9	7.3%	209.6	7.5%
- Desktops	29.1	-3.6%	112.4	2.3%
Source: Canalys estimates © Canalys 2012				

Слика 2. Статистички податоци за продажба на Smart phone и персонални компјутери, 2010 година во однос на 2011 година [3]

Figure 2. Smart phone and client PC shipments, year 2010 in comparison with 2011 [3]

Производителите на електронски уреди се насочени кон понатамошен развој на архитектури за паралелно процесирање кои ќе обезбедат ниска потрошувачка на енергија, отпорност на електромагнетни појави (се користат оптички кола наместо електрични), тренд од 2004 година кој е директно финансиран од Министерството за одбрана на Соединетите Американски Држави [4], воената индустрија и компаниите кои развиваат средства за вонземска употреба [5]. Дополнителен проблем со кој се соочуваат производителите на електронски уреди е тоа што технологијата за производство на нови подобри материјали за производство на процесори заостанува, а во однос на батериите кои се користат кај преносните уреди, нема да постои замена за литиумот кој е најлесен тврд елемент за изработка на батерии. Секоја индивидуа има различни желби за изгледот на софтверската работна околина и типовите на уреди што ги користи, и секој заслужува да го добие тоа што го сака.

Слика 3. Samsung Brix Mobile

Figure 3. Samsung Brix Mobile

На слика 3 е прикажан модел на мобилен уред (телефон) произведен како прототип од страна на Samsung и патентиран дизајн под името Brix Mobile. Идејата која стои позади ваквиот дизајн е динамичко искористување на уредите со нивно поврзување во некој нова целина до цел да се добие подобар визуелен приказ или подобри перформанси при извршување на апликациите.

2.3. Постојни решенија

2.3.1. Вертикална модулација на проектите

Она што како модулација на проектите постои е модулација на кодот во свои елементарни функционални класи во кои се имплементирани соодветни методи за извршување на елементарната функционалност на тој код, но и методи за комуникација со останатите елементарни целини во рамки на проектот. Пример, апликација која би ја користеле матични доктори би имала неколку функционални елементи меѓу кои најава на системот, извршување на преглед и издавање терапија. Најчесто заради зачувување на интегритетот на апликациите повеќе класи на елементарни функционалности се групираат во компајлиран код што потоа се вчитува во некоја матична апликација. Но тука лежи и проблемот, поради тоа што програмерите се соочуваат со факторот на недостаток на време се избегнува дополнителната работа, што се однесува на имплементација на методи за комуникација помеѓу класите на елементарните функционалности и наместо тоа се изведуваат објекти од класите кои содржат елементарни функционалности со директно референцирање, а со тоа паѓа во вода концептот на модуларност односно елементарните функционалности веќе се групирани. Преку гореспоменатиот пример тоа би изгледало отприлика програмерот наместо во некоја класа *терапија* преку која се имплементира елементарната функционалност на апликацијата за препишување терапија да имплементира метода за пронаоѓање на кој е активниот доктор кој е најавен на апликацијата преку елементарната функционалност *најава* и кој е активниот пациент преку елементарната функционалност *преглед* директно референцира објекти кон објектите изведени од класите *најава* и *преглед* во рамки на проектот.

2.3.2. Хоризонтална модулација на проектите

Овој тип на модулација е доста популарен последните години благодарение на MVVM (Model View - View Model) шаблонот кој претставува сегрегација (одвојување) на функционалностите на апликација од изгледот на апликацијата и тоа како позадинските функционалности се користат. Model класите имплементираат сè што треба да може апликацијата да овозможува (комуникација со бази на податоци, имплементација на алгоритми и слично). View класите се однесуваат само на корисничкиот интерфејс, како се пристапува до функционалностите, изглед на компонентите, дефиниции на дополнителни анимации и сл. View Model класите служат за да се овозможи комуникација помеѓу корисничкиот интерфејс и функционалната поддршка во позадина.

И ова е добар пристап од гледна точка дека се врши одвојување на работните задачи за програмерите на апликацијата т.е. има јасна разлика и можност за специјализација во одредена област, пр. дизајн на кориснички интерфејси, дизајн на бази на податоци и со тоа е покриен дел од проблемите кои постоеја и постоја,т а се во насока на ажурно пренесување на информациите околу имплементираните функционалности на еден проект и документирањето на истите. Но со ваквиот пристап без соодветна вертикална

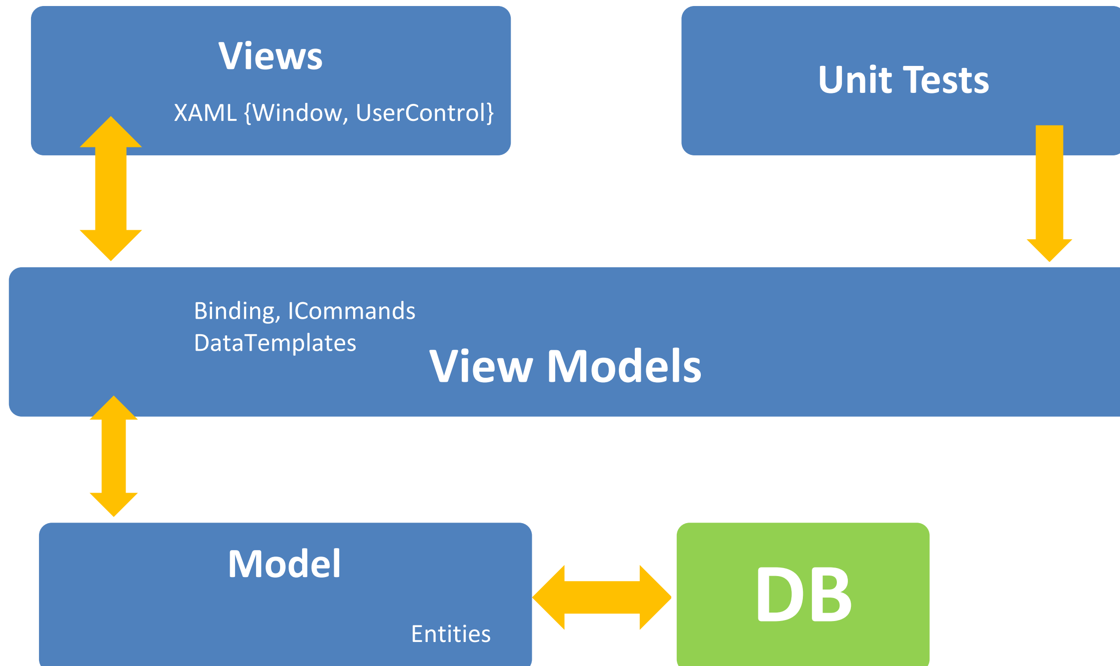
модулација е практично невозможно да се постигне реискористување на елементарните функционалности на апликацијата.

MVVM UI дизајн шаблонот е препорачан да се користи поради тоа што по природа одговара на Microsoft XAML платформата, WPF, Silverlight, Silverlight за Windows Phone 7/8, Windows 8. Овие платформи, кои инаку користат големи податочни извори и влечат мноштво поврзани својства, овој шаблон овозможува view и view Model да комуницираат меѓусебно со лабава врска.

Одделувањето на логиката од View е важно за да се овозможи проектите лесно да се тестираат и одржуваат, а го подобрува и текот на развојот на проектот во смисла на интеракција помеѓу програмерот и дизајнерот. Ако се создаде view со некоја корисничка компонента (контрола) и цел код за нејзината функционалност се смести во рамки на кодот од самото view, ќе се отежни тестирањето на истата поради тоа што ќе треба да се создаде инстанца од View за да се тестира кодот на функционалноста. Ова е особено проблем ако View се изведува или зависи од некоја WPF или Silverlight компонента како дел од нејзиниот контекст на извршување. За да сме сигурни дека unit тестовите на кодот кој ја носи функционалноста на View е во изолација од овие зависности мора да истото е во состојба да меѓу зависностите на контекстот на извршување бидат одделени во посебни класи едни за изгледот на View други за функционалностите кои истото ќе ги повикува. Ако дефинираме view за приказ на некои податоци тогаш треба да нема никаков код во рамки на него којшто ќе врши некаква функција во однос на тие податоци. Затоа таа логика за работа со податоците треба да биде имплементирана настрана. Ваквиот начин на чисто одделување на логиката на функционалностите од изгледот и приказ е потребна за да се овозможи проектите (view) полесно да се тестираат и одржуваат.

2.3.3. Поглед на MVVM шаблонот

Програмерите кои работат со Silverlight или WPF секојдневно во склоп на работата го користат MVVM (Model - View - ViewModel) шаблонот. Тоа е најпопуларната client - side шаблон архитектура која ја користат Silverlight/WPF програмерите. Но се чини дека во тој шаблон постојат некои големи проблеми кои не се добро дефинирани и решени.



Слика 4. Опис на MVVM

Figure 4. Description of MVVM

MVVM навидум го елиминира целиот код во позадина кој нормално би бил сместен во кодот на View (корисничка контрола, компонента или слично) со преместување на овој код во ViewModel. Самиот ViewModel не е никогаш цврсто врзан (спарен) со View. Ако воопшто референцира до него најчесто тоа е преку интерфејс кој го имплементира View наместо со директно референцирање на вистинскиот тип на View. Ова овозможува поголемиот дел од UI кодот (корисничкиот интерфејс) да може да биде тестиран преку unit тестови, нешто што не е можно со традиционалниот начин на програмирање каде што сите функционалности се имплементирани во кодот кој е во позадина на View.

ViewModel најчесто содржи својства за податоците кои треба да бидат прикажани на View и исто така покренува настани со известувања (notification events) кога податоците во тие својства (properties) ќе бидат променети. View ги користи можностите на Silverlight/WPF за податочно поврзување за да ги врзе својствата на ViewModel со

останатите кориснички контроли од кои се состои View. Настаните кои ги покренува корисникот се прифаќаат од View и се праќаат до ViewModel преку Команди (Commands). Најчесто овие команди извршуваат методи кои се наоѓаат во ViewModel, методи кои содржат програмска логика за да променат и/или преземат податоци од својствата на ViewModel кои се однесуваат на Model (најчесто сервисниот слој во Silverlight, во WPF може да биде Domain Model или исто Service Layer). Методите може да извршуваат и некоја бизнис логика во рамки на Model или да преземат податоци од Modelт кои ќе бидат сместени во својствата на ViewModel на кои View-то потоа би се врзало и би ги прикажало.

Накратко, вака би опишале како MVVM шаблонот функционира и тука веќе се поставуваат прашањата, каде што се неговите слабости. Најважно како придобивка ќе анализираме дали тоа што може да се тестира логиката на апликацијата т.е. кодот засебно од физичкиот приказ е добро или не? Секако дека е добро, позитивно и олеснително, но главниот недостаток на овој шаблон е тоа што премногу обврски се концентрирани во ViewModel [6]. Тој е доминантно одговорен за податочното поврзување и интеракцијата со Model и се добива впечаток дека, всушност, истиот е само поинаква верзија на традиционалниот начин на програмирање преку код во позадина, со една единствена разлика што истиот не е цврсто поврзан и индиректно референцира на View (физичкиот приказ).

Во повеќето (ако не и во сите) имплементации на MVVM, за ViewModel постојат многу потенцијални причини да истиот стане предмет на промена. Може да се појави потреба од менување поради промена во начинот на податочно поврзување или можеби кога се менува дел од Model.

Од аспект на ViewModel може да се подвлече фактот дека истиот нема соодветна компатибилност со двата најважни принципи кои се воспоставени во програмерскиот свет:

- Начелото за единствена одговорност (Single responsibility principle);
- Раслојувањето на одговорностите за одредени типови задачи (Separation of concerns).

Ако за момент го тргнеме MVVM на страна и се фокусираме на задачите кои треба да ги исполнува една корисничка контрола, пр. корисничка контрола која прикажува кориснички информации и му овозможува на корисникот да врши измени врз истите притоа овозможувајќи потврда на промените, истата е одговорна за:

- визуализација на корисничката контрола (свој распоред, како и цртање други кориснички контроли од кои таа е составена);
- комуникација и интеракција со Model;
- да овозможи податоците од Model да бидат достапни за прикажување;

- да ги прикаже податоците во соодветните кориснички контроли (пр. повеќе текст полиња);
- едноставна валидација на внесените и модифицираните податоци.

Без користење на MVVM сите овие активности би биле реализирани од View и тоа е класичен пример за користење на пристапот за програмирање со код во позадина на View.

Со користење MVVM овие одговорности и задачи се поделени, имено во View:

- визуализација на самата корисничка контрола (свој распоред, како и цртање други кориснички контроли од кои таа е составена);
- да ги прикаже податоците во соодветните кориснички контроли (пр. повеќе текст полиња);
- едноставна валидација на внесените и модифицираните податоци;

во ViewModel:

- комуникација и интеракција со Model;
- да овозможи податоците од Model да бидат достапни за прикажување.

Во овој случај, после ваквата поделба на View му преостануваат три обврски што и не е многу ценејќи по нивниот обем (а постојат и настојувања да валидацијата, сепак биде преместена во ViewModel).

Не е баш вообичаено да се пишуваат тестови за тестирање на визуализацијата (корисничкиот интерфејс) и што се однесува до начелото за единствена одговорност во однос на View не е голем пропуст ако тоа не се прави и истото се зема за прифатливо.

Сепак ViewModel има две важни задачи во овој случај, но расправа постои за тоа дали воопшто овие две задачи т.е. одговорности треба да бидат сместени заедно. За да се овозможи податоците од Model да бидат достапни за прикажување, треба да се подигнат низа од настани (events) и во најголем дел од случаите нивното повикување е многу јасно и еднонасочно, но кај покомплексните контроли може да се јави потреба од дополнителна логика која ќе определува условено кој настан кога да биде подигнат.

Другата важна задача е комуникацијата и интеракцијата со Model и во повеќето Silverlight апликации Modelт ќе биде сервисниот слој (Service Layer). За да комуницира со Service Layer се потребни пак Service Proxies. Тоа значи дека ViewModel во суштина е одговорен за комуникација со Service Layer, се справува со исклучоци во бизнис логиката подигнати поради некои сервисни повици, а во исто време се справува и со технички исклучоци кои може едноставно да се случат поради мрежни проблеми. Кога ќе ги групираме сите горespoменати и не е баш така логично сите овие задачи да се доверат и поставаат во една класа.

Може да се забележи појавата во секојдневното работење каде што програмерите енергично се залагаат за придржување до начелото за „раслојување на одговорностите за одредени типови задачи“ во бизнис логиката но не се придржуваат до тоа начело кога се однесува на корисничкиот интерфејс.

Во пракса се користат и други алтернативни пристапи т.е. шаблони како пример може да се наведе MVP шаблонот на кој е применета една модификација. Модификацијата се состои со примена на начела од Presentation Model чијашто задача е да го олесни податочното поврзување а во некои случаи и олеснета валидација на податоците. Овој Шаблон се нарекува MVP - PM light.

Ако се навратиме на претходно споменатиот пример со корисничката контрола, нејзините одговорности и задачи, и ако се примени MVP - PM light пристапот тогаш поделбата би изгледала вака:

Во View:

- визуализација на корисничката контрола (свој распоред, како и цртање други кориснички контроли од кои таа е составена);
- да ги прикаже податоците во соодветните кориснички контроли (пр. повеќе текст полиња);
- едноставна валидација на внесените и модифицираните податоци.

Во Presenter:

- комуникација и интеракција со Model која се потпира на содржината од Presentation Model.

Во Presentation Model:

- да овозможи податоците од Model да бидат достапни за прикажување.

Овој начин на поделба овозможува фокусот на класите да е на кон нивните задачи наместо да се грижат за мноштво разнородни работи. Овој пристап изгледа е многу подобар, почист и поприменлив од MVVM. Не само што истиот носи со себе забележлива придобивка во квалитетот на кодот, туку и потенцијалот за реискористување на класите е зголемен. Исто така е зголемена ефикасноста при тестирање споредено со MVVM, бидејќи секогаш е полесно да се тестираат класи кои се фокусирани наспроти да се тестираат класи кои имаат премногу обврски.

Сумарно, MVP - PM light има неколку важни предности наспроти MVVM, а недостатоци при оваа споредба нема и секако предноста со користење на MVP - PM light е олеснетото тестирање.

2.3.4. Начело на единствена одговорност /Single responsibility principle

Во објектно - ориентираното програмирање начелото за единствена одговорност вели дека секој објект треба да има само една и единствена задача (одговорност) и дека таа задача треба да биде целосно имплементирана од страна на класата. Сите сервиси треба да бидат тесно усогласени со таа задача.

Терминот за првпат е употребен од Robert C. Martin во истоимената статија како дел од неговиот труд „Принципи на објектно-ориентиран дизајн“ (Principles of Object Oriented Design), а бил популаризиран преку неговата книга „Агилен развој на софтвер, принципи, модели и практики“ (Agile Software Development, Principles, Шаблонs, and Practices [7]). Мартин ова начело го базира врз принципот на кохезија, како што е опишано од Tom DeMarco во неговата книга „Структурни анализи и системски спецификации“ (Structured Analysis and Systems Specification [8]).

Имено, тој го дефинира начелото за единствана одговорност преку одговорноста (задачата) и евентуалната причина за нејзина промена и заклучува дека за една класа или модул треба да има една и само една причина за промена.

Како пример може да се наведе еден модул чија задача е да компајлира и печати некој извештај. За таквиот модул постојат две причини за негова евентуална промена, првата ако содржината на извештајот треба да се промени и втората ако форматот и изгледот на самиот извештај треба да се променат. Евентуалните причини за промена се од различна природа, едната е суштинска а другата би рекол козметичка. Начелото за единствена одговорност вели дека овие два аспекта на проблемот кој е опфатен со функционалноста на посочениот модул се, всушност, две различни одговорности (задачи) и затоа тие треба да бидат разделени во две засебни класи т.е. во случајов модули. Според начелото би било голема грешка при дизајн ако се спојат две разнородни функционалности за кои во иднина би можело да се појави потреба од промена и тоа во различен период.

Причината поради која е важно да класата остане фокусирана само на една задача т.е. да и се довери само една и единствена одговорност е тоа што тоа ја прави да е поробусна односно постабилна. Во продолжение на горенаведениот пример, ако постои промена на извештајот во процесот на компајлирање ризикот да функцијата на кодот кој е одговорен за печатење на извештајот се наруши е поголем ако тие две функционалности се наоѓаат заедно во истата класа.

Начелото за единствена одговорност се користи во методологии за дизајн водени спрема одговорностите (responsibility driven design methodologies), како на пример Responsibility - driven design (RDD) и Use Case / Responsibility Driven Analysis and Design (URDAD).

2.3.5. Раслојување на одговорностите за одредени типови задачи

Separation of concerns

По дефиниција, *раслојувањето на одговорностите за одредени типови задачи* претставува процес на раслојување (поделба) на компјутерската програма (нејзиниот функционален код) на мноштво елементарни функционални делови чијапшто функционалност што е можно помалку се преклопува.

Одговорноста (concern) се дефинира како точно одреден фокус или точно определен дел од компјутерската програма (функционалниот код). Во основа одговорноста се употребува како синоним и за карактеристиките и однесувањето на функционалноста.

Напредок во примена на раслојувањето на одговорностите за одредени типови задачи вообичаено се постигнува преку модулација при програмирањето и енкапсулација (нетранспарентност на операциите) со помош на прикривање на информациите во нив.

Повеќеслојните дизајни во информационите системи многу често се базираат врз раслојувањето на одговорностите спрема одредени типови задачи и така се дефинираат функционални слоеви (пр. презентациско ниво, бизнис ниво, ниво за пристап до податоци, ниво на база на податоци и сл.)

Интернет мрежата е најпознатиот пример е еден таков информационен систем.

Кај повеќето шаблони кои се користат за развој на софтвер постои тенденција да се постигне и подобри раслојувањето на одговорностите за одредени типови задачи што секако им е од помош на програмерите. Како пример во објектно-ориентираните програмски јазици како Delphi, C++, Java и C# постои раслојување (поделба) на одговорностите во објекти, а моделите за дизајн како MVC нудат одделување на содржината на презентациски дел и дел кој се грижи за процесирањето на податоците односно моделот. Кај сервисно - ориентираните дизајни секоја од одговорностите е одделена во посебни сервиси. Кај процедуралните програмски јазици како C и Pascal засебните одговорности се одделени во процедури. Како аспект - ориентираните програмски јазици секоја одговорност е одделена во аспекти и објекти. Прикажано во табела 1.

Програмска архитектура	Тип на поделба
Објектно - ориентирани	Објекти
MVC	Презентациски, податочни
Сервисно - ориентирани	Сервиси
Процедурални програмски јазици	Процедури
Аспект - ориентирани	Објекти, аспекти

Табела 1. Типови на поделба во однос на архитектурата

Table 1. Types of separation in terms of architecture

Раслојувањето на одговорностите за одредени типови задачи е многу важно начело за дизајн и во други области, како урбаното планирање, архитектурата, а секако и во информатиката. Целта е да се дизајнираат системи чиишто функции можат да бидат оптимизирани независно од останатите функции на системот кој е предмет на дизајн, така што падот на една функција нема да може да предизвика пад и неправилно однесување на останатите функции, а генерално тоа доведува да системот е полесен за разбирање, дизајнирање, а секако е полесно да се одржува множество од независни системи наспроти еден голем комплексен систем. Споредбен пример од секојдневниот живот може да најдеме во внатрешниот дизајн на еден дом. Истиот може да е дизајниран така што целиот простор е распределен во соби меѓусебно поврзани со врати и домот да има еден осигурувач за проток на електрична енергија на кој ќе е поврзано сѐ, од шпорет, па до осветлување или просторот може да биде дизајниран да постои ходник помеѓу собите и секој електричен уред да има свој засебен осигурувач за проток на електрична енергија.

Терминот „Раслојување на одговорностите за одредени типови задачи“ е воведен од Edsger W. Dijkstra во 1974 година во неговата статија „Во улога на научната мисла“ [9].

„Дозволете ми да ви објаснам, што според мене е карактеристика на интелигентното размислување. Тоа е кога некој има желба и е подготвен да учи и истражува во длабина еден единствен аспект на некој предмет во изолација заради одржување на сопствената конзистентност и цело време имајќи на ум дека се занимава само со еден од аспектите. Знаеме дека за една компјутерска програма есенцијално е таа да ја исполнува бизнис логика заради која е создадена и сме насочени само кон исполнување на таа цел. Но, исто така, знаеме дека таа треба да биде ефикасна и наредно ја проучуваме од тој аспект. Ако погледнеме на целата слика се поставува прашањето дали овој начин на пристап кон проблематиката е пожелен и што ако одговорот е да. Во случај да се обидеме да ги изведеме овие два разнородни аспекти на компјутерската програма симултано многу е веројатно дека ќе се

изгубиме некаде на средина. Ова е тоа што се нарекува раслојување на одговорностите за одредени типови задачи што во реалност никогаш не е можно да се изведе перфектно но е единствената техника за ефективно подредување на размислувањето, која јас ја познавам. Тоа е она што го мислам кога велам „да се фокусира вниманието кон еден единствен аспект“, а тоа не значи дека истовремено ги игнорирам сите останати аспекти, туку дека во тој момент на размислување важно е она што се однесува на дадениот аспект, а останатите аспекти се ирелевантни. Се сведува на умствена постапка во која или размислувате на едно нешто или се трудите симултано да пратите повеќе работи“.

Петнаесет години подоцна терминот *раслојувањето на одговорностите за одредени типови задачи* станува сè поприфатена идеја. Во 1989 година Chris Reade во својата книга „Елементи на функционалното програмирање“ [10] терминот и начелото ги опишува вака:

Програмерот има за задача да прави неколку работи истовремено и тоа:

1. *Да опише што е тоа што треба да се процесира;*
2. *Да ја организира програмска секвенца во мали чекори;*
3. *Да го организира раководењето со меморијата во процесот.*

Reade понатаму вели:

„Во идеален случај, програмерот треба да е во состојба да може да се концентрира на првата од наведените три задачи (да опише што е тоа што треба да се процесира) без притоа да вниманието му го одземаат другите две, т.н. административни задачи. Секако, тие административни задачи се, исто така, важни но важно е тие да бидат одделени од примарната задача, затоа што така ќе се добијат поверодостојни резултати и процесот на програмирање ќе биде олеснет преку воведување на автоматизација во административните задачи. Раслојувањето на одговорностите за одредени типови на задачи, исто така, има и други предности. Како, на пример, докажувањето на функционалноста на програмата е многу повеќе изводливо кога организирањето на програмските секвенци и раководењето со меморијата се надвор од програмата. Понатаму, во однос на споменатово описите за тоа што треба да биде процесирано/пресметано треба да бидат без наведени детални чекор по чекор инструкции, како тоа треба да се направи ако притоа програмата треба да функционира на различни машински архитектури. За да се изведе некоја минорна промена во некоја податочна структура ако се дефинираат и најмалите секвенци од инструкции како да се изведе тоа може да се случи кога таа програма че биде извршувана од машина која користи паралелизам и има стотици процесори практично залудно да ги користи своите ресурси, а некоја персонална машина истата таа програмска секвенца да ја процесира и за пократко време. Автоматизирањето на административните аспекти значи дека соодветниот јазичен имплементор треба да се грижи за нив и истовремено на истиот му е препуштена одговорноста да избере соодветен механизам за извршување на процесирањето во однос на машинската архитектура на која се однесува.

Пример имплементации

Internet

Раслојувањето на одговорностите за одредени типови на задачи има клучна улога во дизајнот на интернет. Тој дизајн (пакет на протоколи) е многу добар пример како може и треба да се изврши поделба на одговорностите за одредени типови на задачи во различни добро дефинирани слоеви. Истиот им овозможува да дизајнерите на нови протоколи секогаш се фокусираат на еден единствен слој, притоа може да го игнорираат постоењето на останатите слоеви. Така, на пример, протоколот за размена на email пораки SMTP кој е протокол од апликацискиот слој (ниво) е насочен само кон одговорноста и сите детали за доверливо спроведување на една email порака преку транспортниот сервис (најчесто TCP), притоа воопшто не водејќи грижа како тоа транспортниот сервис ја спроведува таа услуга. Слично на тоа, TCP воопшто не се грижи за тоа како се одвива рутирањето (насочувањето) на податочните пакети, кое е одговорност на интернет нивото (слојот).

HyperText Markup Language (HTML) и cascading style sheets (CSS)

И двата се јазици направени со цел да се одвои содржината и изгледот на една веб-страница. Онаму каде што HTML елементите ја дефинираат апстрактната структура и содржина на документот, CSS директивите се повикуваат за да веб-пребарувачот знае како да ги прикаже истите тие елементи во визуелна форма. Истото се постигнува и со користење на XML и XSLT за трансформација на податоците од XML.

2.3.6. Модуларен дизајн

При дизајнирање на некој синтетички систем (како електрични машини, биолошки ензими или проектирање на згради), системот може да биде дизајниран на два начина. Првиот начин е да се дизајнира целиот систем со користење на познати теории и употреби односно тестира системот за она што е наменет во реални услови. Алтернативен начин би бил да се изградат сите компоненти на системот одделно и да се тестира секоја компонента во изолирани услови. Модуларниот дизајн или „модулирање при дизајн" е пристап преку кој системот се дели на помали делови (модули) коишто може да се создадат независно и потоа можат да бидат користени и во други системи исполнувајќи ја истата функционалност за која се создадени. Еден систем кој е модулиран се карактеризира со:

1. Функционална поделба во дискретни множества од модули кои се состојат од изолирани и автономни функционални елементи;
2. Исклучителна употреба на добро дефиниран модуларен интерфејс, кој вклучува објектно-ориентиран опис на функционалноста на модулите;
3. Транспарентноста на употребена татехнологија, постапките и процедурите кои таа ги користи а со тоа едноставна менливост и проширливост на нејзините функционалности.

Покрај намалувањето на трошоците поради помалку потребни приспособувања и флексибилноста во дизајнот, модуларноста, нуди и други придобивки, како што се надградување (додавање на нови решенија со само приклучување на нов модул), а исто така и исклучување на одредени функционалности на производот зависно од економската оправданост (барања на пазарот). [11] Најдобар пример за системи при чие производство / склопување се почитува модулирањето при дизајн се автомобилите, компјутерите, системи за дистрибуција на електрична енергија и сл. Во компјутерската индустрија се користи модулирањето за лесно да се надминуваат новите и променливи барања на потрошувачите (пазарот) и да процесот на производство биде што е можно поадаптивен на промените. Модулирањето при дизајн е обид да се комбинираат предностите на стандардизацијата (зголемен обем на производство значи намалени трошоци за производство по единица производ) со барањата за приспособување на производот спрема одредени барања на потрошувачот. Недостаток на модулирањето е тоа што модулираните системи не се оптимизирани во перформанси. Тоа е најчесто поради употребата на интерфејси за комуникација помеѓу модулите.

