Софийски Университет „Св. Климент Охридски”
Факултет по математика и информатика
Катедра Информационни Технологии

Aspect .NET - Аспектно-ориентирано разширение на Microsoft Visual Studio 2003

дипломант: **Александър Бойчев Колев**
специалност: **Софтуерни Технологии**
фак.номер: **М-21554**

научен ръководител: **доц. д-р Силвия Илиева**

София ,
14.02.2007

Съдържание

Таблица на използваните фигури

1 Въведение

Ето как започва една публикация на изследователска група от Xerox Palo Alto Research Center ***[11]***:

„Известни са множество проблеми в областта на програмирането, за които не са достатъчни нито процедурни, нито обектно-ориентирани решения, за да обхванат прецизно важните дизайн решения, реализирани от дадена програма. Това принуждава реализацията на тези дизайн решения да бъде разпръсната из целия код на програмата, което води до така наречения 'заплетен' код, който е изключително труден за писане и поддръжка. Ние представяме анализ защо е толкова трудно да пресъздадем дизайн решенията в програмен код. Характеристиките, които се адресират от дизайн решенията, ще наричаме аспекти и ще покажем, че причината, поради която е трудно да се обхванат в кода е, че те пресичат вече съществуващата функционалност на системата в множество точки. Ще представим основата на нова техника на програмиране, наречена аспектно-ориентирано програмиране, която прави възможно по-улеснено писането на програми, използващи гореспоменатите аспект; включваща подходяща изолация, композиция и преизползване на програмния код."

Още от самото създаване на компютрите, езиците за програмиране не престават да се развиват: от машинен код и асемблерни езици до процедурни и обектно-ориентирани. Всеки прогрес в технологиите подобрява средствата, с които можем да постигнем по-добро разделяне на отношенията в кода на една програма, което естествено води до улеснена реализация и последваща поддръжка. През последните 20 години доминиращата парадигма беше тази на Обектно-ориентираното Програмиране (ООП) поради способността му да моделира и декомпозира ефективно предметни области от реалния живот, посредством обектния си модел.

Разделение на отношенията играе основна роля в разработката на софтуер. Ossher ***[4]*** описва разделението на отношенията като: „Най-общо казано става въпрос за възможността да се идентифицират, капсулират и обработват тези части от софтуера, които имат отношение към дадена концепция, цел или нужда". Поради тази причина под 'отношение' (concern) трябва да се разбира модул, който обхваща (encapsulates) определена област,

имаща значение за нас. Отношенията са главната причина за декомпозиция на софтуера. Съществуват много и различни видове отношения, за които програмистът трябва се погрижи по време на различните фази на един проект. Конкретно в ООП, единицата, енкапсулираща отношение се нарича клас.

В рамките на ООП всичко бива моделирано в класова структура. Тази техника е наистина ефективна в голяма част от случаите, тъй като с обектния модел лесно се осъществява описание върху съответните предметни области. За съжаление освен множеството преимущества при използването на ООП, то води и до проблеми. Някои от тях стават явни, когато част от отношенията започнат да имат допирни/пресечни точки. Казвайки, че отношенията се пресичат, имаме предвид, че са налични повече от едно отношения в един елемент от системата (в зависимост от парадигмата за декомпозиция варират, но при ООП – клас, метод и т.н.), а други са разпростряни върху множество от елементи на системата. Това от своя страна води до следните проблеми ***[3]*** :

1. Оплетен код – получава се, когато няколко отношения са реализирани в един елемент на системата. Ефектът тук е, че кодът става по-сложен и съответно по-труден за разбиране и поддръжка
2. Разхвърлян код – получава се, когато реализацията на дадено отношение се постига чрез дублиране на кода или писането на различен код, но разпръснат в множество различни елементи на системата. Ефектът е код, който е ‘разхвърлян’ из цялата система, което автоматично означава усложнена поддръжка, тъй като програмистът трябва да има знание за цялата система при провеждането на корекции или друг вид ‘ремонтна’ дейност.

Повечето хора, които някога са работили с код, в който се срещат crosscutting функционалности, най-вероятно вече са се сблъскали с проблемите, които са породени от липсата на подходяща гранулярност на програмния модел. Поради разпръснатостта на кода, реализиращ crosscutting функционалностите, програмистите често срещат трудности с обосновката, реализацията и поддръжката на този код. Като пример пак можем да посочим кода, реализиращ трасиране – той обикновено е преплетен с код, чиято цел е съвсем друга, често реализираща някаква бизнес логика. В зависимост от сложността и обхвата на функционалността, това сплитане варира от съвсем малко до огромно по размер. Съответно една промяна в начина на трасиране би довела до огромно

усилие от страна на програмиста да обиколи и редактира множеството места, на които се извършва трасирането.