2.3.7. Модуларно програмирање

Модуларното програмирање е техника за дизајн на софтвер преку која се остварува софтверот да се состои од одделни, заменливи компоненти наречени модули, со разделување на функционалностите на програмата во модули и од кои секој остварува една единствена функција и содржи сè што е потребно истата да се оствари. Концептуално модулите претставуваат разделување на одговорностите за одредени типови задачи и го подобруваат степенот на одржливост преку јасно означување на логичките границите помеѓу компонентите.

„Модуларното програмирање претставува стил на програмирање којшто ги разбива функционалностите на програмата во модули, при што секој модул имплементира само една единствена функција и во себе го содржи целиот код и променливи кои му се потребни за да ја извршува функцијата која му е доделена. Модуларното програмирање е решение на проблемот кој се јавува кај големите и комплексни компјутерски програми кои инаку се тешки за одржување, дебагирање и тестирање. [12] *Преку сегментирањето на програмата во модули коишто извршуваат точно одредени задачи, може лесно да се одреди изворот на некоја програмска грешка или недоследност со дефинираната бизнис логика на програмата. Некои објектно-ориентирани јазици како SmallTalk и HyperTalk, во себе ги имаат инкорпорирано принципите за модуларно програмирање".*

Модулите обично се вклучуваат во програмите преку интерфејси. Интерфејсот на модулот ги овозможува елементите кои се достапни и потребни за тој модул. Елементите кои се дефинирани во интерфејсот на модулот можат да бидат достапни до останатите модули. Имплементацијата го содржи програмскиот код кој одговара на даден елемент кој е дефиниран во интерфејсот на модулот.

Во Mozart Programming System развојната платформа за интелигентни, дистрибуирани апликации [13], модулите се дефинирани на следниот начин:

„Модулите, исто така, познати и како пакети, претставуваат збир на процедури и променливи кои се споени заедно да овозможат извршување на точно одредени сродни функции. Модулот најчесто има одреден број на процедури кои не се достапни надвор од него и, исто така, одреден број на процедури кои се достапни во неговиот интерфејс, преку коишто модулот ги овозможува надворешните сервиси за кои постои".

Основен метод на имплементација - Message passing / пренесување на пораки

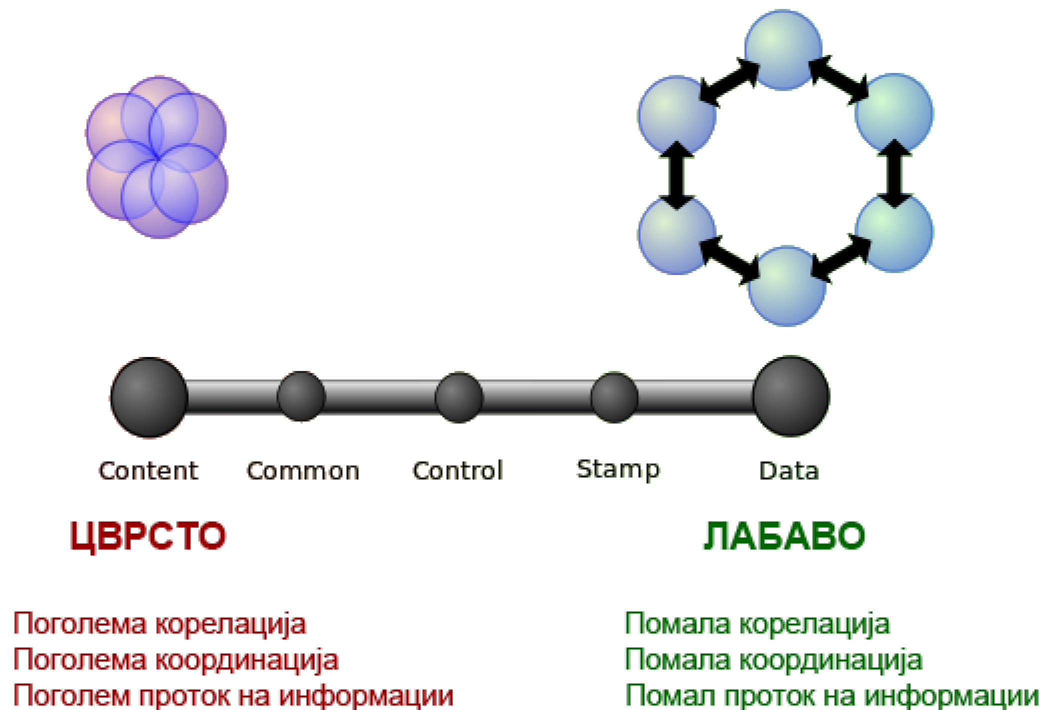
Методот на пренесување на пораки од неодамна добиено има поголемо значење во однос на претходните, повеќе конвенционални "Call" интерфејси, и со тоа станува доминантен начин за поврзување на различни модули, а воедно претставува и обид за решавање на проблемот на постоење на различни верзии на модулите (versioning problem) - понекогаш се јавува кога се користат интерфејси за комуникација помеѓу модулите.

Предности на Модуларното програмирање

Неколку програмери можат да работат на индивидуални програми (модули) во исто време, на тој начин развивајќи ја компјутерската програма побрзо. Кодот разделен на помали функционални целини е полесно да се *дебугира,* тестира, ажурира и менува. Тоа доведува до структуриран приод за решавање комплексни проблеми кои можат да биде поделени на поедноставни задачи.

2.3.8. Спарување (врзување) / Coupling

Спарувањето најчесто е во контраст со кохезијата (поврзаност, привлечност што владее во група). Ниското ниво на спарување е во корелација со високо ниво на кохезија и обратно. Метриката за одредување на квалитетот на софтверот во однос на спарувањето и кохезијата е воведена од Larry Constantine, еден од авторите на „Структуриран дизајн“ [14]. Спарувањето според него е знак за постоење на добро структуриран компјутерски систем и добар дизајн и во комбинација со високото ниво на кохезија овозможува постигнување на главните цели високо ниво на разбирливост и можност за лесно одржување.



Слика 5. Типови на спарување

Figure 5. Types of coupling

Спарувањето може да биде на ниво „ниско“ "low" (или „лабаво“ "loose" и „слабо “"weak") и се до „високо“ "high" (или „цврсто“ "tight" и „јако“ "strong"), како што е прикажано на слика 5.

Типовите на спарување прикажани во редослед од највисоко до најниско ниво се следниве:

Содржинско спарување (цврсто спарување) / Content coupling

Содржинско спарување е случај кога еден модул го модифицира или се потпира врз внатрешното работење на некој друг модул (пр. пристапува до локални податоци од друг модул). Оттука, промените во однос на тоа како вториот модул ги продуцира податоците ќе доведе до потреба да се промени и зависниот модул.

Вообичаено спарување / Common coupling

Вообичаеното спарување (познато и како глобално спарување) е кога два модула делат заеднички глобални податоци. Така, промените во заедничкиот податочен ресурс имплицираат и промени во модулите кои го користат.

Надворешно спарување / External coupling

Надворешно спарување е кога два модула делат надворешно пропишан податочен формат, комуникациски протокол или интерфејс на некој уред. Ова е во основа е поврзано со комуникација со надворешни алатки и уреди.

Спарување со пренесување на контролата / Control coupling

Спарување со пренесување на контролата настанува кога еден модул ја контролира работата на некој друг, со тоа што му пренесува информации што другиот да прави (пр. предавање на what-to-do flag).

Податочно структурирано спарување / Stamp coupling (Data-structured coupling)

Податочно структурираното спарување е кога модулите делат композитна податочна структура, а притоа користат само дел од неа, и тоа по можност различен дел. (пр. предавање на цел ред на некоја функција која користи само еден запис од предадениот ред). Ова може да доведе до промени на начинот на кој модулот го чита редот поради промена на некое поле кое на модулот воопшто не му е потребно.

Податочно спарување / Data coupling

Податочното спарување е кога модулите делат податоци преку некои одредени параметри. Секој податок е основно парче, и тоа е единствениот податок кој се дели. (пр. предавање на цел број на функција која пресметува квадратен корен).

Спарување со предавање на пораки (лабаво спарување) / Message coupling (low)

Ова е најлабавиот начин на спарување на модули. Се постигнува со комплетна раздвоеност на модулите кои комуницираат со тоа што само си предаваат пораки, на кои што може но и не мора да одговорат со акција.

Раздвоени / No coupling

Кога модулите воопшто не комуницираат меѓусебно.

Типови на спарувања во објектно ориентираното програмирање

Поткласно спарување / Subclass Coupling

Се опишува преку односот помеѓу дете и неговиот родител. Детето е поврзан со својот родител, но родителот не е поврзан со детето.

Привремено спарување / Temporal coupling

Кога две акции се во комплет заедно во еден модул само затоа се случува тие да се појават во исто време.

Недостатоци

Цврсто споените (спарените) системи ги покажуваат следните карактеристики, кои често се гледаат како недостатоци:

1. Промена во еден модул најчесто иницира домино ефект на промени во други модули, со тенденција ефектот од почетната состојба да се зголемува;
2. За создавање на збир на модули (assembly) треба да се вложи повеќе напор и време со оглед на зголеменото ниво на меѓусебните зависности на модулите кои го сочинуваат тој збир;
3. Одреден модул може да биде потешко да се употреби повторно и / или тестира, бидејќи зависните модули мора да бидат вклучени.

Прашања поврзани со перформансите / Performance issues

Без разлика дали се лабаво или цврсто спарени модули, ефикасноста на системот изграден од модули е главно намалена заради употребата методот на проследување на пораки и параметри (потребно нивно создавање, пренесување, преведување и интерпретирање).

Намалувањето на перформансите заради зголемена употреба на ресурси се должи на процесите:

Создавање на пораките / Message Creation Overhead and Performance

Со оглед на тоа што сите пораки мора да поседуваат соодветно значење за да бидат употребувани т.е. да значат нешто за логичниот тек кај примачот на пораката, тие мора да бидат создадени со одредено значење. Создавањето на каква било порака бара зголемена употреба на ресурсите (процесорско време или меморија). Создавање на една вредност како цел број, некој стринг, низа или точно одредена податочна структура е секако полесна операција од создавањето на комплицирана податочна структура како SOAP порака. Создавањето на подолги пораки бара повеќе процесорско време и меморија. *За да се оптимизираат перформансите при извршување, должината на*

самите пораки мора да се минимизира и а при тоа значењето на пораката да се максимизира.

Пренесување на пораките / Message Transmission Overhead and Performance

Поради тоа пораката мора да биде пренесена во целост за да го задржи своето значење и процесот на пренос на пораката мора да биде оптимизиран. Испраќањето и примањето на подолги пораки бара повеќе процесорско време и меморија. Исто така, кога е потребно примачот мора да ја состави пораката во оригиналната состојба за нејзиниот прием да се смета за успешен. Оттука, исто така, произлегува дека за да се оптимизираат перформансите при извршување, должината на пораките мора да се минимизира и притоа значењето на пораката да се максимизира.

Преведување на пораките / Message Translation Overhead and Performance

Протоколите за пренос на пораки, но и самите пораки, понекогаш може да содржат екстра информации. Затоа примачот често треба да ги преведе пораките во порафинирана форма преку отстранување на непотребните информации, конверзија на вредности од еден формат во друг и сл. Секоја форма на преведување на пораката доведува до зголемување на времето за процесирање и потребата од работна меморија. За да се оптимизираат перформансите при извршување, формата на пораките и нејзината содржина мора да се редуцирани и рафинирани со цел да се максимизира нејзиното значење и да се редуцира процесот на преведување.

Интерпретација на пораките / Message Interpretation Overhead and Performance

Сите пораки мора да бидат интерпретирани кај примачот. За едноставни пораки како цели броеви може и да нема потреба од дополнително процесирање за истите да се интерпретираат. Сепак, сложените пораки како што се SOAP пораките бараат соодветен парсер и серија трансформации за од нив да се изложат планираните значења. За да се оптимизираат перформансите при извршување, пораките мора да бидат рафинирани и редуцирани за да се сведе на минимум зголемената употреба на ресурси.

Решенија за намалување на меѓу зависностите при спарување

Еден пристап за намалување на меѓу зависностите при спарувањето е преку користење на функционален дизајн, кој што тежнее да ги ограничи одговорностите (задачите) на модулите во текот на функционалноста. Стандарден начин за да се обезбеди функционален дизајн е да се разгледа описот на предметот. Ако описот вклучува сврзници, како што се „и" или „или", а тогаш тој дизајн има повеќе од една одговорност. Одговорностите треба да бидат поделени во неколку модули со цел да се постигне функционален дизајн. [15]

Меѓу зависноста при спарување помеѓу две класи **А** и **Б** се зголемува кога:

А има атрибут кој рефенцира кон (е од тип) **Б;**

А повикува сервис од објектот **Б;**

А има метод кој референцира кон **Б** (преку параметар);

А е поткласа (или ја имплементира) класата **Б.**

Лабавото спарување реферира кон тип на релација во која еден модул е во интеракција со друг модул преку едноставен и стабилен интерфејс и нема потреба да се грижи за интерната логика и операции на другиот модул.

Системи како CORBA и COM дозволуваат објектите да комуницираат меѓусебно без притоа да знаат што било за имплементацијата. И двата наведени системи дури дозволуваат објектите да комуницираат со други објекти напишани во друг јазик.

Спарување наспроти кохезија

Спарување и кохезија се два термини кои многу често се наоѓаат заедно. Заедно тие ни зборуваат за квалитетот кој одреден модул го поседува. Спарувањето ни кажува за меѓузависностите помеѓу разновидни модули додека кохезијата ни опишува колку поврзани функции истите содржат. Ниското ниво на кохезија имплицира дека модулот извршува задачи кои не се многу сродни и оттука тоа може да создаде проблеми како што модулот се проширува.

Концепт за спарување на модули

Спарувањето во софтверското инженерство е опишано преку соодветна метрика која асоцира со овој концепт.

Параметри според кои се одредува податочното спарување и спарувањето со контрола на протокот се:

$\mathbf{d_i}$: број на влезни податочни параметри;

$\mathbf{c_i}$: број на влезни контролни параметри;

$\mathbf{d_o}$: број на излезни податочни параметри;

$\mathbf{c_o}$: број на излезни контролни параметри.

За глобално спарување:

$\mathbf{g_d}$: број на глобални променливи употребени како податоци;

$\mathbf{g_c}$: број на глобални променливи употребени како контроли.

За спарување во однос на опкружувањето:

w: број на повикани модули;

r: број на модули кои повикуваат други модули.

$$\text{Спарување } (C) = 1 - \frac{1}{d_i + 2 \times c_i + \ d_0 + 2 \times c_0 + \ g_d + 2 \times g_c + \ \omega + r}$$

Равенка 1. Одредување на ниво на спарување

Equation 1. Determination of the level of coupling

Спарувањето (C) добива поголема вредност ако спарувањето е поголемо. Овој број е во опсегот од приближно 0.67 (лабаво спарување) до 1.0 (цврсто спарување). [16]

На пример, ако модулот има само еден влезен и еден излезен податочен параметар спарувањето ќе е лабаво со вредност од 0,67.

$$\text{Спарување } (C) = 1 - \frac{1}{1 + 0 + \ 1 + 0 + \ 0 + 0 + \ 1 + 0} = 1 - \frac{1}{3} = 0{,}67$$

Ако модулот има пет влезни и излезни параметри, и исто толку контролни параметри, и пристапува до 10 глобални податочни структури, и користи 3 повикани модули и 4 модули кои повикуваат други модули, тогаш спарувањето изнесува 0,98 и е цврсто.

$$\text{Спарување } (C) = 1 - \frac{1}{5 + 2 \times 5 + \ 5 + 2 \times 5 + \ 10 + 0 + \ 3 + 4} = 0{,}98$$

2.3.9. Попречни задачи (одговорности) / Cross - cutting concerns

Иако повеќе класи во објектно-ориентираниот модел извршуваат една специфична функција, тие често делат некоја обична секундарна функција заедно со други класи. На пример, ако има потреба да се додаде функционалност за најавување (логирање) во рамки на нивото за пристап до податоци а исто и во класите од нивото на корисничкиот интерфејс (UI), се добива ситуација во која функционалноста за најавување се повторува. Иако секоја од класите има многу различна примарна функционалност, кодот потребен да се изврши таа секундарна функционалност е најчесто идентичен. Како совет односно решение од сето ова би произлегло дека тој код за најавување треба да се оддели и повика секаде онаму каде што има потреба од него.

Аспект

Гледано за наведениот пример, го разгледавме аспектот за најавување и кон него дефиниравме соодветно решение.

Во компјутерските науки попречните задачи се аспекти на програмата кои влијаат на други проблеми. Овие проблеми често не може да се извадат надвор од остатокот на системот ниту при дизајнот ниту при имплементацијата, а тоа може да резултира со eitherscattering (дуплирање на кодот), заплеткување (создавање на значајни зависности помеѓу системите) или и двете.

На пример, при пишување на апликација за постапување со некоја медицинска евиденција, книговодството и индексирањето на оваа евиденција се главни задачи, додека ги чувањето на податоци за најавите во историјата на промени на евиденцијата во базата на податоци ќе биде вкрстена задача, бидејќи тие најави се однесуваат на повеќе делови од апликацијата. На овој начин се уништува модуларноста на апликацијата.

Аспект ориентираното програмирање има за цел да ги опфати вкрстените задачи во засебни аспекти за да ја задржи модуларноста на проектот. Преку тоа се дозволува да се изврши изолација и да се реискористи кодот со директно референцирање.

2.3.10. Аспект - ориентирано програмирање

Во софтверското инженерство аспект - ориентираното програмирање (АОП) се смета за програмерска парадигма која има за цел да го зголеми степенот на модуларност на апликациите преку дозволување на поделба (раслојување) на вкрстените задачи. АОП оттука е основа за аспект-ориентираниот развој на софтвер. АОП вклучува и програмерски методи и алатки со кои се поддржува модуларизацијата (поделбата) на одговорностите (задачите) на ниво на изворен код, додека „аспект-ориентираниот развој на софтвер“ се однесува на една цела инженерска дисциплина.

Аспект - ориентираното програмирање дозволува да попречните задачи т.е. одговорности бидат третирани како секундарни. Така, на пример, кај повеќето програми е потребно да постои некоја форма, сигурносни поставки и најавување. Овие сигурносни поставки и најавувањето на апликациите може да се смета за секундарна задача, додека примарна задача на апликацијата е да го исполни бизнис процесот за кој е создадена.

Во основа повеќето задачи во рамки на проектите се сметаат за секундарни. Така, на пример, при градење на еден автоматизиран систем мора да го пројдеме процесот т.е. задачата да изворниот код биде испомпајлиран (преведен) во бинарен код. Тоа е секундарна задача, но примарната е да се изгради автоматизиран систем со цел да се намали ризикот од појава на човечки грешки и за да се намали времето на извршување на дадена задача.

Аспект - ориентирано програмирање подразбира рушење логиката на програмата во различни делови (така наречени одговорности, кохезивни области кои исполнуваат одредена функционалност). Скоро сите програмски парадигми поддржуваат некое ниво на групирање и енкапсулација на одговорностите во одвоени, независни субјекти преку обезбедување апстракции (на пример: процедури, модули, класи, методи) кои можат да се користат за имплементација, апстракција и составување на дадените одговорности (задачи). Задачите кои пркосат на овие форми на имплементација се всушност попречните задачи, бидејќи тие „пресекуваат" повеќе апстракции во програмата.

2.3.11. QML

QML (Qt мета јазик или Qt јазик за моделирање) претставува JavaScript базиран, декларативен јазик за дизајн на апликации. Развиен е од страна на Nokia како дел од Qt framework. Идејата за неговото создавање била со него да бидат покриени сите оперативни платформи наменети за мобилни уреди. Така, апликација создадена со QML да може да биде интерпретирана на било кој мобилен уред.

Но, во 2011 година, Nokia со склучувањето на партнерство со Microsoft за консолидација на компанијата го прекинува понатамошниот развој на QML делумно поради неговата неуспешност, но и поради преземената обврска мобилните уреди кои Nokia ги произведува да користат исклучиво Microsoft технологии, Windows Phone.

2.3.12. Постојни имплементации на модуларни решенија

Можностите за имплементација на модуларниот пристап на градење на апликациите постоеле од секогаш главно во сите архитектури кои некогаш се користеле. Како наједноставен пример за можност за имплементација на модуларниот пристап е во C++ (или дури и изворен C јазик) со користење на компоненти спакувани во .dll датотеки кои потоа се вчитуваат во некоја апликација обвивка која истите ги користи. Модуларниот пристап како идеја односно начин за градење на апликациите е доста стар, како што споменавме Dijkstra уште во далечната 1974 година за модуларниот пристап зборува како за принцип на иднината. Проблемот е што во основа постојните архитектури не го промовираат и пропагираат модуларниот принцип на градење на апликациите и практично не постои архитектура која го има имплементирано начелото на единствена одговорност. Во Java постои обид за имплементирање на модуларни решенија со користење јава портлети кои се вчитуваат во апликација школка, обид кој се оценува како неуспешен поради големиот степен на виртуелизација. Кај постарите Microsoft технологии (Windows Forms) имплементацијата на модуларни решенија се сведува на наведениот пример за градење на модуларна апликација во изворен C јазик, односно целокупниот процес на вчитување на компонентите, комуникација, спречување на повторно вчитување на веќе вчитани компоненти и слично треба да биде осмислен и реализиран од развивачите на апликацијата. Тоа претставува дефокусирање од главната задача на тимот или индивидуалецот кој ја развива апликацијата, наместо да се фокусира на имплементација на бизнис логиката и барањата на проектот фокусот е имплементација на модуларната архитектура, што за крајниот корисник можеби е незначајно. Ова е главниот проблем зошто од периодот кога Dijkstra како врвен научник во информатичката област, научник кој ги поставил основите на скоро сите подобласти увидел дека модуларниот пристап на градење на апликациите е пристапот кој нуди најголеми можности за надградливост и лесно управување со промените во апликациите, поминале

цели 40 години, а практично и не постои некоја општоприфатена ахритектура за развој на модуларни апликации. Од друга страна, неприфаќањето на модуларниот пристап на градење на апликациите и некористењето на најновите технологии од стана на софтверските компании доведува до значајно намалена продуктивност при работењето. Како пример ќе го посочиме фактот дека во Македонија трите водечки софтверски компании во периодот од 1995 до 2000 година, денес во 2013 година се некаде на дното по профитабилност само поради две причини, ненавремено прифаќање на нови технологии и несоодветно пренесување на податоци за експертските системи. Несоодветното пренесување на податоци за експертските системи е директна последица на некористењето на технологии како WPF како замена за Windows Forms, поради недоволната сегрегација на проектите на хоризонтално ниво и потешкотиите кои се јавуваат за документирање и водење на промените.

Водени од желбата за прогрес и олеснување на процесите за развој на софтверски апликации, како и лесно реискористување на веќе изградени компоненти, група на ентузијасти во 2009 година во рамки на CodePlex (програмата на Microsoft за слободно споделување на изворен код) формира група за развој на насоки за градење на композитни апликации (Composite Application Guidance) и истиот ќе прерасне во Prism проект / архитектура за развој на модуларни апликации на .Net платформа.

3. Предлог-иновативно решение за модуларна архитектура (Innovative modular architecture solution)

3.1. Опис на Prism библиотеката [17]

Во почетоците Prism проектот е развиван вон рамки на .Net framework, како проект на codeplex со работно име Composite Application Guidance for WPF, а името Prism го добива подоцна во 2012 во процесот на интеграција и прифаќање во .Net framework-от и MSDN. „Опис на Prism библиотеката“ претставува објаснување за градење на композитни апликации со користење на Prism архитектура преку примери кои ги покажуваат нејзините елементарни функционалности.

3.1.1. Вовед во Prism

Prism претставува принцип на дизајн кој нуди олеснување во градењето на содржајни, флексибилни, лесни за одржување WPF (Windows Presentation Foundation) desktop апликации, Silverlight Rich Internet Applications (RIA) апликации како и Windows Phone 7 апликации.

Користејќи шаблони за дизајн кои олицетворуваат важни архитектурни принципи на дизајн како начелото на единствена одговорност или принципот за раслојување на одговорностите за одредени типови задачи, Prism е од помош во дизајнот и градење на апликациите преку можноста за употреба на лабаво спарени (loosely coupled) компоненти кои може да се развиваат независно, но можат да бидат и понатаму лесно и беспрекорно интегрирани во апликацијата и да формираат една целина. Ова е уште една дефиниција која ги дефинира овие типови на апликации како композитни.

Prism генерално е наменет за програмери кои развиваат WPF или Silverlight апликации кои содржат повеќе екрани, содржајна корисничка интеракција и податочна визуализација и олицетворуваат значајна презентациска и бизнис логика. Ваквите апликации се во интеракција со повеќе системи и сервиси во позадина и преку користење на повеќеслојна архитектура може да се разделени на повеќе слоја. Очекувано е дека со текот на времето апликацијата ќе се развива со цел да одговори на новите потреби кои произлегуваат од новите барања на бизнисот.
Или накратко овие апликации се „дизајнирани да траат“ и „дизајнирани за лесна надградливост“. Апликациите кои немаат потреба од овие карактеристики нема да

добијат нешто значајно со користење на ваквиот тип на архитектура како што налага Prism.

Предизвици при развој на клиентски софтвер

Вообичаено, програмерите на клиентски апликации се соочуваат со неколку предизвици. Функционалностите кои одредена апликација треба да ги поседува може да се менуваат со текот на времето. Новите бизнис потреби, новите технологии кои може да станат достапни, дури и повратните информации кои клиентот ги дава во развојниот циклус може значајно да влијаат и да ги дефинираат барањата за функционалностите на апликацијата. Оттука е важно да се изгради апликацијата така што е флексибилна и да може лесно да биде надградена или променета со текот на времето.

Дизајнирање на ваков тип на флексибилност може да биде тешко да се постигне. Потребна е архитектура која дозволува засебни делови на апликацијата да може да бидат независно развиени и тестирани, како и да може да бидат променети во иднина, во изолација, без притоа да влијаат врз останатите делови на апликацијата.

Повеќето апликации кои се користат во компаниите се доволно комплексни и за нивен развој е потребен повеќе од еден програмер, можеби и поголем тим од програмери во кои би биле вклучени и дизајнери на корисничкиот интерфејс и локализери на различни јазици. Може да биде значаен предизвик носењето на одлука како да се дизајнира апликацијата за да би можеле повеќе програмери или тимови ефективно да работат на различни делови на апликацијата независно едни од други, и сепак да се осигури заедничкото функционирање на тие засебно градени делови на апликацијата кога ќе бидат интегрирани во една целина.

Дизајнирањето и градењето на апликациите во монолитен стил може да доведе до тоа да апликацијата биде многу тешка и неефикасна за одржување. За дадена апликација да може да ја наречеме монолитна треба да важи условот да нејзините градивни компоненти се цврсто спарени и да нема јасна сепарација помеѓу нив. Вообичаено апликациите кои се вака дизајнирани се подложни на проблеми и им го загорчуваат животот на програмерите. Тешко е да се додаваат нови можности на апликацијата или замена на постоечки функционалности, разрешувањето на *бугви* без притоа да се нарушат други делови од кодот е практично невозможно, а отежнат е целокупниот процес на тестирање и поставување во функција на апликацијата. На крај влијае на способноста за здружена ефикасна работа на програмерите и дизајнерите.