АОП допълва ООП като улеснява програмистите, предоставяйки им нов тип модулярност, която събира кода, реализиращ даден crosscutting функционалност в една програмна единица ***[15]***. Тази единица се нарича *аспект*, откъдето е името и на самата техника Аспектно-Ориентирано Програмиране. Използвайки аспекти, програмирането на crosscutting функционалности става лесно и удобно, предоставящо възможност за многократно използване на вече написаните веднъж аспекти. Самите аспекти могат да бъдат добавяни, премахвани или дори променяни по време на компилация, което прави крайния резултат достатъчно ефективен за всякакви цели.

1.1 Мотивация

За пръв път терминът ‘аспект’ се среща в публикация на Kiczales ***[3]***. Аспект е единица, капсулираща дадено отношение, което пресича системните елементи в множество точки. Техниката на програмиране, която се занимава с тези дизайн проблеми се нарича Аспектно-ориентирано програмиране (АОП).

Аспектно-ориентираното разработване на софтуер (АОРС) е нова технология, която се опитва да наложи АОП в масовото производство на софтуер с цел по-добро разграничение и моделиране на отношенията. Аспектите се ползват във всяка една стъпка от цикъла на производствения процес. Типични примери за аспекти са: сигурност, качество на услугата (QoS), поддръжка на транзакции, синхронизация, трасиране и т.н.

Към момента има налични голям брой реализации на инструменти и среди, поддържащи АОП и АОРС ***[6]***. За съжаление обаче, по-голямата част от тях са съвместими (и написани специално) само с Java. По отношение на .Net технологията на Майкрософт, която в последните години набира все по-голяма популярност, броят на АОП и АОРС разработките е чувствително по-малък, което влияе и върху популярността на тези концепции измежду програмистите на .Net езиците – C#, VB.Net, Cobol.Net и т.н.

Именно гореспоменатият факт е и основната ми мотивация за реализирането на инструмент, който ще улесни .Net обществото, предоставяйки средство, с което лесно и удобно ще могат да се експериментират и навлязат в

концепциите на АОП, без значение от конкретния .Net език, на който се програмира.

1.2 Цел и задачи на дипломната работа

Целта на дипломната работа е да се проектира и разработи инструмент, наречен Aspect.Net, който да позволява използването на АОП в рамките на MS .Net Framework 1.1 и по-конкретно чрез интеграция със средата на разработка Microsoft Visual Studio .Net 2003. Задачата включва и разглеждане и оценка на Microsoft .Net като платформа за АОРС.

Конкретни задачи на дипломната работа:

1. Дизайн и реализация на АОП инструмент за Microsoft .Net Framework, позволяващ използването на АОП механизми в .Net
2. Дизайн и реализация на разширение за Microsoft Visual Studio 2003, което улеснява и облекчава работата на разработчиците по отношение на АОП в .Net платформата
3. Оценка на разработката по следните показатели – начин на реализиране на аспекти, методи за композиция, възможности за модуляризация, интегриране в софтуерния процес, използваемост.
4. Изследване на MS .Net Framework с цел установяване на възможностите за успешна реализация на АОСР в рамките на фреймуърка

По отношение на самата реализация на разширението (Aspect .Net), специфичните цели са:

- Да позволим на разработчиците да използват АОП в ежедневната си работа с MS .Net фреймуърка;
- Да позволим на разработването на бизнес логиката и всеки един от аспектите на различен език, част от .Net платформата;
- Да улесним разработчика, чрез предоставяне на разширение за средата на разработка MS Visual Studio 2003, с което той ще може лесно да създава, конфигурира и компилира проект, изграден както със стандартни ООП, така и с АОП техники;

- Разработката на тези инструменти да не включва разширение към никой от .Net езиците, като по този начин ще избегнем нуждата от създаване на компилатор, специфичен за разширението.

1.3 Структура на дипломната работа

Глава 1 **Въведение** – въвежда читателя в областта на езиците за програмиране, мотивацията за този проект, целите и основните идеи на Aspect.Net.

Глава 2 **Основни концепции, проблеми и перспективи в АОП** – запознава читателя с базовите концепции в областта на аспектно-ориентираното програмиране и аспектно-ориентираната разработка на софтуер като цяло. Идентифицира основните проблеми и перспективи пред АОП и АОСР.

Глава 3 **Microsoft .Net като платформа за АОСР –** разглежда в детайли архитектурата на .Net като платформа за АОСР. Запознава читателя с текущите АОП разработки по отношение на .Net.

Глава 4 **Изисквания и дизайн –** идентифицира функционалните и нефункционалните изисквания. Описва общия дизайн и взаимоотношения между основните компоненти на Aspect.Net.