Композитен пристап

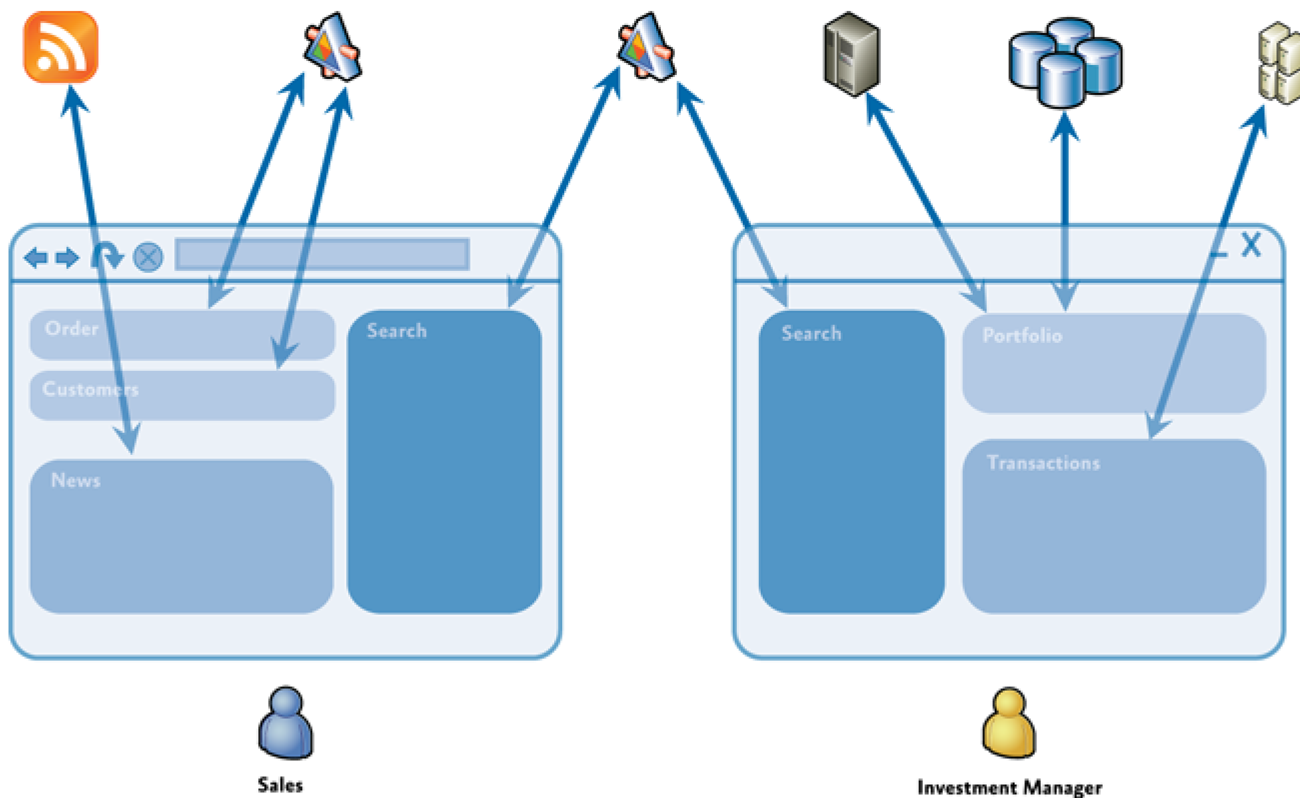
Ефективно средство за наведените предизвици е разделувањето на апликацијата во број на дискретни, лабаво спарени, полунезависни компоненти кои може да бидат интегрирани заедно во апликација Обвивка "shell" и така да формираат кохерентно решение. Апликациите кои се дизајнирани и изградени на овој начин се нарекуваат композитни апликации.

Композитните апликации нудат многу предности, меѓу кои следните:

- Овозможуваат модулите да бидат индивидуално развиени, тестирани и поставувани во функција од различни индивидуи или под тимови. Исто така овозможуваат да бидат полесно менувани и проширувани со нови функционалности, со тоа целокупната апликација да биде лесно проширлива и одржлива. Важно е да се забележи дека секоја индивидуа има лична корист од создавањето на апликации кои полесно се тестираат и одржуваат преку користење на композитниот пристап.
- Нудат заедничка обвивка (shell) која е составена од различни UI компоненти овозможени од разни модули кои се во меѓусебна интеракција преку лабаво спарување. Ова го намалува несогласувањето т.е. раздорот кој се јавува кога повеќе програмери додаваат нови функционалности на корисничкиот интерфејс (UI), а во исто време го поттикнува комбинираната/заедничка појава.
- Поттикнуваат реискористување и јасно раздвојување на одговорностите помеѓу функционалностите на апликацијата на хоризонтално ниво, пример како што се најавување и автентикација, но и на вертикално ниво во однос на бизнис логиката која е специфична за дадена апликација. Ова, исто така, овозможува многу полесно управување на меѓузависностите и интеракцијата помеѓу компонентите на апликацијата.
- Помагаат да се одржи сепарацијата на улогите во развојните тимови дозволувајќи индивидуата или поттимот да се фокусира на одредена задача или парче функционалност во согласност со нивните професионални познавања т.е. поле на експертиза. Особено тоа овозможува појасно раздвојување на корисничкиот интерфејс и бизнис логиката на апликацијата, со други зборови дизајнерот на корисничкиот интерфејс може да се фокусира на создавање на кориснички интерфејс кој нуди по содржајно и по богато корисничко искуство.

Композитните апликации се многу погодни за широк дијапазон на клиентски апликации. На пример, тие се идеални за создавање на апликации кај кои е потребно постигнување на максимална ефикасност на корисничкиот интерфејс, а во позадина да се користат и неколку различни системи.

На слика 6 е прикажан пример на ваков тип на композитна апликација, која користи повеќе системи во позадина.



Слика 6. Композитна апликација, која користи повеќе системи во позадина

Figure 6. Composite application with multiple backend systems

Конкретно кај овој тип на апликација на корисникот може на корисничкиот интерфејс да му се прикажат оние функционалности кои него го засегаат, односно функционалностите кои се поврзани со задачите кои ги извршува. Како што е прикажано на корисникот од кој работи со продажба му се прикажуваат модулите кои овозможуваат преглед на нарачките кои се вршат, вестите кои се важни за неговата работа како и модул за пребарување, додека кај менаџерот кој се занимава со инвестирање функционалностите и нивниот распоред се сосема поинакви. Јасното раздвојување на апликациската логика и корисничкиот интерфејс е тоа што ова го овозможува конзистентен и карактеристичен приказ на составните модули.

Дополнително она што композитните апликации го овозможуваат, а е карактеристично е тоа што тие може да бидат создадени динамички т.е. нова функционалност/модул може да биде додаден во обвивката на апликацијата, дури и во време на нејзино извршување.

Предизвици кои е предвидено да се решат со Prism

Иако Prism помага да се решат многу од предизвиците со кои се соочуваме при градење на WPF или Silverlight апликации, постојат и предизвици за кои Prism не нуди директно решавање. На пример:

- поврзување со извори и синхронизација на податоци;
- сервиси и инфраструктура за проследување на пораки;
- автентикација и авторизација;
- перформанси на апликациите;
- регистрирање на верзии на апликациите;
- справување со грешки.

Цели на дизајнот на Prism

Prism е дизајниран да го помогне дизајнот и градењето содржајни, флексибилни и лесни за одржување WPF или Silverlight апликации. Prism библиотеката имплементира дизајн начела кои олицетворуваат принципи на дизајн на софтвер, како што се раслојување на одговорностите за одредени типови задачи и лабавото спарување на компонентите. Преку користење на овие начела на дизајн и можностите кои ги нуди самата библиотека може да се дизајнираат и градат компоненти кои може да се развиваат независно, а истите би се користеле со лабава врска на спарување и со нив лесно и беспрекорно би се градеде апликациите.

Со тоа што Prism библиотеката во своето јадро ги има инкорпорирано принципот за раслојување на одговорностите за одредени типови задачи и лабавото спарување на компонентите нуди многу придобивки меѓу кои најзначајни се:

Реискористување

Prism промовира повторна и неограничена употреба на компонентите и сервисите. Реискористувањето гледано од аспект дека секоја компонента и нејзината функционалност е тестирана во засебни unit-тестови и како таква готова може да биде интегрирана во апликациите дури и во период на извршување е значајно и поради фактот што на апликациско ниво преку користење на модули одредена функционалност може да се користи лесно помеѓу разнородни апликации.

Проширливост

Prism помага во градењето на апликациите, со тоа што ја остава отворена можноста за проширување на нивната функционалност преку управување на меѓу зависностите на компонентите од кои таа е градена, значи дозволува нивна замена со алтернативни имплементации и во време на извршување на апликацијата, како и со можноста за разградување на апликацијата на меѓусебе независни модули кои како такви можат да се ажурираат. Интересно е тоа што дури и компонентите од кои е градена Prism библиотеката може да бидат проширени или надградени.

Флексибилност

Prism овозможува градење на флексибилни апликации преку можноста која ја нуди за лесна замена на одредени функционалности како што нови се развиваат и интегрираат во концептуалната форма на апликацијата. Prism исто дозволува да WPF и Silverlight апликациите користат заеднички сервиси и компоненти, дозволувајќи притоа да самата апликација биде користена на најсоодветен начин. Дополнително околу флексибилноста може да се каже можноста за надворешна конфигурација на компонентите, која е овозможена.

Тимски развој на апликации

Prism го поттикнува тимскиот развој на апликации со тоа што овозможува различни тимови или индивидуи да работат на еден проект паралелно и независно едни од други. Архитектурата помага да се намалат меѓу тимските зависности и дозволува секој тим или индивидуа се фокусира на одредена функционална област (како графичкиот UI дизајн, имплементација на бизнис логиката, развој на јадрото на инфраструктурата) или различна функционална област на ниво на бизнис логика (пр. функционалност за автентикација, за продажба, сметководство итн.).

Квалитет

Prism цели кон подобрување на квалитетот на апликациите со тоа што сите заеднички сервиси и компоненти се претходно комплетно тестирани пред да се проследат како готови. И Prism е граден врз истите принципи и насоки на дизајн, со истите постапки и упатства кои и самиот ги пропагира така што и Prism налага целиот развој да е фокусиран на имплементација на потребите/функционалностите кои апликацијата треба да ги задоволи наместо фокусот да е на тестирањето на имплементациите.

Важно е да се нотира фактот што Prism е дизајниран така што може да се користат можностите кои ги нуди и насоки за дизајн поединечно, или групно зависно од потребите на одредено сценарио за развој на апликација. Она што е вметнато во неговиот дизајн е тоа што може да биде постепено прифатен т.е. дозволува да се користат една по една сите негови можности за одредена апликација без притоа да се прават крупни структурни промени на апликацијата како што тие можности на архитектурата се усвојуваат.

И финално од причина што тестирањето на софтверот треба да се смета за активност од врвен приоритет и активност која ќе биде цврсто интегрирана во развојниот процес, Prism овозможува широка поддршка за различни типови на софтверски тестирања, и со тоа дозволува дизајн и развој на апликации кои лесно се тестираат.

Клучни концепти на Prism

Prism обезбедува можности и дизајн-шаблони кои може да бидат многу непознати и чудни особено ако за првпат се среќаваме со концептот на композитни апликации. Во овој дел накратко ќе бидат објаснети клучните концепти кои се во позадина на Prism и ќе бидат дефинирани терминологијата и поимите кои се користат во Prism.

Модули / Modules

Модули се пакети на функционалности коишто можат да бидат независно развивани, тестирани и (по избор) поставени во функција. Во повеќето случаи модулите се развиваат и одржуваат од раздвоени тимови. Типична Prism апликација е изградена од повеќе модули. Модулите може да претставуваат некоја специфична функционалност поврзана со бизнис операција и може да ги опфати view-ата (views), сервисите и податочните модели кои се потребни за имплементација на функционалноста. Модулите, исто така, може да се користат за да опфатат одредена заедничка апликациска инфраструктура или сервис (на пример, сервис за најавување или менаџирање на исклучоци) кои може да се реупотребат во повеќе апликации.

Каталог на модули / Module catalog

Во композитните апликации модулите мора да бидат откриени и вчитани во време на извршување на апликацијата. Во Prism, каталогот на модули се користи за во него да се специфицира кои модули ќе се вчитаат, кога ќе се вчитаат и по кој редослед. Каталогот на модули се користи од страна на „менаџерот на модули“ / ModuleManager и „вчитувачот на модули“ / ModuleLoader компонентите, кои се одговорни за преземање на модулите ако тие не се наоѓаат локално, преземање на assembly-јата на модулите во апликацискиот домен, како и за иницијализација на модулот. Prism дозволува каталогот на модули да може да биде специфициран на неколку различни начини, и тоа програмерски преку код, декларативно со користење на XAML или со датотека за конфигурација. Исто така оставена е можноста за дополнителна имплементација на сопствен / прилагоден каталог на модули.

Обвивка (на апликацијата) / Shell

Обвивката е, всушност, апликацијата домаќин во која се вчитуваат модулите. Преку Обвивката се дефинира севкупниот изглед и структура на апликацијата, но обично Обвивката е несвесна за постоењето на модулите кои ќе ги вдоми. Најчесто во неа се имплементирани некои заеднички сервиси или инфраструктура, но во главно функционалностите на апликацијата и содржината се имплементирани во рамки на модулите. Обвивката исто така претставува прозорец или визуелен елемент во кој се вдомуваат различните UI компоненти кои ги нудат вчитаните модули.

View / Views

View се UI контроли кои го опфаќаат корисничкиот интерфејс (UI) за одредена карактеристика/функција или функционален простор од апликацијата. Viewa се користат

сврзани со MVVM или Model-View-Presenter (MVP) шаблоните, кои се користат за да пружат јасно раздвојување на одговорностите помеѓу корисничкиот интерфејс, апликациската презентациска логика и податоците. View-ата се користат за да го опфатат корисничкиот интерфејс (UI) и одредат однесувањето и интеракцијата на корисникот, на тој начин овозможувајќи view-то да биде ажурирано или заменето независно од апликациската функционалност која е во позадина. View-ата користат податочно врзување (data binding) за да комуницираат со view model и presenter класите.

View model-и и presenter-и / View models and presenters

View model-ите се класи кои ја опфаќаат апликациската презентациска логика и состојба. Тие се дел од MVVM шаблонот и опфаќаат голем дел од функционалностите на апликацијата. Presenter-ите се слични на view model-ите од аспект дека и тие ја опфаќаат презентациската логика и состојба, но presenter-ите се дел од MVP шаблонот. И едните и другите дефинираат својства, команди и настани кон кои контролите во view-ата се врзуваат.

Модели / Models

Модел класите ја опфаќаат бизнис логиката и податоците на апликацијата. Тие се составен дел на MVVM и MVP шаблоните. Моделите ги опфаќаат податоците како и поврзаните валидации и бизнис правила за да осигурат конзистентност и интегритет на податоците.

Команди / Commands

Командите се користат за се опфати апликациската функционалност на начин кој дозволува тие да бидат дефинирани и тестирани независно од корисничкиот интерфејс на апликацијата. Тие може да бидат дефинирани како командни објекти или командни методи во view model-от или presenter-от. Prism ги има DelegateCommand и CompositeCommand класите. DelegateCommand служи за доделување на командата, додека CompositeCommand се користи за да претстави збир на команди кои би се повикувале заеднички.

Региони / Regions

Регионите се логички места за чување дефинирани во корисничкиот интерфејс (во Обвивката или внатре во view-ата) во кои view-ата се прикажуваат. Регионите дозволуваат да изгледот на апликацискиот UI (кориснички интерфејс) се ажурира без тоа да носи промени на апликациската логика. Многу заеднички контроли може да се користат како региони, дозволувајќи view-ата да бидат автоматски прикажани во рамки на контролите, како пример следниве контроли (ContentControl, ItemsControl, ListBox, или TabControl). View-ата може да бидат прикажани внатре во регионот со програмирање или автоматски. Prism исто така овозможува поддршка за имплементација на навигација во регионите. Регионите можат да бидат лоцирани од страна на другите компоненти преку RegionManager компонентата, која користи

RegionAdapter и RegionBehavior компоненти за да го координира приказот на view-ата внатре во одреден регион.

Навигација / Navigation

Навигацијата е дефинирана како процес преку кој апликацијата ги координира промените на нејзиниот кориснички интерфејс како резултат како корисничка интеракција со апликацијата или со внатрешна промена на состојбата на апликацијата. Prism поддржува две сценарија за навигација, навигација базирана на состојбата каде состојбата на постојно view е ажурирана и таа треба да се прикаже и второто, навигација за промена на view, каде ново view се создава и се заменува на место од старо view во рамки на корисничкиот интерфејс на апликацијата. Навигација за промена на view користи Uniform Resource Identifier (URI) базиран механизам во комбинација со Prism регионите за да овозможи имплементација на флексибилни навигациони шеми.

Агрегатор на настани / EventAggregator

Компонентите во композитните апликации често имаат потреба да комуницираат со други компоненти и сервиси во апликацијата преку лабава врска. За да го овозможи ова Prism ја има EventAggregator компонентата во која е имплементиран издавач-претплатник / publisher-subscriber механизам на настани (events), на тој начин дозволувајќи компонентите да обајават/издадат настани и други компоненти претплатници на тие настани (events) без притоа ни едните ни другите да имаат потреба од директно референцирање кон другата компонента. EventAggregator се користи и за комуникација помеѓу компоненти дефинирани во различни модули.

Контејнер за вметнување на зависности (зависни елементи) / Dependency injection container

Dependency injection containers најчесто се користи и само контејнер (container) постојат за да ги задоволат меѓузависностите (потребните елементи) помеѓу компонентите. Задоволувањето на овие меѓузависности вообичаено вклучува регистрација и разрешување. Dependency Injection (DI) шаблонот е имплементиран во Prism за да може да се управува со меѓузависностите на компонентите. Dependency Injection овозможува да зависностите (потребните елементи) на компонентите кои се вчитуваат да бидат задоволени во време на извршување, што е во насока на поддршка на проширливоста и олеснетото тестирање. Prism е дизајниран да функционира заедно со MEF и Unity или било кој Dependency Injection контејнер преку ServiceLocator.

Сервиси / Services

Сервисите се компоненти кои опфаќаат функционалности кои немаат ништо со корисничкиот интерфејс, функционалности како на пример најава, менаџирање на исклучоци, пристап до податоци и сл. Сервисите може да бидат регистрирани во апликацијата или внатре во модулите. Сервисите најчесто се регистрирани во Dependency Injection контејнерот за тие да можат да бидат лоцирани кога ќе бидат потребни т.е. побарани од компоненти чии функционалности зависат од нив.

Контролори / Controllers

Контролорите се класи кои се користат за да се координира конструкцијата и иницијализацијата на view-ата кои треба да бидат прикажани ви регион на корисничкиот интерфејс на апликацијата. Контролорите ја опфаќаат презентациската логика која одредува кои view-a треба да се прикажат. Контролорот го користи механизмот за навигација за промена на view, кој овозможува механизам на проширлива URI базирана навигација за координација на конструкцијата и поставувањето на view-a во регионите.

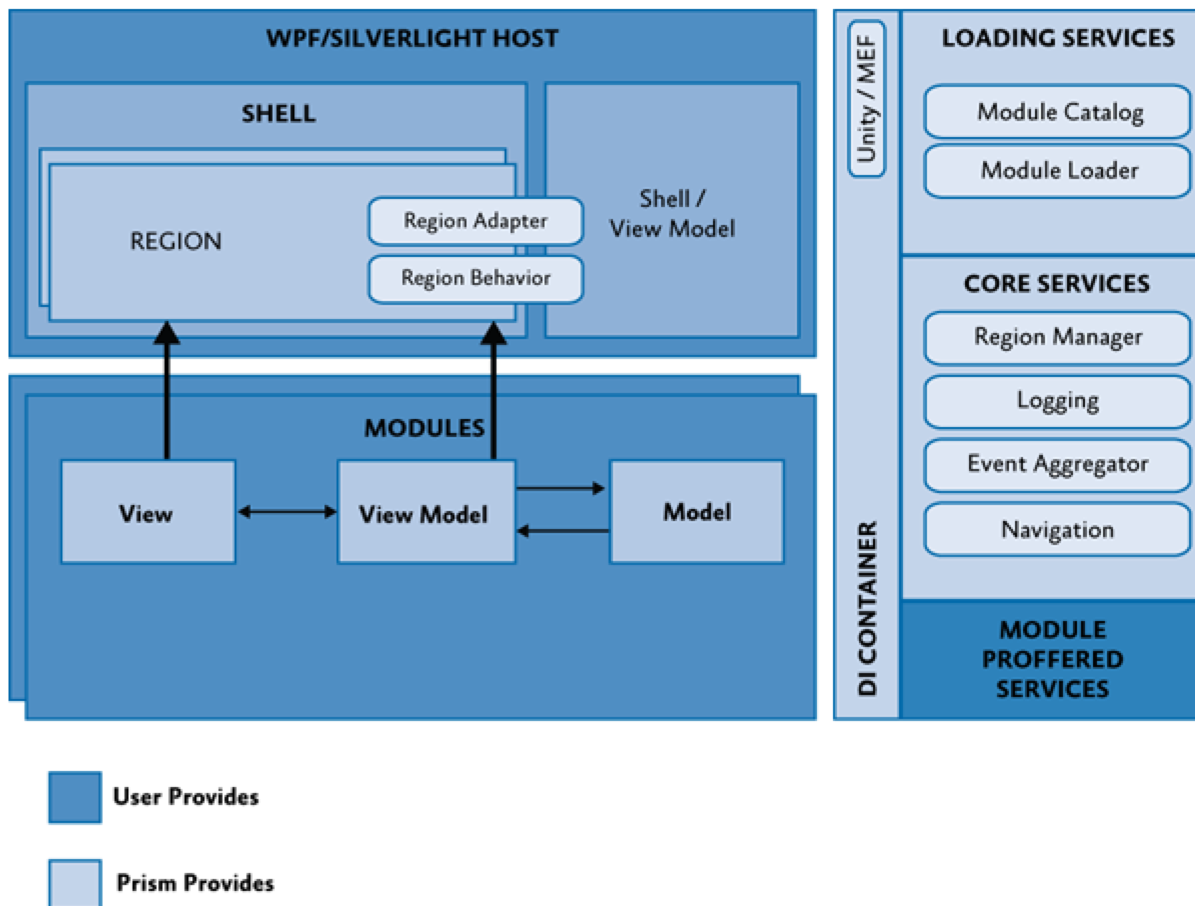
Компонента за иницијализација / Bootstrapper

Bootstrapper компонентата се користи во апликацијата за иницијализација на разни Prism компоненти и сервиси. Се користи за иницијализација на dependency injection контејнерот во кој се регистрираат сите останати компоненти и сервиси. Исто така се користи за конфигурирање на каталогот на модули и view-то на Обвивката и view model-от или presenter-от.

Мултитаргетирање / Multi-targeting

Prism поддржува развој на апликации кои целат и кон WPF и кон Silverlight. Со усвојување на одделен презентациски шаблон како што се MVVM или MVP, може да се оддели корисничкиот интерфејс од презентацискиот дел на апликацијата и бизнис логиката. View model, presenter, и model класите може да се реискористат и во WPF и во Silverlight верзиите на една иста апликација. Потоа можат да бидат дефинирани и соодветни view-a.

Prism е дизајниран така да овозможува користење на било која од неговите можности и дизајн насоки индивидуално или целокупно, зависно од потребите на апликацијата која се развива. Може да се користи MVVM шаблонот, модуларноста, регионите, командите, настаните или било која комбинација од нив без притоа да има потреба од прифаќање на сите нив во целина. Секако ако сакаме да извлечеме целосно искористување на придобивките што раслојувањето на одговорностите за одредени типови на задачи и лабавото спарување ги нудат обично се употребуваат повеќето од капацитетите/можностите на Prism во надоврзувајќи се една над друга. На следнава слика е прикажана типична Prism апликациска архитектура и е покажано како различни можности кои ги нуди Prism можат да функционираат заедно внатре во мулти модуларна композитна апликација.



Слика 7. Типична архитектура на композитна апликација имплементирана со Prism библиотеката

Figure 7. Typical composite application architecture with the Prism Library

Prism апликациите се состојат од обвивка на апликацијата во која се дефинирани региони за приказ на најважните view и заеднички сервиси кон кои што можат да пристапат модулите кои се вчитуваат потоа. Во обвивката е дефиниран и каталог во кој што е специфицирано кои модули ќе се вчитаат во време на стартување на апликацијата а кои по потреба. Dependency injection контејнерот е исто така дефиниран и преку него се овозможува да зависностите (зависните елементи) на компонентите бидат задоволуваат во време на извршување. Заедничките сервиси и компоненти се регистрираат во контејнерот од страна на Bootstrapper во моментот кога апликацијата се стартува.

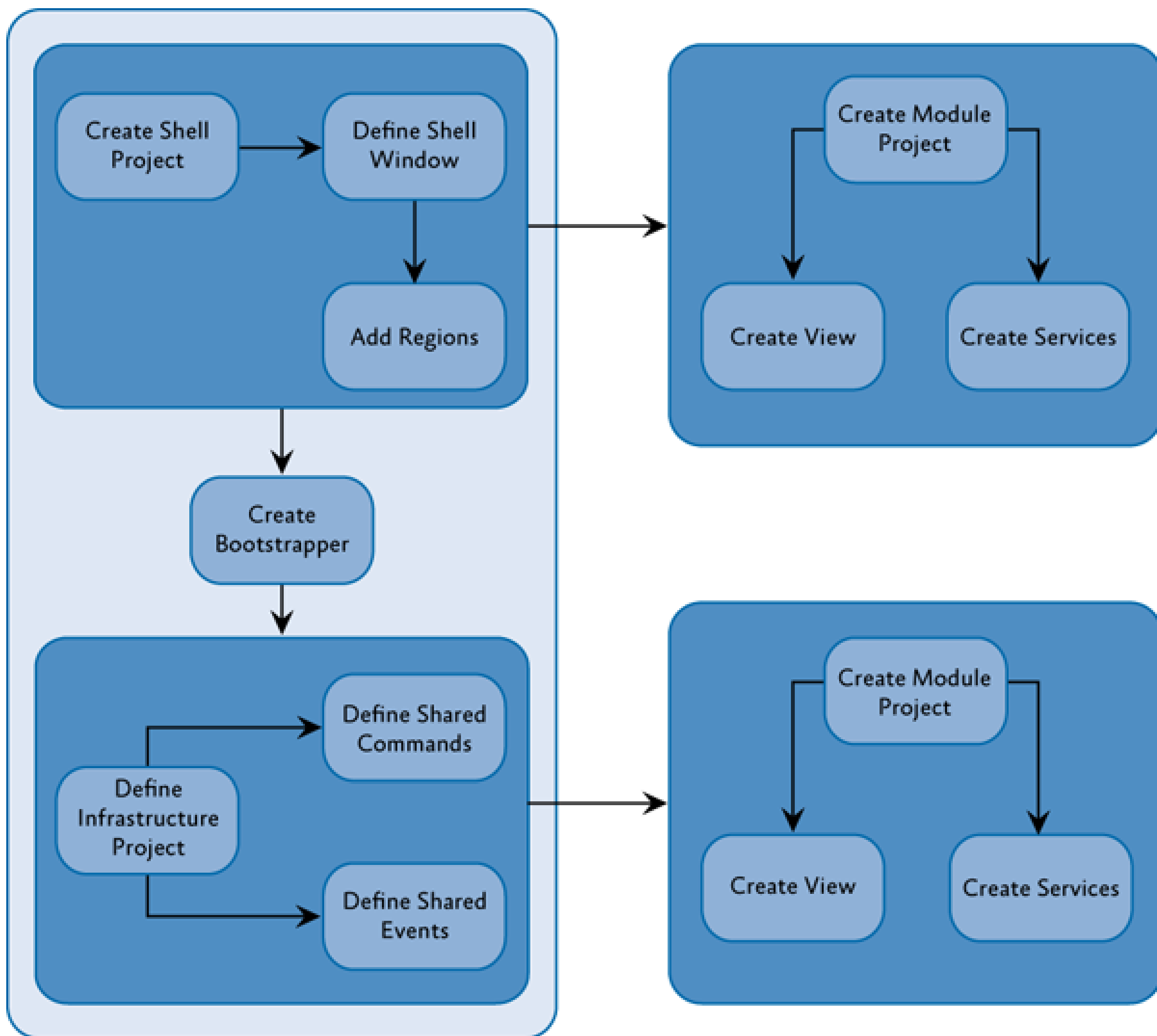
Посебните модули опфаќаат само дел од целокупната функционалност на апликацијата, а со користење на презентациски шаблон како што е MVVM во нив се дефинирани засебни view-a, view model-и, модели и сервисни компоненти. Кога некој модул се вчитува view-то дефинирано во него се прекажува во регионите дефинирани во Обвивката на апликацијата. Кога иницијализацијата на модулот ќе заврши, корисникот се движи низ апликацијата со користење на навигација базирана на состојбата и навигација

за промена на view за да се координираат визуелните промени или приказот на нови view-а внатре во апликациските региони.

3.1.2. Користење на Prism

Откога ги прикажавме главните можности и дизајн шаблони кои Prism ги поседува, може да видиме како изгледа да се користи Prism при развој на нова апликација. Овој дел од тезата дава краток преглед на почетните чекори што треба да се направат за да се создаде елементарна Prism апликација. Оваа елементарна апликација потоа може да биде искористена за проширување со останатите можности и дизајн шаблони кои Prism ги поседува, зависно од сценариото кое треба да се имплементира.

Prism апликациите вообичаено се состои од проект за Обвивката на апликацијата и неколку проекти за модулите. На Слика 8 се прикажани основните активности кои треба да се направата за да се започне со развивање на композитна апликација со користење на Prism библиотеката.



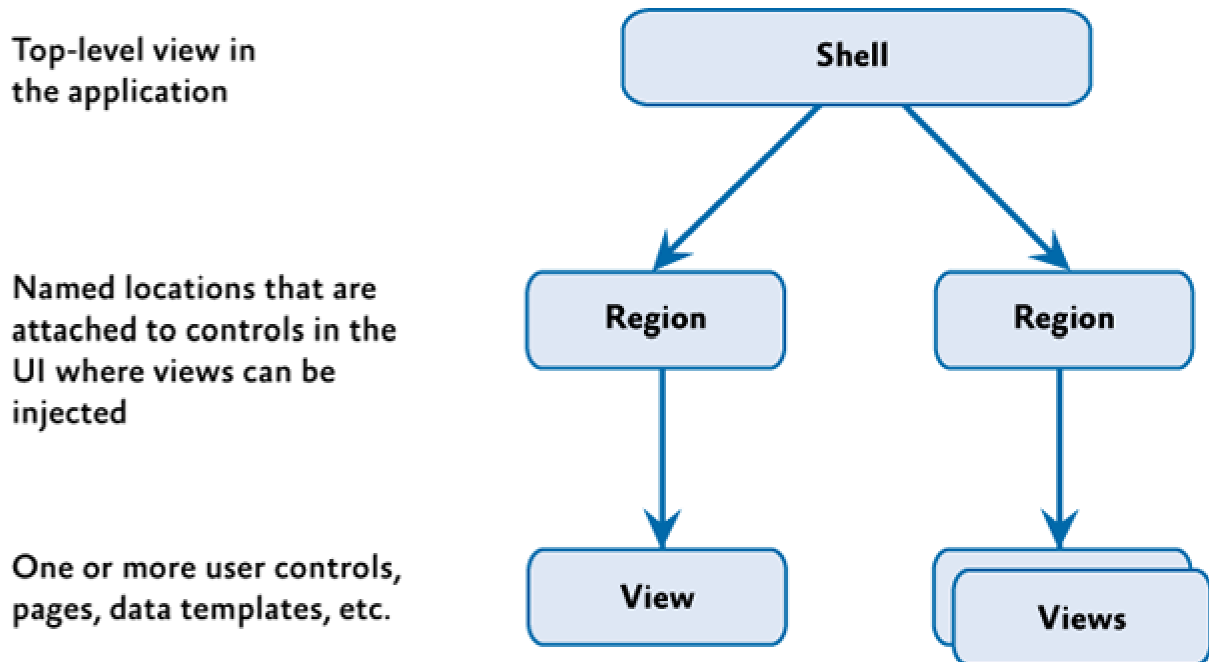
Слика 8. Постапки за создавање на композитна апликација

Figure 8. Activities for creating a composite application

Најчесто Prism апликациите ги искористуваат сите можности на самата архитектура/библиотека кои беа опишани претходно за да би можеле целосно да ги искористат придобивките од принципите за лабаво спарување на компонентите и раслојувањето на одговорностите за одредени типови на задачи. Конкретно во овој пример ќе создадеме базична Prism апликација која ќе се состои од еден модул во кој ќе дефинираме едно view.

Дефинирање на обвивката на апликацијата

Апликациската обвивка служи за дефинирање на основниот изглед на апликацијата. Таквиот изглед е дефиниран со региони во кои модулите можат да ги сместат своите view. Viewa исто како и обвивката може да користат региони во кои може да додадат содржини. Обвивката вообичаено го дефинира целиот изглед на апликацијата и ги содржи сите стилови кои ќе се користат внатре во неа. Прикажано на слика 9.

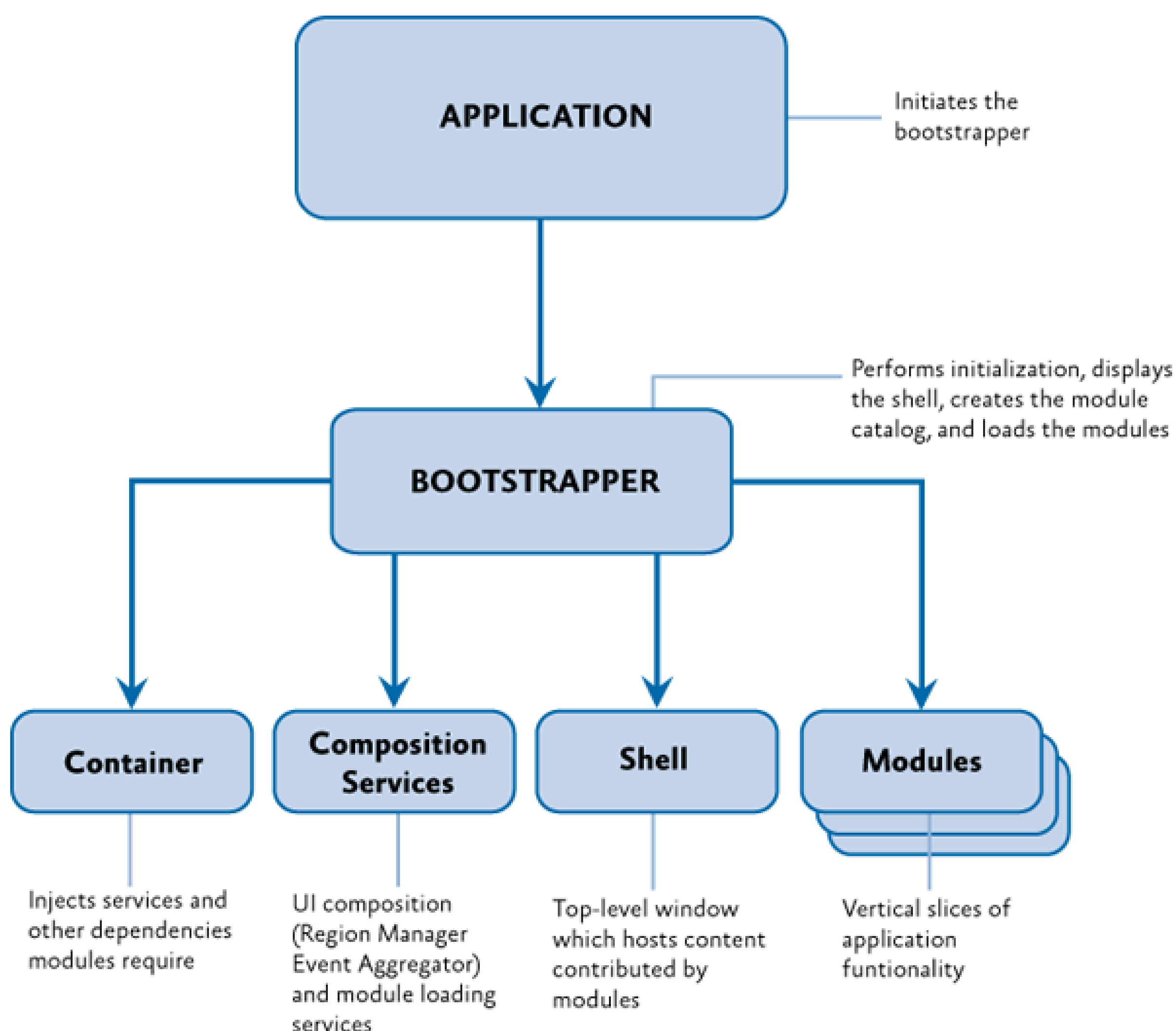


Слика 9. Начин на повикување на обвивката, view и регионите

Figure 9. Shells, views, and regions

Создавање на Bootstrapper

Bootstrapper е она што ги поврзува апликацијата со сервисите на Prism библиотеката и Unity и MEF контејнерите. Секоја апликација има свој специфичен Bootstrapper, кој се создава преку наследување или од UnityBootstrapper или MefBootstrapper класите, како што е прикажано на сликата. Она што треба да се одлучи кој пристап ќе се користи за полнење на каталогот на модули. Стандардно Bootstrapper-от ги логира/чува настаните (event-ите) со користење на .NET Framework Trace класата, но за некои апликации може постои потреба од користење на друг сервис за логирање/чување на event-ите како пр. Enterprise Library logging. Ако постои потреба може да се имплементира некој друг Bootstrapper различен од стандардните UnityBootstrapper и MefBootstrapper.



Слика 10. Дијаграм на поврзување со Prism библиотеката

Figure 10. Diagram demonstrating connecting to the Prism Library

Создавање на модул

Модулот ги содржи view и сервисите специфични за тоа парче на апликациска функционалност. Најчесто модулите се наоѓаат во одделни assembly-a и се развиени од одделни тимови. Модулите се создаваат со изведување на класа која го имплементира IModule интерфејсот. Модулите при иницијализација ги регистрираат нивните view-a и сервиси, и може да ги додаваат во рамки на обвивката. Зависно од типот на модулот, дали зависи од некои други модули се дефинираат таквите меѓу модуларни зависности.

Додавање на view на модулот на обвивката

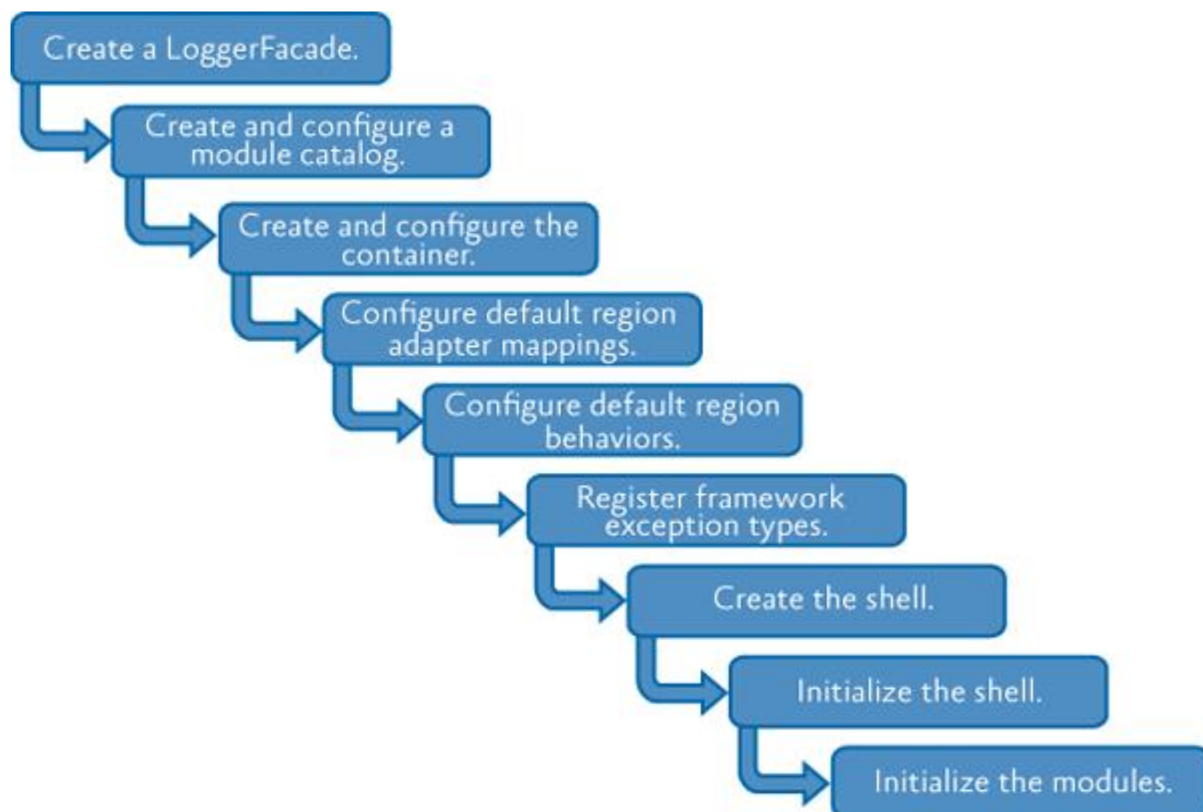
Во време на иницијализација модулите го користат RegionManager-от за да ги лоцираат регионите во обвивката и да додадат едно или повеќе view во тие региони или да регистрираат едно или повеќе типови на view-a во тие региони. RegionManager-от е е одговорен за следење на регионите во апликацијата и е централен сервис кој се иницијализира од bootstrapper.

3.1.3. Иницијализација на Prism апликациите

Што претставува Bootstrapper

Bootstrapper е класа која што е одговорна за иницијализација на апликацијата која ја користи Prism библиотеката. Со користењето на bootstrapper имаме повеќе контрола на тоа како компонентите од Prism библиотеката се поврзани во апликацијата.

Prism библиотеката содржи основна апстрактна Bootstrapper класа која може да биде додефинирана да функционира со кој било контејнер. Многу од методите на Bootstrapper класата се виртуелни методи. Може истите да бидат преклопени за да се создаде соодветна верзија за сопствено изработената имплементација.



Слика 11. Основни чекори на Bootstrapper процесот

Figure 11. Basic stages of the bootstrapping process

Prism библиотеката има и неколку други базични класи изведени од Bootstrapper, во кои се имплементирани соодветните чекори, прикажани на слика 11. Единствената фаза која е оставена за дефинирање во bootstrapper на апликацијата е создавањето и иницијализацијата на обвивката.

3.1.4. Управување / Менаџирање со зависностите помеѓу компонентите

Апликациите базирани на Prism библиотеката се композитни апликации кои најчесто се состојат од мноштво лабаво спарени типови и сервиси. Тие мора да се во интеракција за да ја надополнуваат содржината и добиваат известувања од акциите на корисникот. Од причина што се лабаво спарени/ поврзани, за нив мора да се најде начин да се во интеракција и комуницираат меѓусебно со цел да ја овозможат потребната бизнис функционалност.

За да ги врзат заедно во една целина таквите парчиња, апликациите базирани на Prism библиотеката се ослонуваат врз dependency injection контејнерот. Dependency injection контејнерите ја редуцираат зависноста од спарување помеѓу објектите преку обезбедување на капацитети за инстанцирање на инстанци од класите и управување со нивниот животен век базирајќи се врз нивната конфигурација (конфигурацијата на самиот контејнер). За време на создавањето на објектот контејнерот ги вметнува сите зависни елементи кои му требаат на самиот објект. Ако тие во тој момент се уште се недостапни, контејнерот прво ги наоѓа и создава нив т.е. ги разрешува зависностите па потоа контејнерот го создава објектот.

Постојат неколку предности при користењето на контејнер и тоа:

- контејнерот ја елиминира потребата да самата компонента се грижи за лоцирање на зависностите (зависните елементи) и за управување со времето на нивно постоење.
- контејнерот дозволува заменување на имплементираните зависности без да влијае на компонентата.
- контејнерите го олеснуваат тестирањето со тоа што дозволуваат зависностите да бидат занемарени.
- контејнерот го олеснува одржувањето со тоа што дозволува нови компоненти лесно да може да бидат додавани во системот.

Во контекст на апликации базирани на Prism библиотеката, постојат неколку специфични предности од користењето на контејнер и тоа:

- контејнерот ги вметнува зависностите на модулот во модулот кога тој се вчитува.
- контејнерот се користи за регистрирање и разрешување на view model-ите и view-ата.
- контејнерот може да ги создаде view model-ите и вметне view-ата.
- контејнерот ги вметнува составните сервиси, како менаџерот на региони (region manager) и event aggregator-от.
- контејнерот се користи за регистрирање на специфични сервиси за модулот, кои имаат специфична функција само за тој модул.

Избор на Dependency Injection контејнер

Prism библиотеката има две опции за Dependency Injection контејнер и тоа Unity или MEF. Но поради тоа што и самиот Prism е проширлив дозволува и други контејнери да може да бидат имплементирани. И двата Unity и MEF ги имаат основните функционалности за Dependency Injection (вметнување на зависности), иако тие функционираат на многу различен начин. Некои од можностите кои ги нудат и двата типа на контејнери вклучуваат:

- регистрирање на типови во контејнерот.
- регистрирање на инстанци во контејнерот.
- задолжително создавање на инстанци од регистрирани типови.
- вметнување на инстанци на регистрираните типови во конструктори.
- вметнување на инстанци на регистрираните типови во својства (properties).
- имаат декларативни атрибути за правење типови и зависности кои мора да бидат управувани.
- разрешување зависности во графот на објектот.

Неколку можности кои ги има Unity а MEF ги нема:

- разрешува конкретни типови без регистрирање.
- разрешува отворена генерика.
- користи пресретнување за да прифати повици кон објектите и додава дополнителна функционалност на целниот објект.

Неколку можности кои ги има само MEF:

- Откривање на assembly-а на одредена локација (фолдер).
- користи преземање на хар датотеки и откривање на assembly-а во нив.
- ги составува наново својствата и колекциите кога ќе се открие нов тип.
- автоматски ги експортира откриените типови.
- составен дел е на .NET Framework-от.

Независно од специфичните функционалности на контејнерите, Prism библиотеката ќе функционира слично без разлика кој од нив ќе се користи. Секако кога се избира контејнерот треба на ум да ги имаме наведените можности.

Фактори при користење на контејнерите

Пред да ги користиме контејнерите треба да размислиме за следното:

- Да размислиме дали е соодветно да регистрираме и разрешуваме компоненти со користење на контејнер:
    - Дали влијанието по перформансите со регистрирање на компонентите во контејнер е прифатливо. На пример ако треба да создадеме 10.000 полигони за да исцртаме некоја површина, разрешувањето на сите тие објекти може да има голем импакт по перформансите.
    - Ако има многу или длабоки зависности, цената за нивно создавање може доста да порасне.
    - Ако компонентата нема никакви зависности или не е нечија зависност можеби нема логика да ја ставаме во контејнер.
    - Ако компонентата има еден единствено множество од зависности кои се интегрален дел од типот и нема да се менуваат, пак нема логика и во овој случај компонентата да е регистрирана во контејнер.
- Треба да размислиме дали времетраењето на компонентата треба да биде регистрирано само еднаш (singleton) или како инстанца:
    - Ако компонентата е некој глобален сервис кој делува како менаџер на ресурси но за еден единствен ресурс тогаш подобро е да биде регистрирана како singleton.
    - Ако компонентата обезбедува некоја заедничка состојба на повеќе корисници исто така треба да е регистрирана како singleton.
    - Ако објектот кој се вметнува треба да има нова инстанца од него секогаш кога на некој објект зависен од него му потребува , тогаш тој треба да се регистрира како не-singleton. Пр. на секое view најверојатно му треба засебен view model.
- Треба да размислиме дали сакаме да го конфигурираме контејнерот преку код или преку конфигурација:
    - Ако сакаме централно да управуваме повеќе сервиси, тогаш го конфигурираме преку конфигурација.
    - Ако сакаме условно да регистрираме специфични сервиси, тогаш го конфигурираме преку код.
    - Ако имаме сервиси на ниво на модули, треба конфигурацијата на контејнерот биде преку код за тие сервиси да бидат регистрирани само кога модулот е вчитан.

Некои контејнери пр. како MEF не можат да бидат конфигурирани преку конфигурациска датотека туку само преку код.

Контејнерите се користат за две примарни цели, регистрирање и разрешување на зависностите.

Регистрирање

Пред да може да се вметнат зависностите во објектот, типовите на зависностите мора да бидат регистрирани во контејнерот. Регистрирањето на тип обично вклучува проследување на интерфејс кон контејнерот и конкретен тип кој го имплементира тој интерфејс. Постојат два начина за регистрирање на типовите и тоа преку код и преку конфигурациска датотека. Најчесто има два начина за регистрирање на типови и објекти во контејнерот преку код и тоа:

Со регистрирање на тип или мапирање со контејнерот. Во соодветно време, контејнерот ќе направи инстанца од типот кој е специфициран.

Со регистрирање на инстанца од постоечки објект во контејнерот како singleton. Контејнерот ќе врати референца до постојниот објект.

Регистрирање на типови со Unity контејнерот

За време на иницијализацијата, еден тип може да регистрира други типови, како view-а и сервиси. Регистрацијата дозволува да нивните зависности бидат обезбедени преку контејнерот и дозволува кон нив да биде пристапено од други типови. За ова да се направи, мора контејнерот да биде внесен во конструкторот на модулот. Следниот код покажува пример како OrderModule типот регистрира тип во неговиот конструктор.

C#

```
public class OrderModule : IModule
{
   public void Initialize()
   {
      this.container.RegisterType<IOrdersRepository,
      OrdersRepository>(new ContainerControlledLifetimeManager());
      ...
   }
   ...
}
```

Пример код 1.

Зависно од контејнерот што се користи регистрацијата може да биде направена и надворешно преку конфигурација (конфигурациска датотека). Предноста на регистрирање во код е тоа што регистрацијата се случува само кога модулот се вчитува.

Регистрирање на типови со MEF контејнерот

MEF користи систем базиран на атрибути за регистрирање типови во контејнерот. Како резултат на тоа регистрирањето на тип во контејнерот е многу едноставно, само бара додавање на [Export] атрибутот на типот како што е прикажано во следниов пример:

C#

```
[Export(typeof(ILoggerFacade))]
public class CallbackLogger: ILoggerFacade
{

}
```

Пример код 2.

Друг начин кога се користи MEF е да се создаде инстанца од класата и да се регистрира таа одредена инстанца во контејнерот. Примерот кој следува покажува таква имплементација со преклопување на ConfigureContainer методот:

C#

```
protected override void ConfigureContainer()
{
   base.ConfigureContainer();

   this.Container.ComposeExportedValue<CallbackLogger>(this.callbackLogger);
}
```

Пример код 3.

Кога се користи MEF како контејнер препорачано е да се користат атрибути за регистрирање на типовите.

Разрешување

Откако еден тип ќе биде регистриран, тој може да биде разрешен т.е. зависноста може да биде вметната. Кога типот се разрешува и контејнерот треба да направи нова инстанца, се вметнува зависноста во инстанцата/инстанците.

Главно кога некој тип се разрешува се случува една од следните три работи:

- Ако типот не бил претходно регистриран, контејнерот ќе врати / исфрли исклучок(exception).

Сепак, некои контејнери како што е пример Unity дозволуваат разрешување на типот без тој претходно да се регистрира.

- Ако типот е регистриран како singleton, контејнерот враќа singleton инстанца. При првото повикување на типот контејнерот ја прави инстанцата и ја чува за идните повици.
- Ако типот е регистриран но не како singleton тогаш контејнерот враќа нова инстанца.

Вообичаено типовите регистрирани со MEF се singleton-и и контејнерот чува референца кон објектот. Во Unity по правило се враќаат новите инстанци на објектот, и контејнерот не чува референца кон објектот.

Разрешување на инстанци со Unity

Следниов код претставува пример (изваден од Commanding QuickStart од Prism) каде OrdersEditorView и OrdersToolBar view се разрешуваат од страна на контејнерот за да се поврзат со соодветните региони.

C#

```
public class OrderModule : IModule
{
    public void Initialize()
    {
        this.container.RegisterType<IOrdersRepository,
         OrdersRepository>(new ContainerControlledLifetimeManager());

        // Show the Orders Editor view in the shell's main region.
        this.regionManager.RegisterViewWithRegion("MainRegion",
                        () => this.container.Resolve<OrdersEditorView>());

        // Show the Orders Toolbar view in the shell's toolbar region.
        this.regionManager.RegisterViewWithRegion("GlobalCommandsRegion",
                        () => this.container.Resolve<OrdersToolBar>());
    }
    ...
}
```

Пример код 4.

OrdersEditorPresentationModel конструкторот ги содржи зависностите ordersRepository и commandProxy кои се вметнуваат кога тој се разрешува и тоа на следниот начин:

C#

```
public OrdersEditorPresentationModel( IOrdersRepository ordersRepository,
OrdersCommandProxy commandProxy )
{
    this.ordersRepository = ordersRepository;
    this.commandProxy     = commandProxy;

    // Create dummy order data.
    this.PopulateOrders();

    // Initialize a CollectionView for the underlying Orders collection.
#if SILVERLIGHT
    this.Orders = new PagedCollectionView( _orders );
#else
    this.Orders = new ListCollectionView( _orders );
#endif
    // Track the current selection.
    this.Orders.CurrentChanged += SelectedOrderChanged;
    this.Orders.MoveCurrentTo(null);
}
```

Пример код 5.

Разрешување на инстанци со MEF

Во следниот пример (изваден од Modularity for Silverlight with MEF QuickStart од Prism) е прикажано како Bootstrapper бара инстанца од обвивката на апликацијата. Наместо да бара одреден тип, во кодот се бара инстанца од интерфејсот.

C#

```
protected override DependencyObject CreateShell()
{
   return this.Container.GetExportedValue<Shell>();
}
```

Пример код 6.

Во која било класа која се разрешува со MEF може да се користи вметнување на конструкторот, како што е прикажано во следниот пример (изваден од ModuleA од the Modularity for Silverlight with MEF QuickStart) во кој има вметнување на ILoggerFacade и IModuleTracker интерфејсите.

C#

```
[ImportingConstructor]
public ModuleA(ILoggerFacade logger, IModuleTracker moduleTracker)
{
   if (logger == null)
   {
      throw new ArgumentNullException("logger");
   }
   if (moduleTracker == null)
   {
      throw new ArgumentNullException("moduleTracker");
   }
   this.logger = logger;
   this.moduleTracker = moduleTracker;
   this.moduleTracker.RecordModuleConstructed(WellKnownModuleNames.ModuleA);
}
```

Пример код 7.

Друга опција е да се користи вметнување на својства, како е прикажано во следниот пример (изваден од ModuleTracker класата од Modularity for Silverlight with MEF QuickStart), каде што се вметнува инстанца од ILoggerFacade интерфејсот.

C#

```
[Export(typeof(IModuleTracker))]
public class ModuleTracker : IModuleTracker
{
    // Due to Silverlight/MEF restrictions, this must be public.
    [Import] public ILoggerFacade Logger;
}
```

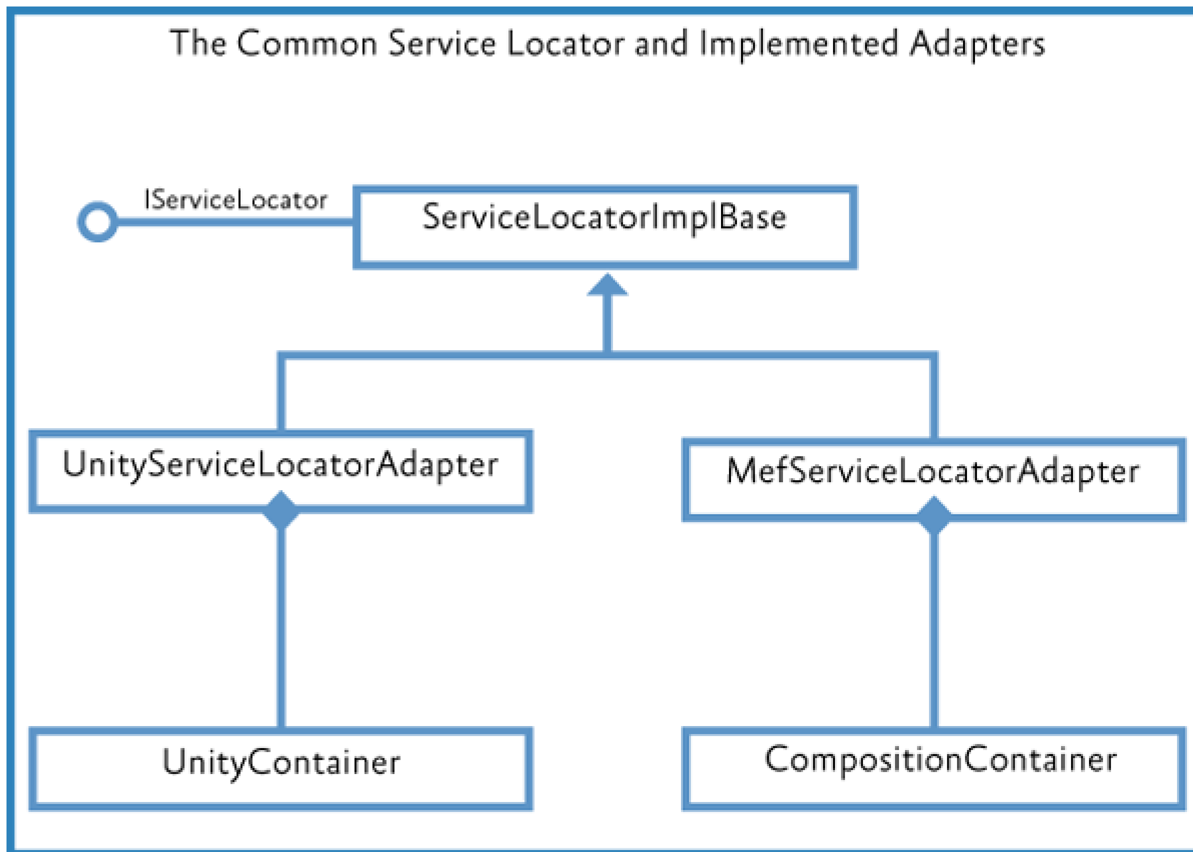
Пример код 8.

Во Silverlight импортираните својства и полиња мора да се јавни.

Користење на Dependency Injection контејнери и сервиси во Prism

Dependency injection контејнерите или скратено само контејнери се користат за задоволување на зависностите помеѓу компонентите. Prism библиотеката има вградена поддршка за Unity и MEF контејнери, но не е врзана само за нив. Од причина што библиотеката пристапува до контејнерот преку IServiceLocator интерфејсот, контејнерот може да биде заменет. За да може тоа да се направи треба контејнерот да го имплементира IServiceLocator интерфејсот. Вообичаено ако имаме потреба од замена на контејнерот го менуваме и bootstrapper кој соодветствува со контејнерот. IServiceLocator интерфејсот е дефиниран во Common Service Locator библиотеката (библиотека - заеднички локатор на сервиси). Ова е само еден начин како да се постигне open source варијанта за апстракција на IoC (Inversion of Control) контролорите. Целта на употребата на оваа библиотека е да се постигне употреба на IoC и Service Location без да се врзе кон одредена имплементација.

Prism библиотеката ги содржи UnityServiceLocatorAdapter и MefServiceLocatorAdapter. И двата адаптери го имплементираат IServiceLocator интерфејсот со проширување на ServiceLocatorImplBase типот. На Слика 12 е прикажана хиерархијата на класите.



Слика 12. Имплементации на Common Service Locator (заеднички локатор на сервиси) во Prism

Figure 12. The Common Service Locator implementations in Prism

IServiceLocator интерфејс

Следниов код го прикажува IServiceLocator интерфејсот.

```
C#
public interface IServiceLocator : IServiceProvider
{
   object GetInstance(Type serviceType);
   object GetInstance(Type serviceType, string key);
   IEnumerable<object> GetAllInstances(Type serviceType);
   TService GetInstance<TService>();
   TService GetInstance<TService>(string key);
   IEnumerable<TService> GetAllInstances<TService>();
}
```

Пример код 9.

Service Locator е проширен во Prism библиотеката со методи како што е прикажано во кодот што следува. Во него може да се забележи дека IServiceLocator се користи само за разрешување да преку него се добие инстанца, и не се користи за регистрирање.

```
C#
public static class ServiceLocatorExtensions
{
   public static object TryResolve(this IServiceLocator locator, Type type)
   {
      try
      {
         return locator.GetInstance(type);
      }
      catch (ActivationException)
      {
         return null;
      }
   }

   public static T TryResolve<T>(this IServiceLocator locator) where T: class
   {
      return locator.TryResolve(typeof(T)) as T;
   }
}
```

Пример код 10.

Методот TryResolve (Unity контејнерот не го поддржува) враќа инстанца од типот што се разрешува ако тој претходно е регистриран, ако не е ќе врати null.

ModuleInitializer користи IServiceLocator за разрешување на модулите во време на вчитување на модулите, како што може да се види во следните примери.

```
C#
```

```
IModule moduleInstance = null;
try
{
   moduleInstance = this.CreateModule(moduleInfo);
   moduleInstance.Initialize();
}
...
protected virtual IModule CreateModule(string typeName)
{
   Type moduleType = Type.GetType(typeName);
   if (moduleType == null)
   {
      throw new ModuleInitializeException(string.Format(CultureInfo.CurrentCulture,
Properties.Resources.FailedToGetType, typeName));
   }

   return (IModule)this.serviceLocator.GetInstance(moduleType);
}
```

Пример код 11.

Важни фактори при користење на IServiceLocator

IServiceLocator не е наменет за користење како (интерфејс на) контејнер за општа намена. Контејнерите имаат поинаква семантика (друго значење) на употреба, која што често ја одредува одлуката зошто токму тој контејнер е избран. Имајќи го ова на ум, добра практика е во апликациите да се користи директно dependency injection контејнер наместо користење на IServiceLocator. Само во неколку случаи е подобро да се користи директно IServiceLocator и тоа кога постои експлицитна потреба од користење на повеќе различни контејнери.

3.1.5. Комуникација помеѓу лабаво врзани компоненти

Кога се развиваат големи комплексни апликации, најчест пристап е да се поделат функционалностите во дискретно множество на assembly на модули. Пожелно е да се елиминира употребата на статичко референцирање помеѓу модулите, со сепак треба да се овозможи начин како модулите меѓусебно да комуницираат во лабава врска. Prism библиотеката ги нуди следниве начини на комуникација:

3.1.5.1. Директно командување / Solution commanding.

Се користи кога се очекува непосреден одговор од страна на корисникот. Prism библиотеката ги има DelegateCommand команда што овозможува повикување на метод и CompositeCommand која овозможува комбинирање на повеќе команди во една. И двете претставуваат имплементација на ICommand интерфејсот. Во наредниот пример е прикажано создавање на DelegateCommand, примерот е земен од ArticleViewModel.cs.

C#

```
public class ArticleViewModel : NotificationObject
{
    private readonly ICommand showArticleListCommand;

    public ArticleViewModel(INewsFeedService newsFeedService,
                            IRegionManager regionManager,
                            IEventAggregator eventAggregator)
    {
        this.showArticleListCommand = new DelegateCommand(this.ShowArticleList);
    }

    public ICommand ShowArticleListCommand
    {
        get { return this.showArticleListCommand; }
    }
}
```

Пример код 12.

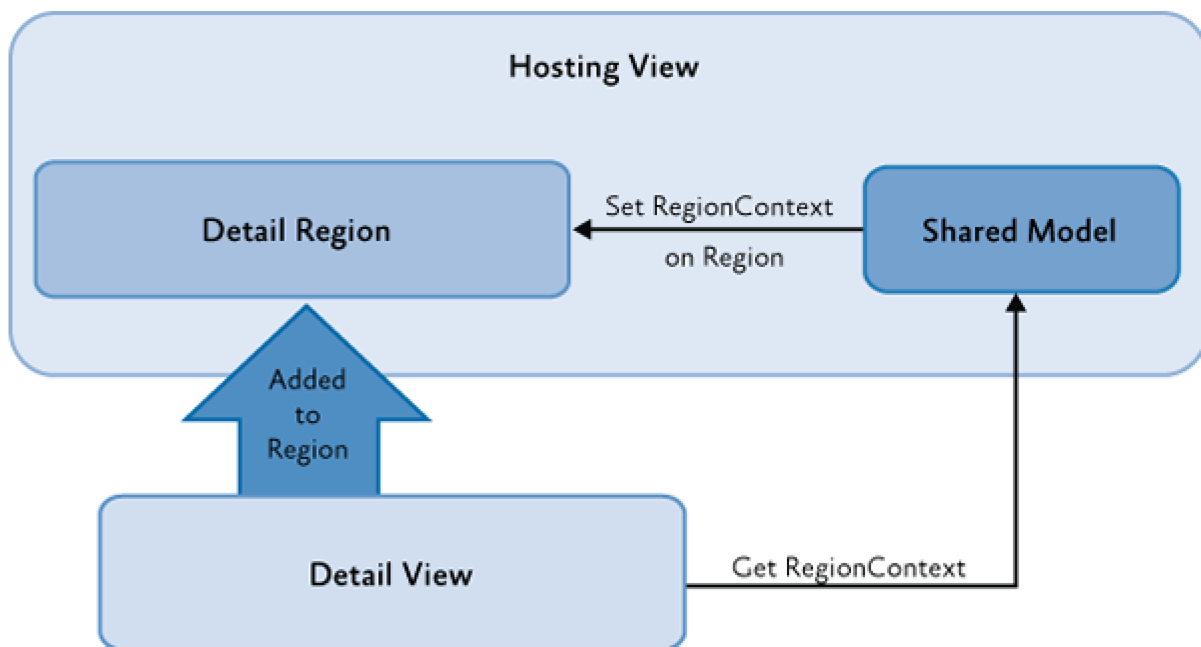
И врзување на команда.

```
XAML
<Button Name="MyCommandButton"
 prism:Click.Command="{Binding MyCompositeCommand}"/>Execute MyCommand
</Button>
```

Пример код 13.

3.1.5.2. Контекст на Региони / Region context

Се користи кога се разменуваат информации помеѓу регионот домаќин и view кои се наоѓаат во него. Најчесто се користи кога во рамки на некој регион се прикажани информации за нешто и сакаме во друг регион да прикажеме некои дополнителни информации, како што е прикажано на Слика 13.



Слика 13. Користење на RegionContext

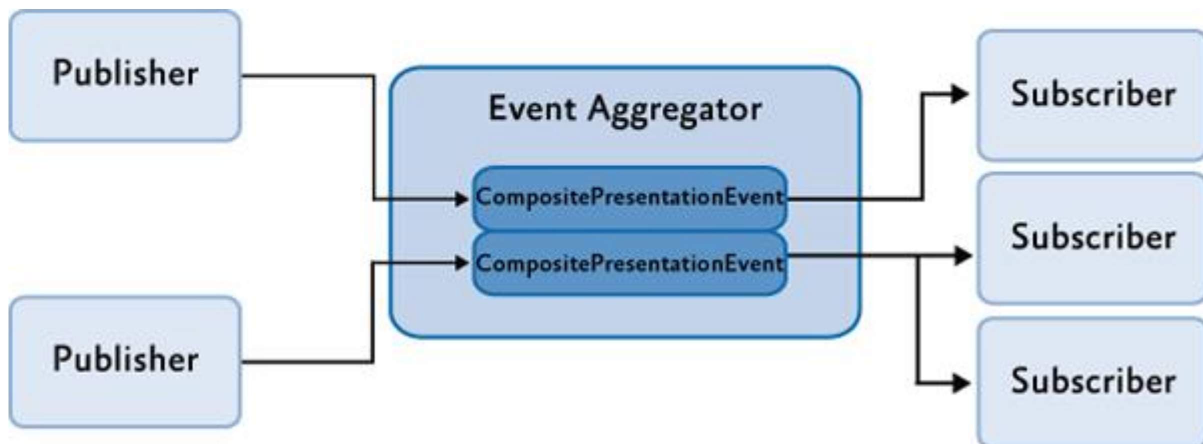
Figure 13. Using RegionContext

3.1.5.3. Делени сервиси / Shared services

Повикувачот повикува метод од сервис кој што подигнува настан (event) кон примачот. Се користи кога претходните два начини не може да се користат.

3.1.5.4. Собирање на ивенти / Event aggregation

Се користи за комуникација помеѓу view моделите, презентерите или котролерите кога не се очекува директна реакција на акцијата. Prism библиотеката овозможува ивент механизам за комуникација помеѓу лабаво врзаните компоненти во апликацијата. Овој механизам е базиран на event сервис за агрегација (собирање) и дозволува издавачите и претплатниците да комуницираат преку ивенти и притоа да немаат меѓусебна директна референца.



Слика 14. Собирач на настани (Event aggregator)

Figure 14. Event aggregator

3.1.6. Составувањето на корисничкиот интерфејс / Composing the User Interface

Корисничкиот интерфејс т.е. изгледот на апликациите може да биде изграден со користење на некое од следниве методологии:

- Сите потребни контроли за составување на формата се содржани во една Extensible Application Markup Language (XAML) датотека, и составувањето на формата е во времето на дизајнирање.
- Логичките области од формата се поделени во различни делови, обично кориснички контроли. Деловите се директно референцирани од формата, која пак е составена во времето на дизајнирање.
- Логичките области од формата се поделени во различни делови, обично кориснички контроли. Деловите се непознати за формата и динамички се додаваат во време на извршување. Апликациите која ја користат оваа методологија се нарекуваат композитни апликации.

Корисничкиот изглед на една композитна апликација е составен од лабаво поврзани визуелни компоненти т.н. view кои обично се во составен дел на апликациските модули но не е задолжително потребно да бидат. Ако ја поделиме некоја апликација на модули, треба да се најде начин и Корисничкиот интерфејс да биде лабаво поврзан, но исто така може да се избере овој пристап на градење дури и ако View-ата не се во самите модули. За крајниот корисник сето ова не треба да прави разлика поради тоа што за него апликацијата треба да изгледа и да е комплетно функционално интегрирана. [18]

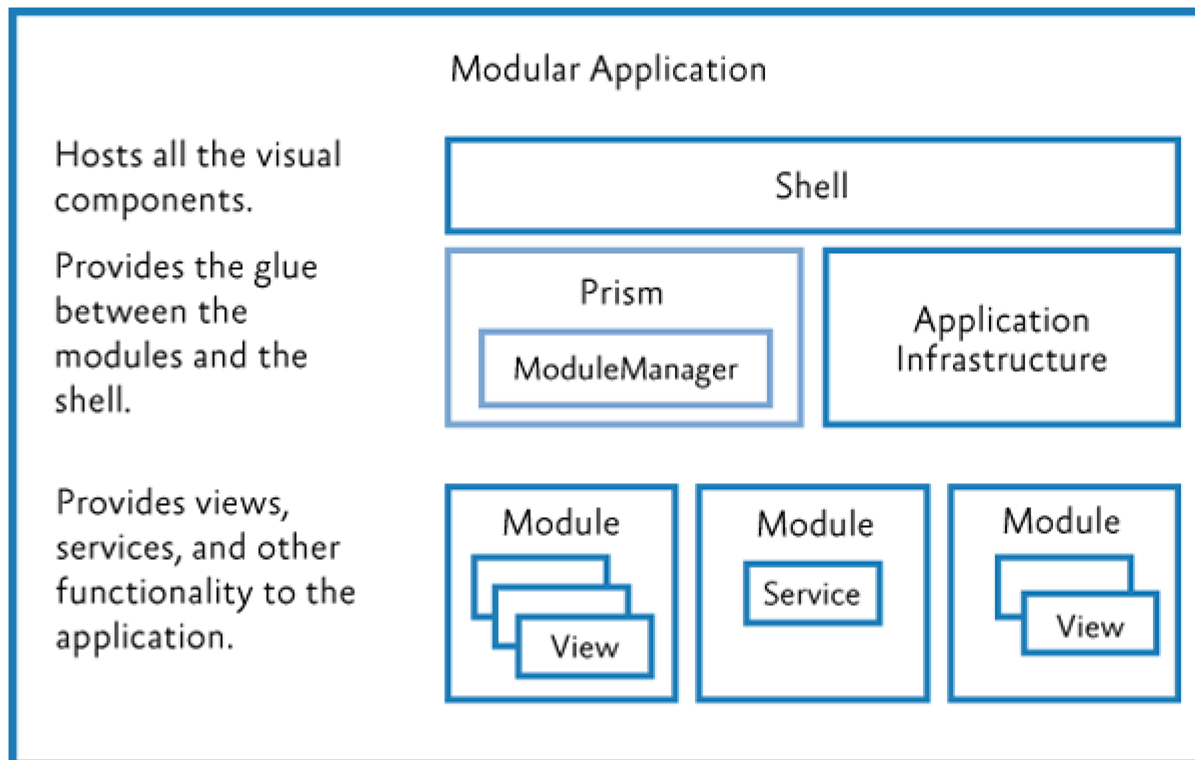
За да биде составен таков кориснички интерфејс, архитектурата која се користи треба да овозможува да се создаде изглед составен од лабаво спарени (поврзани) визуелни елементи кој би се создал во времето на извршување на апликацијата. Дополнително архитектурата треба да овозможи начин да тие визуелни елементи меѓусебно лабаво комуницираат.

3.1.7. Развој на модуларни апликации

Модуларна апликација е тип на апликација која е поделена на множество од функционални единки (наречени модули) кои можат да бидат интегрирани во поголема апликација. Модулот опфаќа дел од вкупната функционалност на апликацијата и најчесто претставува сет од сродни задачи/одговорности. Може да вклучува и збир на сродни компоненти, функции или карактеристики на апликацијата вклучувајќи кориснички интерфејс и бизнис логика, дел од инфраструктурата на апликацијата како на пример сервиси на апликациско ниво за најава и автентикација на корисници. Модулите се независни едни од други но можат да комуницираат помеѓу себе во лабава врска (со лабаво спарување). Модуларните апликации во основа се по лесни за развој, тестирање, поставување во функција/употреба, како и за надградување.

Како пример може да земеме апликација за персонално банкарство. Корисникот може да пристапи до различни функции, како што се трансфер на средства помеѓу сметки, плаќање сметки, ажурирање на лични информации преку еден ист кориснички интерфејс. Како и да е во позадина, секоја од овие функции е врзана за одреден модул. Модулите комуницираат помеѓу себе и со системот во позадина (сервери со бази на податоци, веб сервиси и сл.). Сервисите на апликацијата ги интегрираат разните компоненти кои се составен дел на различните модулите и управуваат со комуникацијата со корисникот. Крајниот корисник гледа една интегрирана апликација (едно view) која изгледа баш исто како и монолитните апликации.

На слика 15 е прикажан дизајн на модуларна апликација составена од неколку модули.



Слика 15. Дизајн на модуларна апликација составена од неколку модули

Figure 15. Design of Modular application composed of modules

Придобивки од градењето на модуларни апликации

Најверојатно повеќето од нас користат добро дизајнирани апликации кои користат асембleја, интерфејси, класи и користат проверени објектно-ориентирани принципи. Дури и така, освен ако не се води голема грижа, дизајнот на апликациите може пак да биде монолитен (сите функционалности се имплементирани во цврста врска со апликацијата), што може да доведе до тоа да апликацијата ја направи тешка за развивање, тестирање, надградување и одржување.

Модуларниот пристап на развој на апликациите од друга страна може да служи за одделување на парчиња од функционалности од апликацијата која треба да се изгради и дозволува тие функционалности да бидат независно развиени и тестирани. Ова го прави целиот процес на развој и тестирање полесен, но исто така ја прави апликацијата по флексибилна и поедноставна за до развивање во иднина. Придобивката од користењето на модуларниот пристап е тоа што ја прави целата апликациска архитектура по флексибилна и полесно одржлива затоа што дозволува разделување на истата до најмали парчиња со кои лесно може да се управува. Секое парче опфаќа точно одредена функционалност, и секое парче е интегрирано преку јасни но лабаво спарени (врзани) комуникациски канали.

Поддршката која Prism ја овозможува за развој на модуларни апликации

Prism овозможува поддршка за развој на модуларни апликации и за менаџмент на модулите во време на извршување на апликацијата. Користењето на функционалноста на Prism за развој на модуларни апликации може да заштеди многу време затоа што не мора од корен да градиме сопствен framework за развој на модуларни апликации ако веќе сме решиле да развиваме модуларни апликации.

Prism ги поддржува следните можности за развој на модуларни апликации:

- Поседува каталог на модули за регистрирање на именувани модули и локацијата на секој од нив. Каталогот на модули може да биде направен на еден од следните начини:
    - Дефинирање на модулите во код или во Extensible Application Markup Language (XAML).
    - За Windows Presentation Foundation (WPF) преку откривање на модули во директориум од каде може да се вчитаат сите достапни модули без притоа експлицитно да се дефинираат во каталог.
    - За WPF со дефинирање на модулите во датотека за конфигурација .
- Користи декларативни мета-податочни атрибути за модулите за да ја овозможи иницијализацијата и задоволување на зависностите на модулите.
- Интеграција со dependency injection контејнерите за да овозможи лабаво спарување/врзување помеѓу модулите.
- За вчитување на модулите:
    - Менаџмент на зависности, вклучувајќи откривање на дупликати и циклуси за да осигури дека модулите се вчитани по правилен редослед и се вчитани и иницијализирани само еднаш.
    - Преземање на модули по потреба / по барање и преземање во позадина за да се минимизира времето за стартување на апликацијата. Модулите може да бидат преземени и иницијализирани или во позадина или кога ќе бидат потребни.

Основни концепти

Во овој дел ќе ги претставиме основните концепти поврзани со модуларноста во Prism, вклучувајќи ги тука IModule интерфејсот, процесот на вчитување на модули, каталогот на модули, комуникацијата помеѓу модули и dependency injection контејнерите.

IModule: Основниот градбен блок на модуларните апликации

Модул претставува логички збир на функционалности и ресурси кои се спакувани на таков начин да можат да бидат засебно развиени, тестирани, поставувани во употреба и интегрирани во апликација. Пакет може да биде едно или повеќе assembly-ја, или како слободна колекција или во комплет заедно во XAP датотека. Секој модул има една централна класа која е одговорна за иницијализација на модулот и интегрирање на неговата функционалност во апликацијата. Таа класа го имплементира IModule интерфејсот. Постоењето на класата која го имплементира IModule интерфејсот е доволно за пакетот да може да биде идентификуван како модул. IModule интерфејсот има еден метод наречен Initialize, во кој што се имплементира целата логика за иницијализација и интеграција на функционалноста на модулот во апликацијата. Зависно од намената на модулот, тој може да регистрира view-а во композитниот кориснички интерфејс, да додава сервиси на апликацијата или проширува функционалностите на апликацијата. Следниов код прикажува минималистичка имплементација на модул.

C#

```
public class MyModule : IModule
{
   public void Initialize()
   {
      // Do something here.
   }
}
```

Пример код 14.

Животен циклус на модулите / Module Lifetime

Процесот на вчитување на модулите во Prism го вклучува следното:

1. Регистрирање/откривање на модулите

Модулите кои треба да се вчитаат во одредена апликација за време на нејзиното извршување се дефинирани во каталогот на модули. Каталогот содржи информации за модулите кои треба да се вчитаат, нивната локација и редоследот по кој треба да се вчитаат.

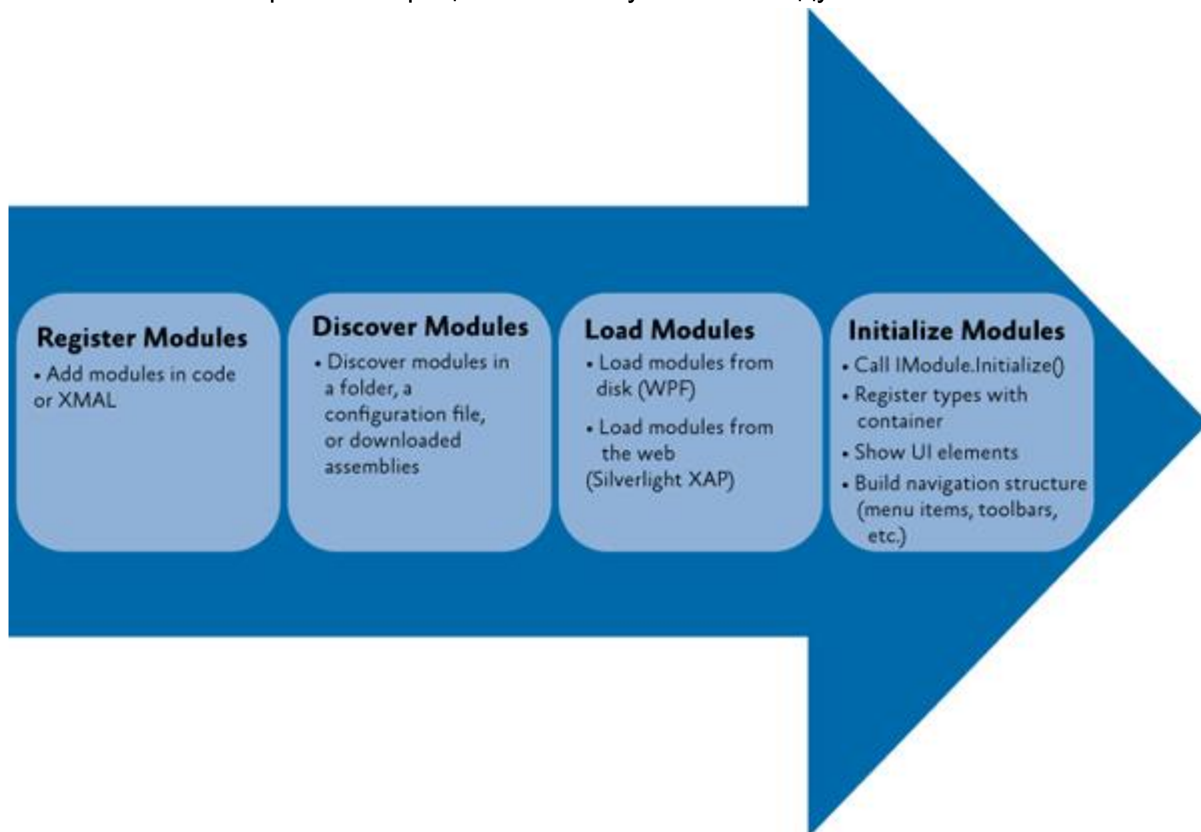
2. Вчитување на модулите

Асемблејата кои ги содржат модулите се вчитуваат во меморија и во оваа фаза може да е потребно модулите да бидат преземени од веб или на некој начин преземени од некоја локација локална или оддалечена.

3. Иницијализација на модулите

Потоа модулите се иницијализираат. Тоа значи дека се создаваат инстанци од класата на модулот и се повикува **Initialize** методот преку **IModule** интерфејсот.

На слика 16 е прикажан процесот на вчитување на модулите.



Слика 16. Процесот на вчитување на модули

Figure 16. Module loading process

Каталогот на модули / Module Catalog

Каталогот на модули (ModuleCatalog класата) чува информации за модулите кои може да бидат користени во апликацијата. Каталогот претставува збирка од ModuleInfo класи. Секој модул е опишан преку ModuleInfo класа во која се чуваат податоците поврзани со модулот (атрибутите на модулот) меѓу кои податоци за името, типот, локацијата. Има неколку начини за пополнување на ModuleCatalog со инстанци од ModuleInfo :

- регистрирање на модулите преку код;
- регистрирање на модулите во XAML;
- регистрирање на модулите во датотека за конфигурација (само за WPF);
- откривање на модулите во локален директориум (само за WPF).

Механизмот за откривање и регистрирање кој што ќе се користи зависи од потребите на апликацијата. Користењето на датотека за конфигурација или XAML датотека дозволува да апликацијата нема потреба да чува референци до модулите. Додека кога се користи регистрирање со откривање на модулите во локален директориум дозволува користење на модулите без истите да бидат некаде специфицирани, во некоја форма на датотека.

Контролирање на процесот кога да се вчита модулот / Controlling When to Load a Module

Prism апликациите може да ги иницијализираат модулите, најбрзо штом тие станат достапни или кога апликацијата има потреба од нив и ги повика. Во Silverlight апликациите модулите може да бидат преземени заедно со апликацијата или во позадина откога апликацијата е веќе стартувана. Да ги разгледаме ги следниве упатства за вчитување модули:

- модулите кои се потребни за апликацијата да започне да се извршува мора да бидат преземени заедно со неа и иницијализирани кога таа ќе се стартува;
- модулите кои содржат функции кои скоро секогаш се користат во апликацијата може да бидат преземени во позадина и иницијализирани кога ќе станат достапни;
- модулите кои содржат функции кои ретко се користат може да бидат преземени во позадина и иницијализирани кога ќе бидат потребни.

Треба добро да се размисли како една апликација ќе се подели на модули, треба да се земат предвид најчестите сценарија на употреба, стартувањето на апликацијата, бројот и големината на преземањата кои треба да се направат за да се одреди како ќе се конфигурираат модулите за преземање и иницијализација.

Интегрирање на модулите во апликацијата

Како што веќе е образложено во претходниот дел од презентацијата на Prism библиотеката, во неа се вклучени две класи за стартување (bootstrap) на апликациите UnityBootstrapper и MefBootstrapper. Овие класи може да се користат за создавање и конфигурирање на менаџерот на модули за да ги открие и вчита модулите. Конфигурацискиот метод за регистрирање на модулите специфициран во XAML датотека, датотека за конфигурација или локален директориум може да биде преклопен со неколку редови на код.

Initialize методот кој го содржи модулот се користи за модулот да се интегрира со остатокот од апликацијата. Начинот на којшто ова може да се изведе варира од случај до случај, зависно од структурата на апликацијата а и од содржината на модулот. Следните работи се основните што треба да се направат за модулот да се интегрира во апликацијата:

- Се додава view од модулот на структурата за навигација на апликација. И тоа најчесто кога се гради композитен кориснички интерфејс за апликацијата со користење на метод на пронаоѓање на view-ата или вметнување на view-ата.
- Се претплатува на event-и и сервиси од апликациското ниво.
- Се регистрираат сервисите кои модулот ќе ги споделува во dependency injection контејнерот.

Комуникацијата помеѓу модули

И покрај тоа што модулите треба да се лабаво спарени помеѓу себе, сепак нормално е тие да комуницираат меѓусебно. Постојат неколку начини за лабаво спарена (loosely coupled) комуникација, и секој од нив си има свои предности. Најчесто се користи комбинација од овие начини. Дел од начините за остварување на лабаво спарената комуникација се следните:

- Лабаво спарени ивенти / Loosely coupled events

Модулот може да емитува одреден event кој настанал. Другите модули може да се претплатат (subscribe) на тие event-и така што ќе бидат известени кога одреден event се случил. Користењето на лабаво спарените event-и е најлесен начин за поставување на комуникацијата помеѓу два модули, затоа лесно и се имплементира. Од друга страна, таков дизајн кој целосно се потпира на event-и може да стане тежок за одржување, посебно ако многу event-и треба да бидат организирани за да исполнат одредена задача. Во таков случај попаметно е да се користат делени сервиси.

- Делени сервиси / Shared services

Деленииот сервис е класа која може да биде пристапена преку заеднички интерфејс. Најчесто се наоѓаат во делено assembly и овозможуваат сервиси за целиот систем, како што се автентикација, најавување и конфигурирање.

- Делени ресурси / Shared resources

Ако не сакаме модулите директно да комуницираат помеѓу себе, може да се изведе тие да комуницираат индиректно преку делен ресурс, како што е база на податоци или сет од веб сервиси.

Dependency Injection и модуларни апликации

Контејнери како Unity Application Block (Unity) и Managed Extensibility Framework (MEF) дозволуваат лесна употреба на Inversion of Control (IoC) и Dependency Injection, кои што се моќни дизајн шаблони кои помагаат лабаво да се спарат компонентите. Дозволуваат компонентите да побараат референца од други компоненти од кои што зависат без при тоа тие референци да се „hard code"-ирани (внесени во кодот), и со тоа промовираат подобро реискористување на кодот и подобрена флексибилност на апликацијата. Dependency injection е многу корисен кога се гради модуларна апликација составена од лабаво спарени компоненти. Prism е така дизајниран да не е свесен за постоењето на dependency injection контејнерот кој се користи за составување на компонентите внатре во апликацијата. Изборот за тоа кој dependency injection контејнер ќе се користи останува на дизајнерот на апликацијата, да одреди спрема потребите кои треба да се задоволат. Во секој случај двете основни dependency injection framework-а од Microsoft се Unity и MEF.

Unity Application Block нуди целосно опремен dependency injection контејнер. Тој поддржува вметнување преку својства (property-based), преку конструктор (constructor-based) и преку policy, што дозволува транспарентно вметнување на зависностите, однесувањата и политиката (policy) помеѓу компоненти, а исто така поддржува и други функционалности карактеристични за еден dependency injection контејнер.

MEF (кој е дел од .NET Framework 4 и Silverlight 4) овозможува поддршка за градење на надградливи .NET апликации со тоа што има dependency injection базиран механизам за составување на компонентите и овозможува и други функционалности кои се поддршка за развојот на модуларни апликации. Една од најјаките страни му е тоа што овозможува апликацијата да открие компоненти во време на извршување и да ги интегрира тие компоненти во апликацијата со лабаво поврзување (спарување). MEF е исто така важна работна рамка (framework) за надградување и составување на апликации и во себе вклучува откривање на assembly-ја и типови, разрешување на зависности на типови, dependency injection (вметнување зависности), и некои многу практични можности како што се преземањето на XAP и assembly-ја.

Prism овозможува искористување на овие можности на MEF, меѓу кои следните:

- поврзување на типовите на модули со нивната XAP локација;
- регистрирање на модули преку XAML и атрибути преку код и за WPF и Silverlight;
- регистрирање на модули преку конфигурациски датотеки и пребарување на директориуми за WPF;
- следење на состојбата (State tracking) при вчитување на модулот.

- сопствени декларативни мета-податоци за модулите кога се користи MEF;
- и двата Unity и MEF dependency injection контејнерите работат беспрекорно со Prism.

Клучни одлуки

Постојат мноштво придобивки од развојот на апликациите во модуларна форма изнесени во оваа теза, но мора да постои посветеност во смисла на време и напор да тие придобивки реално да бидат искористени. Ако веќе сме се одлучиле да развиваме модуларна апликација има уште неколку работи што треба да ги имаме на ум и тоа:

- Точно да го определиме framework-от кој ќе се користи, Prism, MEF или некој друг.

- Треба да одлучиме како ќе го организираме решението. Пристапот кај модуларната архитектура е најпрвин да ги одредиме границите на секој модул, вклучувајќи ги и assembly-јата кои ќе бидат дел од секој модул. Може да одлучиме да користиме модулација на апликацијата со цел да го олесниме развојот, да имаме подобра контрола врз тоа како апликацијата ќе биде изготвена или како поддршка на „plug-in “ компонентите и надградливата архитектурата.

- Да се одлучи како ќе се поделат модулите. Модулите можат да бидат поделени врз база на потребите, на пример по припадност на функционални делови, по тоа кога се потребни апликацијата да ги вчита, по тоа на кој тим се доделени за развивање, по итност за ставање во употреба итн.

- Да се определат централните сервиси кои што апликацијата ги овозможува за сите модули. Пример за таков сервис е сервис за известување за настанати грешки или сервис за автентикација и авторизација.

- Ако користиме Prism треба да се одреди кој пристап ќе се користи за регистрација на модулите во каталогот на модули. За WPF може модулите да се регистрираат во код, во XAML, во датотека за конфигурација или со откривање на модулите во локален директориум. За Silverlight регистрирањето на модулите може да е во код или во XAML.

- Треба да се одреди начинот на комуникација помеѓу модулите и стратегија за меѓусебните зависности. Модулите ќе мора да комуницираат меѓусебно и ќе мора да се решаваат зависностите помеѓу нив.

- Да се определи кој dependency injection контејнер ќе се користи. Модуларните системи имаат потреба од вметнување на зависности (зависни елементи), инверзија на контрола (inversion of control) или локатор на сервиси за да овозможи лабаво спарување и динамичко вчитување и создавање на модулите. Prism го

остава изборот на Unity, MEF, или друг тип на контејнер и соодветни библиотеки за Unity или MEF-базирани апликации.

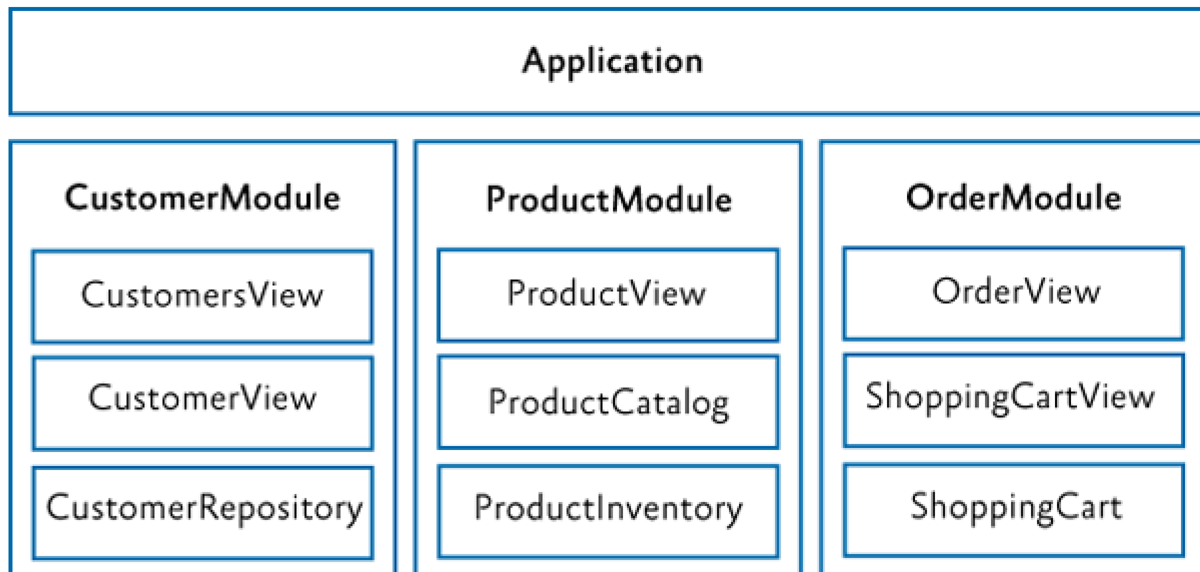
- Минимизирање на времето потребно за стартување на апликацијата. Треба да размислиме за преземање на модулите по потреба или во позадина за да се намали времето за стартување на апликацијата.

- Да се одредат потребите за пуштање во употреба. Треба да се размисли за тоа како и каде планираме да ја пуштиме апликацијата во употреба. Ова може да има ефект врз перформансите на самата апликација.

Поделба на апликацијата во модули

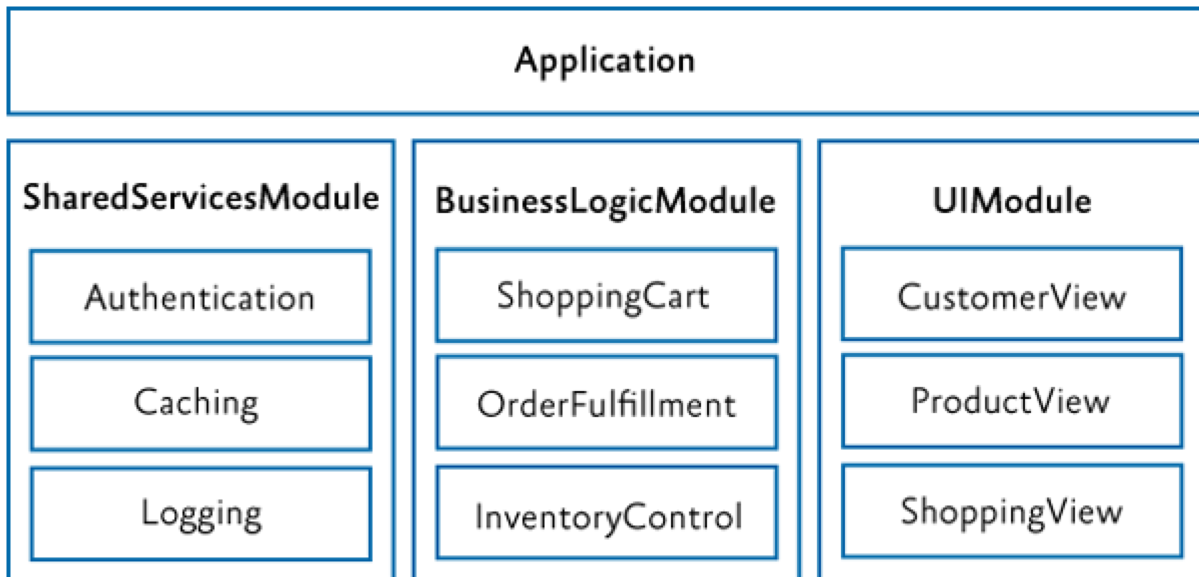
Кога развиваме апликација во модуларен стил ја структурираме апликацијата во засебни клиентски модули кои што може засебно да се развијат, тестираат и постават во употреба. Секој модул опфаќа дел од функциите кои ги има апликацијата. Една од првите одлуки која треба да се донесе е да се одлучи како ќе се врши поделбата на функционалностите на апликацијата во дискретни модули.

Еден модул треба да опфати дискретно множество на сродни одговорности/задачи и да има одреден сет на одговорности. Модул може да претставува дел од вертикалната поделба на апликацијата или сервис од хоризонталната поделба. Обемните апликации може да ги содржат и двата типа на поделба т.е. и двата типа на модули.



Слика 17. Апликација со модули кои се делови од вертикална поделба

Figure 17. An application with modules organized around vertical slices



Слика 18. Апликација со модули организирани во хоризонтални слоеви

Figure 18. An application with modules organized around horizontal layers

Поголемите и пообемни апликации може да имаат модули организирани и како вертикални делови и како хоризонтални слоеви. Ќе наведеме неколку примери на модули:

- модул кој содржи специфична функционалност на апликацијата, пример модул за вести и известувања од социјални медиуми во апликација за водење маркетинг;
- модул кој содржи некоја под систем од функции, пример модул за печатење фактури во апликација за сметководство;
- модул кој содржи инфраструктурни сервиси како најавување, кеширање, авторизациски сервиси, веб сервиси и сл.

Еден модул треба да има што е можно помало множество од зависности т.е. други модули. Кога некој модул има зависност (зависи од друг модул) тогаш тие треба да бидат лабаво спарени со користење на интерфејси дефинирани во некоја делена библиотека наместо со конкретни типови или со користење на EventAggregator за комуникација со другите модули преку EventAggregator event типовите.

Целта на модулирањето е да ја раздели апликацијата на таков начин да таа стане/остане флексибилна, лесно одржлива и стабилна дури и кога функционалности и технологии се додаваат и одземаат. Најдобриот начин да се постигне целта е да се дизајнира апликацијата на таков начин да модулите бидат што понезависни, да имаат добро дефинирани интерфејси и да се што е можно повеќе изолирани.

Одредување на соодносот на проекти и модули

Има неколку начини како се создаваат и пакуваат модулите. Препорачан и најчесто употребуван начин е да се создава едно assembly (еден проект) за еден модул. Овој начин на работа помага да се сочуваат логички модулите одвоени и поттикнува тие соодветно и изолирано да се чуваат. Исто така самото размислување потоа за assembly-то природно доаѓа како тоа да е модулот, и во смисла на развивање и во смисла на пуштање во употреба. Но секако ништо не не спречува да сместиме повеќе од еден модул во едно assembly, во некои случаи тоа е практично ако сакаме да го намалиме бројот на одделни проекти во решението (Visual Studio Solution). За обемни апликации нормално е да имаме 10-15 модули и нивното одделување во засебни проекти во рамки на решението може да го усложни работењето и евентуално да го забави и намали перформансите на самото Visual Studio. Некогаш може да има смисла (кај обемни апликации) еден модул или сет од модули да бидат развиени во посебни решенија и пак на тој начин би се држеле до принципот еден модул во едно assembly т.е. еден проект.

XAP и прераспределба на модули / XAP and Module Factoring

Во Silverlight апликациите модулите се вообичаено спакувани во одделни XAP датотеки, но во некои случаи и повеќе од еден модул по XAP. Треба да се има во предвид колку XAP датотеки ни требаат со цел да се намали бројот и големината на преземања потребни да се направат за стартување на апликацијата или за овозможување на некоја нејзина функција. Ако избереме да го одвоиме секој модул во засебен проект/assembly, треба да одлучиме дали тоа assembly ќе има свој XAP кој ќе биде поставен во употреба или ќе сместиме неколку assembly/модули во ист XAP.

Некои од факторите кои влијаат на одлуката дали да вклучиме повеќе модули во еден XAP или да ги одвоиме се следните:

- Големината на датотеката која се презема и делените зависности. Секоја XAP датотека содржи во себе и мал дел од дополнителни податоци (manifest) и од .zip пакувањето. Исто така ако има зависности помеѓу модулите и тие не се одвоени од зависниот модул или од некоја кеширана библиотека, секој XAP ќе ги вклучи во себе овие зависни библиотеки, што може значајно да ја зголеми датотеката која се презема.

- Временската рамка (timing-от) кога повеќе модули и се потребни на апликацијата. Ако повеќе модули се преземаат и користат во исто време, како што се view-ата кои се појавуваат при стартување на апликацијата, нивното пакување во единствен XAP ќе доведе до тоа да тие бидат побрзо преземени и да бидат физички достапни на клиентот исто времено. Модуларната карактеристика на Prism осигурува да модулите кои имаат дополнителни зависности бидат преземени по правилен редослед, така што не може да се случи модулите да бидат вчитани по неправилен редослед кога тие се сместени во повеќе XAP датотеки. Но важно е да се спомене дека различно е да се направат две преземања наместо едно дури и

ако вкупната големина што треба да се преземе е иста, перформансите ќе се намалат во случајот кога имаме две засебни преземања.

- Водење на верзии на модулите / Module versioning. Ако различни модули се развиваат временски независно, а можеби и пуштени во употреба засебно, логично е дека ќе бидат сместени во различни XAP датотеки кои би ги воделе со маркер на верзија за да можат полесно и појасно надградувани.

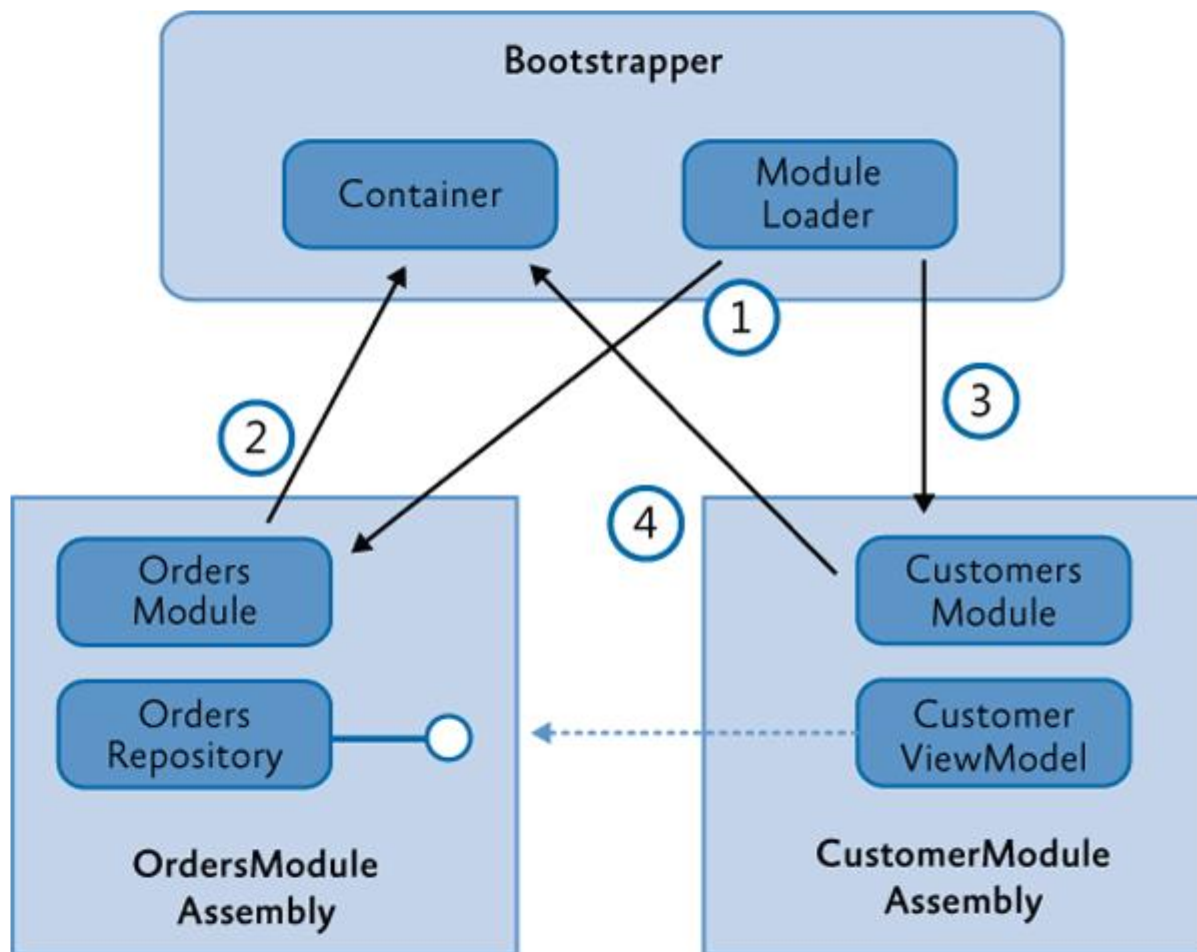
За да се избегне преземање на исти асемблеја повеќе од еднаш преку различни XAP датотеки, постојат два пристапи и тоа:

- да се пренесат делените зависности (зависни модули) во посебен инфраструктурен модул и другите модули да ги земаат зависностите од тој делен модул;

- да се користи Assembly Library Caching во Silverlight за да се сместат делените типови во делена библиотека која ќе се презема еднаш и ќе се кешира од страна на Silverlight наместо од Prism.

Користење на Dependency Injection за лабаво спарување / Dependency Injection for Loose Coupling

Модулот може да има зависности (компоненти или сервиси) кои му се ставаат на располагање од страна на апликацијата домаќин или од другите модули. Prism ја поддржува можноста за регистрирање на зависностите помеѓу модулите така што тие да можат да бидат преземени и иницијализирани по правилен редослед. Prism исто така го овозмоѓува иницијализирањето на модулите кога тие ќе бидат преземени во апликацијата. За време на иницијализацијата модулот може да добие референци до други компоненти и сервиси што му се потребни и/или да регистрира компоненти и сервиси кои ги содржи за да ги направи достапни за користење за другите модули. Секој модул треба да има независен механизам за да ги добива инстанците од надворешните интерфејси наместо директно да инстанцира одреден тип, на пример со користење на dependency injection контејнер. Dependency injection контејнерите како што се Unity или MEF дозволуваат одреден тип автоматски да ги добие инстанците од интерфејсите и типовите кои му се потребни преку dependency injection (вметнување на зависностите). Prism е интегриран со Unity и MEF и така овозможува модулите едноставно да ја го користат вметнувањето на зависностите. На следниот дијаграм (слика) е прикажана типична секвенца од операции кога модулите се вчитуваат и имаат потреба да регистрираат референци до други компоненти и сервиси.



Слика 19. Пример за вметнување на зависности (dependency injection)

Figure 19. Example of dependency injection

Во примерот на слика 19, во OrdersModule assembly е дефинирана OrdersRepository класа (заедно со view и класи кои ја имплементираат функционалноста). Во CustomerModule assembly е дефинирана CustomersViewModel класата која зависи од OrdersRepository, преку интерфејс изложен/овозможен од сервисот. За време на стартување на апликацијата и bootstrapping процесот се прават следните чекори:

1. Bootstrapper го стартува процесот на иницијализација на модулите и вчитувачот на модули (module loader) го вчитува и иницијализира OrdersModule модулот.

2. Во иницијализацијата на OrdersModule го регистрира OrdersRepository во контејнерот.

3. Вчитувачот на модули потоа го вчитува CustomerModule. Редоследот на вчитување на модулите може да се специфицира преку зависностите во мета податоците на модулот.

CustomersModule прави инстанца од CustomerViewModel разрешувајќи ја преку контејнерот. CustomerViewModel има зависност (зависи) од OrdersRepository и тоа го укажува преку конструкторот или со вметнување на својства (property injection). Контејнерот ја вметнува таа зависност во конструкцијата на view model-от брз основа на типот регистриран од страна на OrdersModule. Крајниот резултат е интерфејс референциран од CustomerViewModel до OrderRepository без притоа тие две класи цврсто да се спарат/поврзат.

Интерфејсот кој се користи за изложување на OrderRespository (IOrderRepository) може да биде сместен во одделен делен сервис assembly или assembly кое ќе ги содржи само интерфејсите сервисите од OrdersModule и типовите потребни да се изложат тие сервиси. На тој начин нема да има воопшто никаква цврста зависност помеѓу CustomersModule и OrdersModule модулите.

НАЈЧЕСТИ СЦЕНАРИЈА

Во овој дел ќе ги опишеме најчестите сценарија/потреби со кои може да се сретнеме кога работиме со модули и модуларни композитни апликации. Овие сценарија вклучуваат дефинирање на модули, регистрирање и откривање на модули, вчитување на модули, иницијализација на модули, специфицирање на зависностите на модулите, вчитување на модули по потреба, преземање на модули од други локации во позадина и откривање кога некој модул е веќе преземен. Модулите може да се откријат и регистрираат преку код во XAML или апликациска конфигурациона датотека или преку скенирање на локален директориум.

Дефинирање на модул во Prism

Во Prism да се потсетиме модулот претставува логички збир на функционалности и ресурси спакувани на таков начин да може да бидат одделено развивани, тестирани, поставени во употреба и интегрирани во апликација.

Таа класа го имплементира IModule интерфејсот како што е прикажано во следниот пример.

C#

```
public class MyModule : IModule
{
   public void Initialize()
   {
```

```
        // Initialize module
    }
}
```

Пример код 15.

Начинот на кој што се имплементира Initialize методот зависи од конкретните потреби на апликацијата која се развива. Типот на класата на модулот, начинот на иницијализација и било кои зависности на модулот се дефинираат во каталогот на модули. За секој модул во каталогот, вчитувачот на модули создава инстанца од класата на модулот и потоа го повикува Initialize методот. Модулите се процесираат според редоследот специфициран во каталогот на модули. Редоследот за иницијализација на модулите во време на извршување на апликацијата се базира врз тоа кога модулите се преземени, достапни и дали се задоволени зависностите.

Зависно од типот на каталог на модули кој што апликацијата ќе го користи, зависностите на модулите можат да бидат поставени или преку користење на декларативни атрибути на класата на модулот или во рамки на каталогот на модули.

Регистрирање и откривање на модули / Registering and Discovering Modules

Модулите кои што апликацијата може да ги вчита се дефинирани во каталогот на модули. Вчитувачот на модули (Module Loader) на Prism го користи каталогот на модули за да одреди кои модули се достапни за да бидат преземени во апликацијата, кога да бидат вчитани и по кој редослед да бидат вчитани.

Каталогот на модули е претставен преку класа која го имплементира IModuleCatalog интерфејсот. Класата на каталогот на модули е создадена од страна на bootstrapper класата на апликацијата за време на иницијализацијата на апликацијата. Prism нуди неколку различни имплементации на каталог на модули. Каталог на модули може да се направи и од некој друг податочен извор (data source) со повикување на AddModule методот или со изведување од ModuleCatalog за да се направи каталог на модули со некои специфични функции.

Вообичаено модулите во Prism користат dependency injection контејнер и заеднички сервис локатор (Common Service Locator) за да добијат инстанци од типовите кои им се потребни за иницијализација. И двата Unity и MEF контејнерите се поддржани во Prism. И покрај тоа што целокупниот процес на регистрација, откривање, преземање и иницијализација на модулите е ист, некои детали се различни врз основа на тоа дали се користи Unity или MEF.

Регистрирање на модули во код

Најосновниот каталог на модули е достапен преку ModuleCatalog класата. Овој каталог на модули може да се користи за регистрирање на модули со програмирање преку специфицирање на типот на класата од модулот. Исто така, може преку код да се специфицира име името на модулот и начинот на иницијализација. За да се регистрира

модулот директно преку ModuleCatalog класата се повикува AddModule методот во Bootstrapper класата на апликацијата. На пример:

C#

```
protected override void ConfigureModuleCatalog()
{
    Type moduleCType = typeof(ModuleC);
    ModuleCatalog.AddModule(
      new ModuleInfo()
      {
         ModuleName = moduleCType.Name,
         ModuleType = moduleCType.AssemblyQualifiedName,
      });
}
```

Пример код 16.

Во претходниот пример модулите се директно референцирани од страна на Обвивката, така што типовите од класата на модулот се дефинирани и можат да бидат кориатени за повикување на AddModule методот. Затоа во наведениот пример се користи typeof(ModuleC) за додавање модул во каталогот.

Ако апликацијата има директна референца кон типот на модулот, може модулот да биде додаден како на примерот преку типот, во друг случај ќе треба да се проследи целосното име на типот и локацијата на assembly.

Bootstrapper класата има CreateModuleCatalog метод кој може да се користи како помошен при создавање на ModuleCatalog. Во основа овој метод создава инстанца од ModuleCatalog, но може да биде преклопен во изведена класа со цел да се создадат различни типови на каталог на модули.

Регистрирање на модули со користење на XAML датотека

Може да се дефинира каталог на модули декларативно со специфицирање во XAML датотека. Во XAML датотеката се специфицира каква класа на каталог на модули да се создаде и кои модули да бидат додадени во каталогот. Вообичаено XAML датотека се додава како ресурс на проектот на Обвивката. Каталогот на модули се создава од страна на bootstrapper-от со повик на CreateFromXaml методот. Гледано од техничка перспектива, овој пристап е многу сличен на дефинирањето на ModuleCatalog во код поради тоа што XAML датотеката едноставно само ја дефинира хиерархијата на објекти што треба да бидат инстанцирани.

Следниов пример претставува XAML датотека која специфицира каталог на модули

Примерот е земен од (XAML ModularityWithUnity.Silverlight\ModulesCatalog.xaml)

XAML

```
<Modularity:ModuleCatalog
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        xmlns:Modularity="clr-
namespace:Microsoft.Practices.Prism.Modularity;assembly=Microsoft.Practices.Prism">
    <Modularity:ModuleInfoGroup Ref="ModuleB.xap" InitializationMode="WhenAvailable">
        <Modularity:ModuleInfo ModuleName="ModuleB" ModuleType="ModuleB.ModuleB,
ModuleB, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
    </Modularity:ModuleInfoGroup>
    <Modularity:ModuleInfoGroup InitializationMode="OnDemand">
        <Modularity:ModuleInfo Ref="ModuleE.xap" ModuleName="ModuleE"
ModuleType="ModuleE.ModuleE, ModuleE, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null" />
        <Modularity:ModuleInfo Ref="ModuleF.xap" ModuleName="ModuleF"
ModuleType="ModuleF.ModuleF, ModuleF, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null" >
            <Modularity:ModuleInfo.DependsOn>
                <sys:String>ModuleE</sys:String>
            </Modularity:ModuleInfo.DependsOn>
        </Modularity:ModuleInfo>
    </Modularity:ModuleInfoGroup>

    <!-- Module info without a group -->
    <Modularity:ModuleInfo Ref="ModuleD.xap" ModuleName="ModuleD"
ModuleType="ModuleD.ModuleD, ModuleD, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null" />
</Modularity:ModuleCatalog>
```

Пример код 17.

ModuleInfoGroups класата нуди погоден начин за групирање на модулите кои се во иста .xap датотека или assembly се иницијализираат на ист начин, или имаат зависности од модули иста група.

Во Bootstrapper класата на аплиакцијата треба да се специфицира дека XAML датотеката е изворот за ModuleCatalog, како што е прикажано во следниот код:

C#

```
protected override IModuleCatalog CreateModuleCatalog()
{
   return ModuleCatalog.CreateFromXaml(new
Uri("/MyProject.Silverlight;component/ModulesCatalog.xaml",
UriKind.Relative));
}
```

Пример код 18.

Регистрирање на модули со користење на датотека за конфигурација

Во WPF возможно е да се специфицираат информациите за модулите во App.config датотека. Предноста од користењето на овој пристап е тоа што оваа датотека не е компајлирана во апликацијата. Ова е корисно затоа што лесно се додаваат и одземаат модули во време на извршување без притоа да треба да се прекомпајлира апликацијата.

Следниов код е пример во кој е прикажана конфигурациска датотека за специфицирање на каталогот на модули. Ако сакаме модулот автоматски да се вчита се поставува startupLoaded="true".

(примерот е земен од XML ModularityWithUnity.Desktop\app.config)

Xml

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="modules"
type="Microsoft.Practices.Prism.Modularity.ModulesConfigurationSection,
Microsoft.Practices.Prism"/>
  </configSections>
  <modules>
    <module assemblyFile="ModularityWithUnity.Desktop.ModuleE.dll"
moduleType="ModularityWithUnity.Desktop.ModuleE, ModularityWithUnity.Desktop.ModuleE,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" moduleName="ModuleE"
startupLoaded="false" />
    <module assemblyFile="ModularityWithUnity.Desktop.ModuleF.dll"
moduleType="ModularityWithUnity.Desktop.ModuleF, ModularityWithUnity.Desktop.ModuleF,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" moduleName="ModuleF"
startupLoaded="false">
      <dependencies>
        <dependency moduleName="ModuleE"/>
      </dependencies>
    </module>
  </modules>
</configuration>
```

Пример код 19.

Во Bootstrapper класата на апликацијата треба да се специфицира дека конфигурациската датотека е изворот за ModuleCatalog. За да се направи ова се користи ConfigurationModuleCatalog како што е прикажано во следниот код:

```
C#
protected override IModuleCatalog CreateModuleCatalog()
{
    return new ConfigurationModuleCatalog();
}
```

Пример код 20.

Без разлика што каталогот на модули и додавањето на модули е изведено на овој начин сепак може да се додаваат модули на ConfigurationModuleCatalog преку код.

Silverlight не поддржува употреба на конфигурациски датотеки. Ако сакаме да употребуваме конфигурациски стил / пристап за Silverlight препорачаниот начин е создавање на сопствен ModuleCatalog што ќе ја чита конфигурацијата од некој веб сервис.

Откривање на модули во директориум

DirectoryModuleCatalog класата овозможува да биде специфициран локален директориум од каде би бил создаден каталогот на модули во WPF. Таквиот каталог на модули ќе го скенира целиот директориум и ќе го пребара барајќи assembly-ја кои содржат модули. За да може овој начин да се користи секој од модулите треба да има декларативни атрибути на класата кои го специфицираат неговото име и зависностите кои тој модул ги има. Следниот пример покажува како еден каталог на модули се полни со методот на откривање на assembly-ја во локален директориум.

C#

```
protected override IModuleCatalog CreateModuleCatalog()
{
   return new DirectoryModuleCatalog() {ModulePath = @".\Modules"};
}
```

Пример код 21.

Оваа функционалност не е подржана во Silverlight, затоа што Silverlight-овиот сигурносен модел (модел за заштита) не дозволува да се вчитуваат assembly-ја од локалниот датотечен систем.

Вчитување на модули

Откога ModuleCatalog е пополнет, модулите се спремни за вчитување и иницијализација. Вчитувањето на модулите значи дека assembly на модулот треба да биде преместено од диск во меморија. Ако assembly-то не постои на диск, тогаш тоа првин треба да се добие таму. Пример за тоа е преземањето на assembly од веб со користење на Silverlight .xap датотеки. ModuleManager е одговорен за координирање на процесите на вчитување и иницијализација.

Иницијализација на модули

Откога модулите ќе бидат вчитани тие треба да се иницијализираат. Ова значи дека се создава инстанца од класата на модулот и се повикува нејзиниот Initialize метод. Иницијализацијата е местото каде треба модулот да се интегрира со апликацијата. При процесот на иницијализација на модулот треба да имаме предвид за следните работи:

- Регистрирање на view на модулот во апликацијата. Ако модулот учествува во составот на корисничкиот интерфејс било преку откривање на viewа (view discovery) или вметнување на view (view injection), тој ќе мора да ги поврзе своите view или view model со соодветен регион. Ова овозможува да view динамички се појавуваат на менија, лентите со алатки (toolbars) и други визуелни региони во рамки на апликацијата.

- Претплатување на event и сервиси од апликациското ниво. Често апликациите поседуваат и изложуваат некои специфички сервиси и/или event коишти модулите ги користат или нудат. Треба да се користи Initialize методот за додавање на функционалностите на модулот на тие ивенти и сервиси од апликациското ниво. На пример апликацијата може да подигне ивент кога се исклучува и модулот можеби има потреба да реагира на таквиот event. Исто така можно е да модулот мора да проследи некои податоци до сервис на апликациско ниво. На пример ако имаме создадено MenuService (сервис кој ќе е одговорен за додавање или отстранување на ставки од менито), Initialize методот е местото каде треба да се изврши корекција на листата на ставки од менито.

- Регистрирање на типови во dependency injection контејнерот. Ако користиме dependency injection шаблон како што се Unity и MEF, модулот може да регистрира типови за апликацијата или за други модули да ги користат. Дополнително може да побара од контејнерот за разреши инстанца од типот што е потребен.

Во основа животниот циклус на инстанците од модулите е краток. После повикувањето на Initialize методот кој се повикува во време на вчитување, референцата кон модулот се ослободува. Ако не воспоставиме јака референца кон инстанцата на модулот, таа ќе биде отстранета од (garbage collector). Ова може да зададе проблеми при debug-ирање ако се претплатуваме на event-и кон модулот, затоа што едноставно модулот може да исчезне.

Специфицирање на зависностите на модулот

Модулите може да зависат од други модули. Ако модулот А зависи од модулот Б, модулот Б мора да биде иницијализиран пред модулот А. ModuleManager води траг на зависностите и ги иницијализира модулите соодветно. Зависно од тоа како сме го дефинирале каталогот на модули, може да се дефинираат зависностите на модулот преку код, конфигурација или XAML.

Специфицирање на зависностите во код

Во WPF апликациите кои ги регистрираат модулите во код или со откривање на модулите во локален директориум, Prism да може да се користат декларативни атрибути кога се создава самиот модул како што е прикажано на следниот пример:

C# (кога се користи Unity)

```
[Module(ModuleName = "ModuleA")]
[ModuleDependency("ModuleD")]
public class ModuleA: IModule
{
   ...
}
```

Пример код 22

Специфицирање на зависностите во XAML

Следниов пример покажува дека модулот „ModuleF“ зависи од модулот „ModuleE“.

XAML

```
<Modularity:ModuleInfo Ref="ModuleF.xap" ModuleName="ModuleF"
ModuleType="ModuleF.ModuleF, ModuleF, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null" >
<Modularity:ModuleInfo.DependsOn>
   <sys:String>ModuleE</sys:String>
</Modularity:ModuleInfo.DependsOn>
</Modularity:ModuleInfo>
```

Пример код 23

Специфицирање на зависностите во конфигурација

Следниов пример на App.config покажува дека модулот „ModuleD“ зависи од модулот „ModuleB“.

XML App.config

```
<modules>
  <module assemblyFile="Modules/ModuleD.dll" moduleType="ModuleD.ModuleD, ModuleD"
moduleName="ModuleD">
    <dependencies>
      <dependency moduleName="ModuleB"/>
    </dependencies>
</module>
```

Пример код 24

Вчитување на модули по потреба / на барање

За да може модулите да бидат вчитани по потреба треба да се специфицира тоа во каталогот на модули со својството InitializationMode поставено на OnDemand. Откога ова ќе се направи мора во кодот да биде дефинирано кога тој да биде вчитан.

Специфицирање на вчитување на модул по потреба во код

Модулот е се специфицира како „модул кој се вчитува по потреба“, како што е прикажано во следниот пример:

C# Bootstrapper.cs

```
protected override void ConfigureModuleCatalog()
{
  Type moduleCType = typeof(ModuleC);
  this.ModuleCatalog.AddModule(new ModuleInfo()
  {
      ModuleName = moduleCType.Name,
      ModuleType = moduleCType.AssemblyQualifiedName,
      InitializationMode = InitializationMode.OnDemand
  });
}
```

Пример код 25

Специфицирање на вчитување на модул по потреба во XAML

Може да се специфицира InitializationMode.OnDemand кога се дефинира каталогот на модули во XAML како што е прикажано на следниот пример:

XAML ModulesCatalog.xaml

```
...
 <Modularity:ModuleInfoGroup InitializationMode="OnDemand">
       <Modularity:ModuleInfo Ref="ModuleE.xap" ModuleName="ModuleE"
ModuleType="ModuleE.ModuleE, ModuleE, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null" />
...
```

Пример код 26

Специфицирање на вчитување на модул по потреба во конфигурација

Може да се специфицира InitializationMode.OnDemand кога се дефинира каталогот на модули во App.config како што е прикажано на следниот пример:

XML App.config

```
...
<module assemblyFile="Modules/ModuleC.dll" moduleType="ModuleC.ModuleC, ModuleC"
moduleName="ModuleC" startupLoaded="false"/>
...
```

Пример код 27

Барање за вчитување на модул по потреба

Откако модулот ќе биде специфициран дека е за „по потреба“ апликацијата може да го повика тој да биде вчитан. Кодот во кој што ќе се започне процесот на вчитување има потреба од референца до IModuleManager сервисот кој е регистриран во контејнерот од страна на bootstrapper.

C#

```
private void OnLoadModuleCClick(object sender, RoutedEventArgs e)
{
   moduleManager.LoadModule("ModuleC");
}
```

Пример код 28

Преземање на оддалечени модули во позадина

Преземањето на модули во позадина се прави откога апликацијата е веќе стартувана или кога корисникот има потреба од намалување на времето за стартување.

Подготовка на модулот за преземање од далечина

Во Silverlight апликациите, модулите се спакувани во .xap датотеки. За да се преземе модул независно од апликацијата, се прави засебна .xap датотека. Може да сместиме неколку модули во иста .xap датотека со цел да се оптимизира бројот на барања за преземања спрема големината на секоја од .xap датотеките.

За секоја .xap датотека треба да се создаде посебен нов Silverlight Application во Visual Studio, затоа што само application проектите продуцираат засебна .xap датотека. Во овие проекти нема потреба од постоење на App.xaml и MainPage.xaml датотеки.

Следење на прогресот на преземање

ModuleManager класата поседува event преку кој апликациите можат да го следат прогресот на преземањето на модулите. Овој event нуди сооднос на бајтовите кои се преземени во однос на оние кои што преостануваат за преземање. Оваа можност може да се користи за визуелно да се прикаже прогресот на корисникот како што модулот (и функционалноста која корисникот ја чека) се преземаат.

C#

```
this.moduleManager.ModuleDownloadProgressChanged +=
this.ModuleManager_ModuleDownloadProgressChanged;
```

Пример код 29.

C#

```
void ModuleManager_ModuleDownloadProgressChanged(object sender,
    ModuleDownloadProgressChangedEventArgs e)
{
   ...
}
```

Пример код 30

Откривање кога модулот е целосно преземен

ModuleManager класата поседува event преку кој апликациите можат да следат кога модулот е вчитан или дали не успеал да се вчита. Може да се добие референца до овој сервис преку dependency injection од IModuleManager интерфејсот.

C#

```
this.moduleManager.LoadModuleCompleted += this.ModuleManager_LoadModuleCompleted;
```

Пример код 31.

C#

```
void ModuleManager_LoadModuleCompleted(object sender, LoadModuleCompletedEventArgs
e)
{
   ...
}
```

Пример код 32

За да се зачуваат апликацијата и модулите лабава врска, во апликацијата треба да се избегнува користењето на овој event за интеграција со апликацијата. Наместо тоа Initialize методот треба да се справи со интеграцијата со апликацијата. LoadModuleCompletedEventArgs содржи IsErrorHandled својство. Ако еден модул не успее да се вчита и апликацијата сака да избегне ModuleManager-от да исфрли exception може само да го постави ова својство со вредност „true“.

Штом еден модул е вчитан и иницијализиран, assembly на модулот не може да биде отстрането. Референцата до инстанцата на модулот нема да се чува од Prism библиотеките, така што инстанцата од класата на модулот може да биде отстранета после иницијализација од страна на garbage collector.

Модули во MEF

Во овој дел само ќе ги издвоиме разликите ако се избере користење на MEF за dependency injection контејнер.

Кога се користи MEF, MefModuleManager се користи од страна на MefBootstrapper. Ги проширува функциите на ModuleManager и имплементира IPartImportsSatisfiedNotification интерфејс за да обезбеди ModuleCatalog-от да е ажуриран кога нови типови се внесени од MEF.

Регистрирање на модули во код со користење на MEF

Кога се користи MEF може да се аплицира ModuleExport атрибутот на класите на модулите за MEF автоматски да ги открива типовите. Следува пример:

C#

```
[ModuleExport(typeof(ModuleB))]
public class ModuleB : IModule
{
   ...
}
```

Пример код 33.

Исто така, може да се користи MEF за откривање и вчитување на модули со користење на AssemblyCatalog класата, којашто може да се користи да ги открие сите извезени класи на модули во некое assembly а AggregateCatalog класата овозможува неколку каталози да бидат комбинирани во еден логички каталог. Во основа MefBootstrapper класата создава инстанца од AggregateCatalog. Потоа можеме да го преклопиме ConfigureAggregateCatalog методот за регистрирање на асемблејата, како што е прикажано во следниов код:

C#

```
protected override void ConfigureAggregateCatalog()
{
   base.ConfigureAggregateCatalog();
   //Module A is referenced in in the project and directly in code.
   this.AggregateCatalog.Catalogs.Add(
   new AssemblyCatalog(typeof(ModuleA).Assembly));

   this.AggregateCatalog.Catalogs.Add(
      new AssemblyCatalog(typeof(ModuleC).Assembly));
}
```

Пример код 34

MefModuleManager имплементацијата на Prism ги чува AggregateCatalog на MEF и ModuleCatalog на Prism синхронизирани, и со тоа дозволува на Prism да открива модули додадени и преку ModuleCatalog и преку AggregateCatalog.

Откривање на модули во директориум со користење на MEF

MEF поседува DirectoryCatalog којшто може да се користи за пребарување асемблеја кои содржат модули (како и други MEF извезени типови) во одреден директориум. Во тој случај вршиме преклопување на ConfigureAggregateCatalog методот за го регистрираме директориумот. Овој пристап е можен само во WPF. За да се користи овој пристап прво треба на сите модули да им се постават атрибутите за име и нивните зависности со користење на ModuleExport атрибутот, како што е прикажано во кодот кој следува. Ова дозволува MEF да може да увезува модули и дозволува Prism да го чува каталогот на модули (ModuleCatalog) ажуриран.

C#

```
protected override void ConfigureAggregateCatalog()
{
   base.ConfigureAggregateCatalog();

   DirectoryCatalog catalog = new DirectoryCatalog("DirectoryModules");
   this.AggregateCatalog.Catalogs.Add(catalog);
}
```

Пример код 35

Специфицирање на зависностите во код со користење на MEF

За WPF апликациите кои користат MEF се користи ModuleExport како што е прикажано во следниов пример:

C#

```
[ModuleExport(typeof(ModuleA), DependsOnModuleNames = new string[] { "ModuleD" })]
public class ModuleA : IModule
{
   ...
}
```

Пример код 36

Од причина што MEF дозволува откривање на модули во време на извршува на апликацијата, може да бидат најдени и нови зависности помеѓу модулите. Иако може MEF да се користи паралелно со ModuleCatalog важно е да се напомене дека ModuleCatalog го валидира синџирот на зависностите кога тие се вчитуваат од XAML или конфигурација (пред било кој модул да е вчитан). Ако некој модул постои во ModuleCatalog и потоа е вчитан со користење на MEF, ќе се користат зависностите од ModuleCatalog и DependsOnModuleNames атрибутот ќе биде игнориран.

Специфицирање на вчитувањето по потреба со користење на MEF

Ако користиме MEF и ModuleExport атрибутот за специфицирање на модулите и нивните зависности може да се користи InitializationMode својството за специфицирање на модулот дека треба да биде вчитан по потреба, како што е прикажано во следниов код:

```
C#
[ModuleExport(typeof(ModuleC), InitializationMode = InitializationMode.OnDemand)]
public class ModuleC : IModule
{
}
```

Пример код 37

Подготовка на модул за оддалечено преземање со користење на MEF

Prism апликациите кои користат MEF ја користат DeploymentCatalog класата за преземање на XAP датотеки и откривање на асемблеја и типови во нив. MefXapModuleTypeLoader го додава секој DeploymentCatalog на AggregateCatalog**.** Ако две различни XAP датотеки се додадени и имаат некое исто делено assembly, истите типови се увезуваат повторно. Ова може да предизвика грешки кога за некој тип е предвидено да биде singleton а се наоѓа во assembly кое се дели помеѓу модулите. Microsoft.Practices.Prism.MefExtensions.dll е пример за такво assembly.

За да се избегнат дупликатите при увезување, во секој проект за модул референците и делените dll-ки треба да бидат поставени како 'Copy Local'=false. Ова значи дека се спречува да тоа assembly биде спакувано во XAP датотеката на модулот. Оваа постапка ја намалува големината на секоја XAP датотека, но она што мора да го осигуриме е постоењето на асемблето кое не е вклучено во одделните модули да биде составен дел на апликациската датотека.

3.1.8. Prism - заклучоци

Prism директно не зависи од MVVM туку само е граден по истиот принцип. Користењето на истиот принцип е во насока на олеснување на одржувањето на посебни кориснички интерфејс модели.

Хоризонтална модулација кај Prism

И покрај тоа што Prism библиотеката не бара директно користење на хоризонтална модулација на проектите, треба да се разгледа користењето на Model - View - ViewModel (MVVM) кога се имплементира функционалност на view. Пример во кој е прикажаната употреба на Model - View - ViewModel (MVVM) во пример проектите на Prism е Stock Trader RI каде е имплементирано прецизно одделување на логиката на view и неговиот изглед.

Карактеристични имплементации во Prism библиотеката

Нов тип на events поради тоа што постојните не можат да овозможат (loosely coupled) лабава врска помеѓу publisher и subscriber. Ако се покрене event не е важно дали некоја компонента го очекува или не, ако има компонента која ќе го препознае и евентуално реагира по него во ред ако не нема да има никаков проблем. Кај командите (Commands) мора да има компонента од корисничкиот интерфејс која референцира до нив и во основа тоа е генералната разлика помеѓу командите (commands) и настаните (events).

3.2. Windows Phone архитектура во корелација со Prism

3.2.1. Prism за Windows Phone 7

Prism во себе содржи мала библиотека наречена Prism Library for Windows Phone 7, која претставува ограничена верзија на главните функционалности на Prism библиотеката и е специјално направена за да им помогне на програмерите во некои чести случаи кога развиваат апликации за Windows Phone 7. Таа лимитирана библиотека содржи класи кои на програмерите им помагаат да имплементираат команди, навигација, нотификации, интеракција со известувања, интеракција со апликациското ниво итн.

Windows Phone 7 апликациите кои се развиени со користење на Silverlight се природно правени според MVVM шаблонот, што го обесхрабрува користењето на код во позадина на самите view-а и наместо тоа да се користи имплементација во view model-от за справување со event-и. Но и покрај тоа постојат случаи каде тоа е многу тешко да се изведе пр. врзување на команди за објекти од интерфејсот, поврзување на методи со копчиња од апликацијата, известувања (нотификации) за промени на својства на објекти, откривање на промени на текст базирани контроли кои се наоѓаат во view-то а без притоа да се користи директно код во позадина на view-то. Помошните класи и компоненти на Prism библиотеката за Windows Phone 7 се специјално дизајнирани за да ги упростат овие задачи при развој на Silverlight апликации за Windows Phone 7.

Дополнително Prism библиотеката за Windows Phone 7 содржи и помошни класи за објавување и претплата (publishing/subscribing) на ивенти, прикази на пораки со известувања и сл. Пример за користење на овие помошни класи има во Tailspin.PhoneClient проектот кој може да се најде на CodePlex [19].

Prism библиотеката за Windows Phone 7 постои во изворна форма во два проекти Microsoft.Practices.Prism и Microsoft.Practices.Prism.Interactivity во рамки на Tailspin решението. Решението ја имплементира Prism библиотеката за Windows Phone 7, и во рамки на решението постојат и неколку тест проекти кои ги демонстрираат можностите на библиотеката.

3.2.2. Цели во развојот на Windows 8 и Windows Phone 8

Главна цел во дизајнирањето и развојот на Windows 8 и Windows Phone 8 е минимизирање на потрошувачката на енергија на уредите а во исто време максимизирање на времето на реакција на сервисите и услугите.

Апликациите имаат голем ефект врз потрошувачката на енергија на уредите преку директното искористување на ресурсите на уредот: процесор, меморија, диск итн., а секој од овие ресурси си има своја енергетска побарувачка. Идејата која е вметната при дизајнот на Windows 8 е тоа да се дозволи на апликациите да ги искористуваат ресурсите кои се на нивно располагање кога активно работат и им требаат, но да се редуцира на минимум искористувањето на ресурсите кога апликацијата не се користи активно, т.е. се прави суспензија на апликацијата.

Суспензијата на неактивните апликации се однесува за на Metro style апликациите а не и на базичните функции на оперативниот систем како што се копирање на датотеки и сл.

Базичното правило врз кое е поставена суспензијата на апликациите во Windows 8 е ако апликацијата не е на екранот тогаш не треба да влијае на потрошувачката на енергија, т.е. да ја празни батеријата на уредот. Но сепак воведувањето на суспензијата на апликациите не значи спречување на функционирањето на multi-tasking начинот на работа, напротив претставува само нов принцип кој што има за цел да извлече минимална потрошувачка на уредите кои го користат оперативниот систем а кои во голем процент се преносни.

Генерално апликациите можат да бидат во 3 состојби:

- активно да работат на екран и бидат во преден план;
- да бидат суспендирани во позадина;
- да извршуваат некоја позадинска активност.

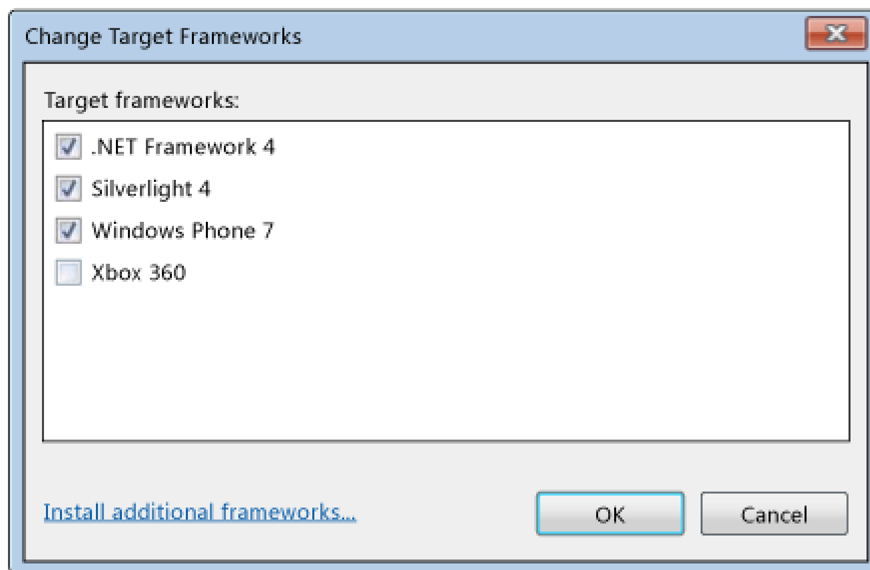
Windows Metro style - развој на апликации

Придобивка е што самите апликации се описни т.е. користат XAML, и одговараат на модуларниот концепт за градење на композитни апликации. Придобивката која е важна за градење на композитни апликации на Windows 8 платформа е тоа што освен што оперативниот систем врши активно прераспределување на користењето на ресурсите од апликациите за да ја намали потребата од енергија, дополнително со концептот на композитни апликации и самите тие активно ги користат само модулите кои реално се потребни во тој момент.

3.3. Библиотеки наменети за мултиоколинско таргетирање / Portable Class Libraries

Со користење на Portable Class Library во .NET Framework се овозможува да бидат развиени assembly-ја кои можат да бидат користени на повеќе од една платформа од .NET Framework-от. На овој начин класите можат да бидат пишувани така да ја опфаќаат бизнис логиката на проектите, и потоа овие класи да бидат референцирани и делени помеѓу разнородни типови на проекти.

Преносните библиотеки (асемблеја) кои се изградени како Portable Class Library проекти без дополнителна модификација работата на .NET Framework, Silverlight, Windows Phone 7/8 и Xbox 360 платформите [20]. Без постоењето на Portable Class Library типот на проекти кон секоја од платформите би била додавана истата класа рачно. На Слика 21 е прикажана дијалог форма на која во Visual Studio се означува за кои платформи библиотеката ќе може после компајлирање да се користи.



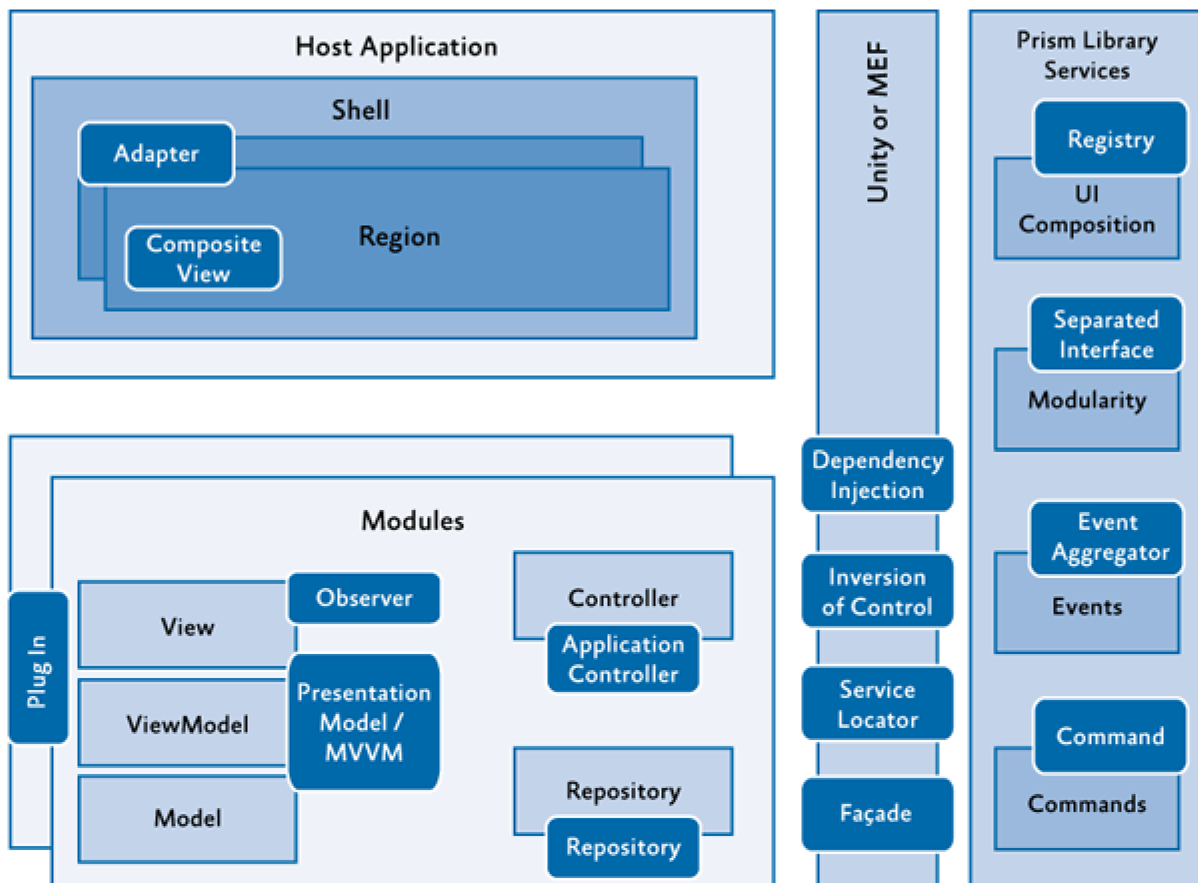
Слика 20. Portable Class Libraries за различни платформи

Figure 20. Portable Class Libraries, different platform targeting

4. Пример за примена на предложената модуларна архитектура (Show case Application)

4.1. Корелација на апликацијата со предлог-архитектурата

За да бидат претходно презентираните архитектури за развој на софтвер прифатени од заедницата и компаниите тие мора да нудат дополнително и начини за заштита на интелектуалната сопственост и можности за комерцијална употреба и пазарна оправданост. Решението го презентираме со цел исполнување на овие цели се заснова врз композитна структура на софтверска апликација изградена од модули, прикажан на слика 21, кој дополнително е надграден со механизам за авторизација и автентикација на модулите.



Слика 21. Композитна апликациска архитектура

Figure 21. Composite application architecture

4.2. Предлог-решение

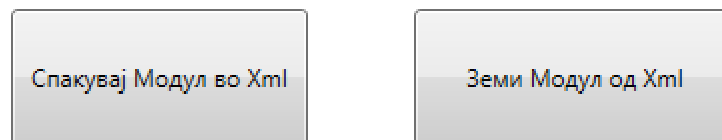
Решението се заснова на начелото за единствена одговорност и раслојување на одговорностите за одредени типови задачи. Се користи модуларно програмирање со тоа што за секој модул постои соодветен XML опис, додека за самите компоненти се имплементира верификација на модулите при поврзување со користење на XML потпишување на модулите односно механизам за авторизација и автентикација на модулите.

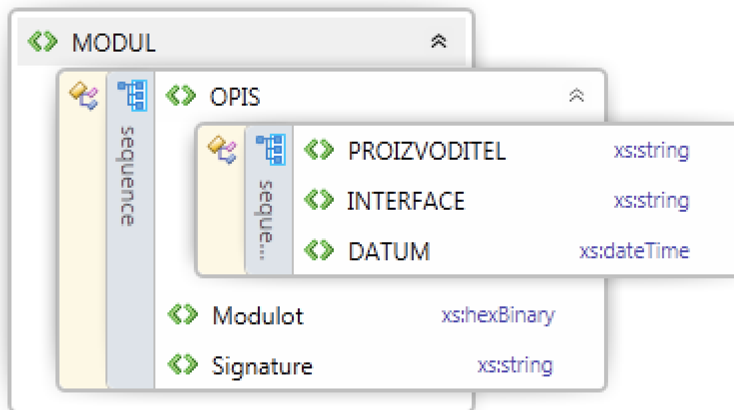
4.3. Опис на апликацијата

Апликацијата во рамки на Bootstrapper и обвивката (Shell) има дополнување со механизам за авторизација и автентикација на модулите, односно подесено е вчитување на сите модули од директориум со користење на MEF а самите модули се спакувани во xml датотеки и дигитално потпишани од страна на оној кој ги создал.

4.3.1. Размена на модули преку xml

Идејата е да секој модул после негово компајлирање биде спакуван во xml датотека заедно со описен таг за производителот на тој модул, интерфејсот на модулот, како и датумот и времето на пакувањето. Прикажано на слика 22.





Слика 22. Модул спакуван заедно со описот (интерфејсот) во xml датотека

Figure 22. Module packed in xml file with the corresponding interface

По пакувањето на модулот се врши и дигитално потпишување на модулот со приватен клуч и на крајот на xml датотеката се додава потпишаната hash вредност во Signature таг со помош на која потоа во рамки на композитната апликација се верифицира со користење на јавниот клуч веродостојноста на податоците содржани во xml датотеката која го содржи модулот.

Со користење на овој механизам се отвора моќноста да компаниите производители на модули ги дефинираат своите производи динамички, без притоа да стравуваат дека модулите кои ги изградиле ќе станат јавно достапни.

Дигитално потпишување и валидирање

Парот клучеви со кои се врши потпишувањето и валидацијата на xml датотеката која го содржи модулот е генериран со користење на RSA алгоритамот со користење на RSACryptoServiceProvider и големина од 1024 бита.

Xml приватен потпис


```
<RSAKeyValue>

<P>veUh0OiWAITvdFv0yJs8cA38IcaTobG8pn8p7f/0iXKjQ7OzR95PvcLHq+SnGLIwPJb7HOAR
25SBL77MNkiF2IFyBli5s9gUZA8KwLy/2xKWDIMDID7GPLGsWgVsBkQosQ13jv0C5kQJWrBzL
eSfajj/2mD0mx2UL8Hzj0086uc=</P>
  <Q>3jMIBdNcvP/tqHSzLZIvZMS2ApM=</Q>

<G>LI5QMSXh6hxUUfNHFy/Qx4tjn4H0xLNDeC0Mxn2hbuOHAWztnDihPXmVwet7yTkJy717Z
XPaZp+h5mJpuE9dlnFI+fiQ+w8ALVsKwsMRpZAdMbNDigpdGsIyBteRgu1212aIwb6LOn6vVInl
orec0oVRKexMJwA4NZZbXZDf+pg=</G>

<Y>FN0OQLNV1ArqcT6/kuIXWeYwjmER3WivKGSB7kQnoA0qpp7IZKcCPKJui0A/5pC2hLvTpr
g61lfq/wdcaxAhWH1H6NeehQDWXgJqssWjjo9KoU0psHmgi+fPRIFXIQFh2x6TKhqv9feKwOzF
urqfS7ERb62xPxjjF2ZKFGjhKIU=</Y>

<J>2sgYU/wGXIs9s9z0HNskVtxna7Fmb2OEIOgzYuB9EcI/8GFd+whwYhBi9EIV7c7kr4scyV94y
cTK/X+mbcGqdkHyxNEbnzQSabfGIqYStcjXyI2225oNaGAxqUr+4eg9L4B2e75spORQwltC</J>
  <Seed>GIAzCGgth3VaUHxTHi9pB687H+A=</Seed>
  <PgenCounter>3g==</PgenCounter>
  <X>qqO8URZo/nCp+Nd7wTJfjUWx488=</X>
</RSAKeyValue>
```


Пример код 38

Дополнително постои можност и за имплементација на решение во кое комплетно би се криптирале модулите т.е. нивната содржина и практично да бидат читливи само за одредена апликација која го поседува јавниот клуч пар на приватниот со кој се тие претходно криптирани.

5. Заклучок (Concluding remarks)

5.1. Заклучоци за архитектурните предности и недостатоци

Клучните разлики помеѓу Unity, MEF и PRISM се важни да бидат истакнати со главните предности на секоја од техниките/технологиите со цел да може да се изберат што по соодветни техники на работа во дадени сценарија.

5.1.1. Unity наспроти MEF

Unity се користи за статички зависности и во главно се користи како техника за интерна употреба, односно надворешно не може да се осознае дека Unity е користен во рамки на една апликација.

MEF се користи во сите случаи каде што се јавува потреба од употреба на динамички зависности, динамичко вчитување на компоненти. MEF користи атрибути за опишување на модулите и со тоа овозможува да апликациите лесно да можат да бидат надградени со таквите модули.

5.1.2. MEF наспроти Prism

Главната разлика помеѓу MEF и Prism е во тоа што Prism е предвиден за развој на композитни (модуларни) апликации вклучувајќи го и корисничкиот интерфејс, додека фокусот на MEF е кон развој на компоненти кои би биле лесно надградливи.

Prism користи управувач со региони (RegionManager), прибирач на настани (EventAggregator) и команди (Commands) за разрешување на прашањата поврзани со компонентите од корисничкиот интерфејс, додека MEF се фокусира само на динамичкото вчитување на компоненти.

5.1.3. Клучни точки во споредба на Unity, MEF и Prism

- MEF не претставува Inversion of Control (IoC) контејнер, иако ги користи концептите на IoC;
- Unity е IoC контејнер, и тоа е негова главна задача;
- MEF се фокусира само на надградувањето на апликациите со динамичко откривање и композиција на компоненти;
- Prism претставува целосно решение за развој на UI апликации;
- MEF не претставува целосно решение за развој на UI апликации и покрај тоа што се користи за развој на апликации кои се надградливи;
- MEF засега не може да се користи на Windows Phone платформата.

5.1.4. Оптичко процесирање и паралелно програмирање

Можноста за паралелно процесирање преку користење на различна фреквенција е една од двете главните придобивки на оптичкото процесирање т.е. оптичките кола. Практично со исто оптичко коло со само вклучување на нова фреквенција се добива нов виртуелен процесор со иста моќ како и претходниот, и тоа без дополнителни нус појави како загревање и слично. Втората придобивка е отпорноста на ваквите кола од електромагнетни појави.

Ова е во насока на создавање на степенско активирање на ресурсите т.е. процесорската моќ, тоа во моменти кога има потреба истиот уред може неговата моќ да се мултиплицира (преку активирање на различни фреквенции и виртуелни процесорски кола), а и обратно да се намали кога потребите се помали. Впрочем, може да се спомене иако можеби надвор од излагањето дека пристапот со користење на степенско активирање на ресурсите го однесе човештвото надвор од границите на планетата Земја. Таквите ресурси можат да бидат оптимално искористени доколку апликациите се градени на тој начин да користат паралелизам. Модуларниот пристап на градење на композитни апликации во кои модулите се лабаво врзани помеѓу себе е de facto пристап спремен за преку соодветен компајлер апликациите бидат компајлирани во форма која може да биде интерпретирана / извршувана на паралелни архитектури со нивно оптимално искористување.

5.2. Значење на тезата

Практиките кои се користат во развојот на софтверски апликации може да се каже дека се рефлексија на реалните животни навики на луѓето или рефлексија на постојни филозофии. Како добар пример може да се посочи заедницата која се занимава со развој на таканаречениот слободен софтвер, заедница која целокупното свое работење го темели на некоја филозофија на споделување со другите а за својот труд прифаќа само доброволни донации, филозофија на живеење која влече директни корени од прото-христијанската филозофија во која основни постулати се меѓусебното почитување, споделување и заедничко делување за доброто и благодатот на целото општество. Од друга страна практиката која ја сметаме за стандардна каде програмерот односно тимот е креатор на се и се грижи за сите идни измени, може да се каже дека е модел во согласност со филозофијата на Платон за постоењето на универзални идеи и следствено на тоа се што постои одговара на некоја од тие идеи. Врз основата на некоја идеја како некоја софтверска апликација треба да изгледа и што треба да задоволи како кориснички барања и бизнис логика и без разлика каков ќе биде крајниот резултат од процесот на развој на апликацијата пак таа ќе одговара на иницијалната идеја. Како што би рекол Платон ако имаме на ум одредена идеја за тоа како треба да изгледа едно животно коњ, без разлика на каков коњ ќе наидеме бел, црн, со шарки или без опаш пак ќе знаеме дека се работи за коњ односно дека одговара на нашето перципирање на идејата како еден коњ изгледа и кои карактеристики треба да ги исполнува. Ако можеме да изведеме паралела помеѓу идејата за софтверската апликација и коњот, движењето во насока односно развојот кон исполнување на идејата нема никогаш да не донесе до целосно исполнување на оригиналната идеја од причина што таа е променлива и најчесто не е дефинирана прецизно. Клучниот недостаток во овој начин на размислување и практиката која од него произлегува е во тоа што се секогаш исполнувањето на идеите е минимум периодично еден чекор назад од нивното дефинирање. Конкретно ако ја земеме идејата за развој на софтверска апликација за пишување и средување на документи и Microsoft Office Word и Open Office Writer апликациите одговараат на оригиналната идеја и базичните спецификации на таа идеја. Но идејата е жива и променлива, ако во 1995 година во идејата за апликација за пишување и средување на документи се подразбираше да со истата може да се замени функционалноста на машината за пишување денес во 2013 година под таа идеја се подразбира апликација која може да се нарече паметна, ќе ги предвидува потребите на корисникот и наредните негови чекори (за на најсоодветно место на екранот ќе ги прикаже менијата), ќе нуди поддршка за граматички правопис, автоматски превод на други јазици, гласовно пишување итн. Решението кое го моментално го применуваат софтверските компании е исполнувања на идејата од претходен период во идна верзија на софтверската апликација, пр. во Microsoft Office 2013 е реализирано она што произлегло како потреба во 2011 година. Затоа почитувањето на овој пристап по дефиниција заостанува зад потребите, во конкретниот случај со Microsoft Office со цели 3 години.

Алтернативниот пристап е практиката на постепено воведување на модуларниот пристап на градење на софтверските апликации. Паралелата во овој пристап со

секојдневните навики на живеење и филозофии присутни во општеството може да ја најдеме во културното движење на преродбата т.е. Ренесансата. Ќе го земеме примерот со Microsoft Office Word апликација која непретенциозно можеме да заклучиме дека го има приматот во рангот на апликации за пишување документи (и на Windows и на Macintosh платформи). Но ако некој некаде во светов добие некаква идеја за надополнување на функционалностите на конкретнава апликација која му произлегла од секојдневното работење и цени дека и на многу други како него би им го олеснила животот, да се запрашаме кој е патот за имплементација на една таква идеја. Едниот пристап е развој на апликација клон пристап кој од старт е осуден на неуспех, и вториот да се проба да се продаде идејата на Microsoft Office тимот а е релативно немерливо дали би успеал овој пристап. Можностите кои ги нуди модуларниот пристап на градење на апликациите имаат директно решение за конкретниот проблем. Кога Microsoft Office Word би бил развиен како модуларна композитна апликација нов функционален модул може да биде вклучен кон оригиналниот пакет од модули многу лесно и понуден независно на пазарот. На тој начин би биле мерливи и повратните информации во однос на таа нова функционалност, второ нема да постои периодично каснење кон прифаќање на новите идеи и функционалности на апликациите. Реалната мерливост на компонентите кои ќе се покажат како инвентивни и корисни и ќе го преживеат процесот на прифаќање од корисниците, многу е веројатно дека спрема ефектот што го постигнале ќе бидат вреднувани (откупени како авторски елементи) и вклучени кон оригиналните пакети од модули во нови верзии на софтверската апликација. Да се потсетиме и ренесансните филозофи, научници и уметници употребувале хуманистички метод во учењето и барале реалистичност и емотивност во науката и уметноста.

Овој принцип на работа се покажал како многу успешен во други индустрии пр. индустријата за производство на играчки, автомобилската индустрија. Конкретно во автомобилската индустрија јавна тајна дека неколку производители на автомобилски делови ги снабдуваат сите светски брендови на автомобили, но она што е различно и од што произлегува квалитетот на автомобилите е тоа што зад дизајнот на секој автомобил стои одреден бренд (дизајнерски тим) и секој производ (автомобил) е доверен на еден тим кој одговара за неговиот квалитет. Тој тим произведува еден автомобил од почеток до крај и става свој печат за неговиот квалитет и квалитетот на деловите од кои тој е составен.

Модуларниот пристап на градење на софтверските апликации нуди континуиран развој затоа што во општеството постои потреба за постигнување на совршенство и секогаш ќе постојат проби (мутации) за негово постигнување. Би го цитирале основачот на Lego, Ole Kirk Christiansen најуспешната приказна за успех со примена на модуларниот пристап на работење, „det bedste er ikke for godt“– најдоброто никогаш не е доволно добро. Секогаш има можност за подобрување на производите, ќе се нудат мутации на функционалностите а преку процесот на природна селекција секој ќе ги користи оние софтверски компоненти кои за неговите процеси ќе даваат најдобри резултати (брзина на извршување, кориснички интерфејс, адекватност на кодот кон соодветна софтверска или хардверска архитектура и сл.) и ќе бидат издвоени најсоодветните компоненти во нови изданија на апликациите составени од модули. Ова е принцип е дефиниран од Charles

Darwin во неговото дело „За ПОТЕКЛОТО НА ВИДОВИТЕ, со помош на природна селекција или со преживување на најприспособливите раси во борбата за живот“ издадено во 1859 год. [21]

На крајот да веруваме дека тоа што за општеството од 17 век беше воведот во мануфактурното производство, во 21 век ќе воведот во воведувањето на модуларниот пристап на градење на композитни софтверски апликации, и во иднина наместо да произведуваме готови играчки, ќе произведуваме софтверски „лего коцки“.

5.3. Предности и недостатоци

Модулирањето при дизајн е обид да се комбинираат предностите на стандардизација (зголемен обем на производство значи намалени трошоци за производство по единица производ) со барањата за прилагодување на производот спрема одредени барања на потрошувачот. Главен недостаток на модулирањето е тоа што модулираните системи не се оптимизирани во перформанси. Тоа е најчесто поради употребата на интерфејси за комуникација помеѓу модулите.

Многу е поедноставна организацијата на развојот и доделувањето на задачи на програмерите кои работат на композитни модуларни апликации. Ако апликацијата е добро дизајнирана по начелото на единствена одговорност на одреден модул, практично може да се дефинира оптимално време за развој на индивидуална компонента / модул кое во некоја проценка е од 5 до 30 минути. На проект лидерот значајно му е олеснета работата поради лесната пресметливост на ангажманите а и на самите програмери им се дава дополнителен мотив, затоа што од секоја задача (изработен модул) ја добиваат персоналната сатисфакција за добро завршена работа, мала но завршена, и со тоа се добива на психолошкиот момент кај секоја индивидуа за самомотивација. Од таа гледна точка а секако во насока на објектно ориентиран начин на размислување многу е полесно и побрзо развивањето на пр. 100 мали модули секој во просек по половина час наспроти развивањето на еден проект кој би ги опфаќал функционалностите на тие модули а преставува врзана целина. Искуството покажува дека развојот на проектот како хомогена целина нема да е математички сто пати по половина час односно седум работни денови туку најверојатно десет и повеќе.

Придобивката која е важна за градење на композитни апликации на Windows 8 платформа е тоа што освен што оперативниот систем врши активно прераспределување на користењето на ресурсите од апликациите за да ја намали потребата од енергија, дополнително со концептот на композитни апликации и самите тие активно ги користат само модулите кои реално се потребни во тој момент. Предност ќе се добие со тоа што во рамки на апликацијата активни ќе се само потребни функционалности но ќе се изгубат перформанси заради дополнителните процеси и енкапсулации кои се воведуваат за да се овозможи лабава врска помеѓу компонентите. Како компензација на тој недостаток се отвора можноста секој модул да биде изграден на најниско можно ниво, компајлиран најсоодветно, прилагоден на соодветна архитектура или систем и од тоа ќе се добијат подобри перформанси.

На кратко да споменеме што се овозможува примената на развојот на композитните модуларни апликации:

- можност за избор на кориснички интерфејс;
- можност за слободно и неограничено делење на апликациите, нивните модули и податоците.

- можност за филтрирање на приказот на информациите притоа водејќи грижа за ресурсите на едноставен начин (за кој развивачите на апликациите не мора дополнителен труд да вложуваат);
- практично одржување на лабавата врска (loosely coupled) помеѓу елементарните функционални компоненти - модули;
- можност за користење известувања (events) во лабава врска;
- овозможување на појава на процес на мутација и природна селекција на компонентите кои се користат во рамки на композитните апликации што ќе доведе до природен процес на еволуција на апликациите спрема потребите на корисниците.

5.4. Придобивки од развивањето на модуларни апликации

Генерално земено повеќето програмери создаваат добро структурирани апликации со користење на асембleja, интерфејси, класи, како и проверени објектно ориентирани принципи на дизајн. И покрај тоа, освен ако не се води голема грижа, апликациите сепак можат да станат „монолитни“ т.е. во себе да содржат функционалности кои се имплементирани со меѓусебна цврста врска, што ги прави апликациите да се тешки за развивање, тестирање, надградување и одржување.

Пристапот за развивање на модуларни апликации од друга страна помага да се идентификуваат функционалните делови на апликацијата кои можат да бидат одвоени и како такви засебно развивани и тестирани. Овој пристап го прави развојот и тестирањето да бидат многу полесни а од друга страна самата апликација по флексибилна и полесна за надградување во иднина. Придобивката од модуларниот пристап е тоа што ја прави целата апликациска архитектура по флексибилна и полесно одржлива затоа што дозволува апликацијата да биде разделена на мноштво лесно управливи делови. Секој таков дел опфаќа точно одредена и специфична функционалност, и секој дел е интегриран во апликацијата преку јасни но лабаво врзани комуникациски канали.

Предноста при имплементација е тоа што секој модул има една и единствена намена, може да биде многу едноставен и со тоа лесен и поисплатлив за развој и имплементација. Системите кои се дизајнирани со користење на функционални делови се полесни за модифицирање, затоа што секој дел го прави само тоа за што е наменет.

Поаѓајќи од фактот дека просечно повеќе од три четвртини од периодот на постоење на еден систем опфаќа одржување, ова е клучна предност. Дополнително ги прави системите да бидат полесни за разбирање, документирање и ја упростува обуката за нивно користење.

Во согласност со новиот концепт за развој на оперативни системи каде поентата е да апликациите функциите со кои во даден момент да не се оперира и на некој начин се стопираат или трајно или привремено замрзнат концептот на модулација доаѓа дополнително до израз, поради тоа што и апликациите ќе бидат функционално намалени (не сите модули од една апликација се користат во ист момент).
= референца кон WIN 8 – Минимизирање на потрошувачката на енергија на уредите а при тоа максимизирање на перформансите, правејќи ги системите “fast and fluid”.

5.5. Иден развој

Развојот на модерните објектно-ориентирани софтверски архитектури е во насока на постепено воведување на модуларниот пристап на градење на софтверските апликации и зголемување на степенот на реискористувањето на модули развиени од различни компании. Од причини што постојните технологии како Silverlight уште во својот основен дизајн се ограничени да оневозможуваат динамичко вчитување на компоненти треба да се направат напори да се променат за целиот концепт за развој на модуларни апликации биде што по применлив. Евентуалната замена на Silverlight со HTML 5 може да доведе до обиколно решавање на овој проблем, затоа што HTML 5 во основа е дизајниран да динамички вклучува компоненти во апликацијата која се извршува.

Дополнително дистрибуцијата на модули меѓу разнородни физички уреди / оперативни околини е предмет на размислување. Дали со тек на времето неминовно ќе преовлада еден оперативен систем како доминантен или развојот ќе продолжи во насока на неколку доминантни оперативни системи е од клучно значење, но без разлика пристапот на развој на модуларни апликации може да биде прифатен и надвор од светот на Microsoft.

Во наредната верзија на Prism наменета за Windows Runtime (Windows 8) проект насловен како Kona не се очекува ништо позначајно во однос на дополнителни функционалности кои би ги овозможила Prism архитектурата и всушност таа ќе претставува адаптација за оперативни системи базирани на Windows Runtime. Главниот проблем кој што може да се воочи од сега е големиот степен на виртуелизација кој ќе се создаде и кој ако не е пропратен паралелно со имплементација на начелото за единствена одговорност на компонентите ќе биде причина истиот да стане непопуларен и никогаш да не заживее, сценариото е веќе видено преку обидот за имплементација на композитни решенија на Java архитектурата.

6. Користена литература (Bibliography)

[1] "www.idc.com," International Data Corporation (IDC) , 01 02 2012. [Мрежен]. Available: http://www.idc.com/getdoc.jsp?containerId=prUS23297412. [Пристапено на 28 04 2012].

[2] Charles Arthur, "Guardian," 01 02 2013. [Мрежен]. Available: http://www.guardian.co.uk/technology/2013/feb/01/tablets-crossing-point-pcs. [Пристапено на 28 04 2013].

[3] "Smart phones overtake client PCs in 2011," Canalys , 03 02 2012. [Мрежен]. Available: http://www.canalys.com/newsroom/smart-phones-overtake-client-pcs-2011. [Пристапено на 22 04 2013].

[4] "U.S. DoD Electro-Optical/Infrared Systems Market," Finannce Yahoo, 27 03 2013. [Мрежен]. Available: http://finance.yahoo.com/news/u-dod-electro-optical-infrared-191100346.html. [Пристапено на 23 04 2013].

[5] U. DoD, "U.S. DoD Electro-Optical/Infrared Systems Market," 10 2012. [Мрежен]. Available: http://www.reportlinker.com/p01020582/U-S-DoD-Electro-Optical-Infrared-Systems-Market.html. [Пристапено на 04 2013].

[6] D. Brion, "The MVVM Pattern Is Highly Overrated," 21 07 2010. [Мрежен]. Available: http://davybrion.com/blog/2010/07/the - mvvm - pattern - is - highly - overrated/. [Пристапено на 10 05 2011].

[7] R. C. Martin, „Агилен развој на софтвер, принципи, модели и практики" Agile Software Development, Principles, Patterns, and Practices., Prentice Hall. ISBN 0135974445., 2002.

[8] T. DeMarco, „Структурни анализи и системски спецификации" Structured Analysis and System Specification., Prentice Hall. ISBN 0138543801., 1979.

[9] E. W. Dijkstra, "On the role of scientific thought," во%1 *Selected Writings on Computing: A Personal Perspective*, том ISBN 0–387–90652–5, New York, NY, USA, Springer-Verlag, ISBN 0–387–90652–5, 1982, p. 60–66.

[10] C. Reade, Elements Of Functional Programming, Boston, MA, USA: Addison - Wesley Longman Publishing Co. ISBN 0201129159, 1989.

[11] H. H. P. Johansson, On the Modularity of a System., MALMO HOGSKOLA, Sweden: Centre for Technology Studies, August 19, 2010.

[12] J. Haas., “"Modular programming",” [Мрежен]. Available: http://linux.about.com/cs/linux101/g/modularprogramm.htm. [Пристапено на 25 09 2011].

[13] S. Haridi и N. Franzén., “Mozart Programming System Documentation,” [Мрежен]. Available: http://www.mozart-oz.org/home/doc/tutorial/node7.html. [Пристапено на 15 03 2011].

[14] G. M. L. C. W. Stevens, “"Structured Design",” *IBM Systems Journal,* том 13 (2), pp. 115-139, 1974.

[15] S. D. F. Beck, “On the Congruence of Modularity and Code Coupling.,” во%1 *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (SIGSOFT/FSE '11)*, Szeged, Hungary, September 2011..

[16] R. S. P. Pressman, Software Engineering - A Practitioner's Approach - Fourth Edition., ISBN 0-07-052182-4, 1982.

[17] G. C. ,. D. H. ,. B. N. ,. M. P. ,. K. S. Bob Brumfield, Developer's Guide to Microsoft® Prism 4: Building Modular MVVM Applications with Windows® Presentation Foundation and Microsoft Silverlight®, Redmond, Washington: MSDN, ISBN 13: 9780735656109, 29.03.2011.

[18] “Composing the User Interface,” MSDN Microsoft, [Мрежен]. Available: http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40).aspx. [Пристапено на 16 02 2012].

[19] “Developing an Advanced Windows Phone 7.5 App that Connects to the Cloud,” Microsoft, 25 5 2012. [Мрежен]. Available: http://go.microsoft.com/fwlink/?LinkId=205602. [Пристапено на 2 6 2012].

[20] “Portable Class Libraries,” MSDN - Microsoft, 15 3 2012. [Мрежен]. Available: http://msdn.microsoft.com/en-us/library/gg597391(v=vs.100).aspx. [Пристапено на 08 11 2012].

[21] C. Darwin, On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life, London: John Murray, 24 November 1859.

6.1. Прилози

Слика 1. Приказ на продажба на Smartphone-и, песонални компјутери и таблети по квартали од 4-ти квартал 2009-та. Извор IDC [2] 10
Слика 2. Статистички податоци за продажба на Smart phone-и и персонални компјутери, 2010 година во однос на 2011 [3] 11
Слика 3. Samsung Brix Mobile 12
Слика 4. Опис на MVVM....................15
Слика 5. Типови на спарување....................27
Слика 6. Композитна апликација, која користи повеќе системи во позадина 41
Слика 7. Типична архитектура на композитна апликација имплементирана со Prism библиотеката....................49
Слика 8. Постапки за создавање на композитна апликација....................51
Слика 9. Начин на повикување на обвивката, view и регионите 52
Слика 10. Дијаграм на поврзување со Prism библиотеката....................53
Слика 11. Основни чекори на Bootstrapper процесот....................55
Слика 12. Имплементации на Common Service Locator (заеднички локатор на сервиси) во Prism 66
Слика 13. Користење на RegionContext 71
Слика 14. Собирач на настани (Event aggregator)....................72
Слика 15. Дизајн на модуларна апликација составена од неколку модули 75
Слика 16. Процесот на вчитување на модули 78
Слика 17. Апликација со модули кои се делови од вертикална поделба 84
Слика 18. Апликација со модули организирани во хоризонтални слоеви....................85
Слика 19. Пример за вметнување на зависности (dependency injection) 89
Слика 20. Portable Class Libraries за различни платформи 110
Слика 21. Композитна апликациска архитектура 111
Слика 22. Модул спакуван заедно со описот (интерфејсот) во xml датотека 113

Figure 1. Smartphone, PC and tablet shipments by quarter from 4Q 2009. Source: IDC [2] 10
Figure 2. Smart phone and client PC shipments, year 2010 in comparison with 2011 [3]..........11
Figure 3. Samsung Brix Mobile 12
Figure 4. Description of MVVM 15
Figure 5. Types of coupling..27
Figure 6. Composite application with multiple backend systems 41
Figure 7. Typical composite application architecture with the Prism Library 49
Figure 8. Activities for creating a composite application 51
Figure 9. Shells, views, and regions...52
Figure 10. Diagram demonstrating connecting to the Prism Library 53
Figure 11. Basic stages of the bootstrapping process 55
Figure 12. The Common Service Locator implementations in Prism 66
Figure 13. Using RegionContext 71
Figure 14. Event aggregator...72
Figure 15. Design of Modular application composed of modules..75
Figure 16. Module loading process 78
Figure 17. An application with modules organized around vertical slices...................................84
Figure 18. An application with modules organized around horizontal layers. 85
Figure 19. Example of dependency injection..89
Figure 20. Portable Class Libraries, different platform targeting 110
Figure 21. Composite application architecture 111
Figure 22. Module packed in xml file with the corresponding interface 113

Равенка 1. Одредување на ниво на спарување 32

Equation 1. Determination of the level of coupling...32

Табела 1. Типови на поделба во однос на архитектурата 21

Table 1. Types of separation in terms of architecture 21