Глава 5 **Реализация –** описва реализацията на проекта, както и реализацията на някои от основните му характеристики. Включва примерен код, потвърждаващ някои от основните концепции.

Глава 6 **Оценка на решението –** дефинира критериите, по които ще бъде направен сравнителен анализ на вече съществуващи инструменти в областта на АОП и Aspect.Net. Запознава читателя с резултатите от оценката.

Глава 7 **Пример на употреба на Aspect.Net –** описва реален проблем в реален проект, който трябва да бъде решен. Разглежда два подхода – стандартен (ООП) и с използване на Aspect.Net, като в края на главата предоставя на читателя няколко графики, резултат от сравнението на двата подхода.

Глава 8 **Изводи и заключение –** преглежда целите на дипломната работа и как последните са изпълнени. Описва предложения за бъдещи разработки и подобрения на Aspect.Net.

1.4 Обобщение

В тази глава представихме основните причини за поява и концепции на AOCP; защо обектният модел не успява да разреши всички проблеми на фазата на дизайна и като следствие от това получаването на ‘разхвърлян’ и ‘оплетен’ програмен код. AOCP е още в началото на развитието си, но все пак вече има десетки групи, занимаващи се с разработки в тази област и определящи бъдещото й развитие.

2 Основни концепции, проблеми и перспективи в АОП

Аспектно-ориентираната Софтуерна Разработка е бурно развиваща се парадигма, предоставяща специални абстракции за отношения, които пресичат множество компоненти на системите, като по този начин се сплитат със самите компоненти. Представяйки пресичащите се отношения, наричани още аспекти, като самостоятелни абстрактни единици и чрез предоставяне на средства за комбиниране на аспектите и компонентите, може да бъде подобрена модулярността на системата, като по този начин се намалява сложността и се улеснява поддръжката.

Основна задача на тази глава е да даде една по-широка концептуална представа за целта, средствата и проблемите пред АОСР.

2.1 Принцип на обособяване на отношенията

За да разберем идеите на АОСР първо трябва да се запознаем с принципа за обособяване на отношенията, който може да бъде считан за един от основните принципи в разработването на софтуер. Този принцип гласи, че даден проблем въвлича различни видове отношения, които трябва да бъдат идентифицирани и обособени, за да се справим със сложността и да постигнем желаните фактори по отношение на качеството – адаптивност, лесна/улеснена поддръжка, многократна употреба. Принципът може да бъде прилаган по много начини и не е преувеличение ако кажем, че той е един от основните при разработването на софтуер.

Въпреки общоприетото одобрение на нуждата да се прилага обособяване на отношенията, все още няма добре установено разбиране на идеята. Например в ООП разделените отношения се моделират посредством употребата на обекти и класове, които обикновено са наследени от съответните функционални изисквания и случаи на употреба (use cases). В структурния подход, отношенията са представени като процедури. В АОП терминът *отношение* е разширен с така наречените *пресечни характеристики* като синхронизация, управление на паметта и съхранение на данните. Последното може да бъде считано като генерализация на дефиницията на *отношение* в контекста на езиците за програмиране. Въпреки, че считаме това за естествена разработка на

софтуер, то увеличава нуждата за опресняване на знанието ни за това какво представлява *отношението,* тъй като отношенията вече не са ограничени до обекти и функции. Задачата за обособяване на правилните отношения се усложнява още повече, тъй като вече трябва да се справяме с по-голям набор и разнообразие от отношения.

За да предоставим подходяща дефиниция на понятието *отношение,* трябва да гледаме на разработката на софтуер, като на процес на решаване на задача, в която се търси софтуерно решение, отговарящо на дадени изисквания. Това можем да изразим така:

$$R \rightarrow S$$

където R представлява изискванията, S решението, а стрелката процеса на трансформация. Например за дизайн на разпределена система, изискванията R може да бъдат поддържането на стабилност под влияние на постоянни технически повреди. Съответно решението S може да бъде като протокол за възстановяване от състояние на повреда.

Като цяло се смята, че изискванията R включват специфицирането на конкретните проблеми. В практиката, обаче, търсенето на правилните проблеми и изразяването им не е тривиална задача. За да опростим това би било добре да въведем междинна фаза, в която съществените проблеми се отделят, използвайки първоначалната спецификация на изискванията:

$$R \rightarrow P \rightarrow S$$

Тук P представлява реален проблем, за който се търси съответно софтуерно решение. В примера за възстановяване от техническа повреда, P може да представлява проблем с дизайна и проектирането на алгоритъм за автоматично и прозрачно възстановяване от състояние на грешка.

Нека сега разгледаме модела на решението S в повече подробности. Можем да дефинираме S като множество от абстракции и множество от релации, изпълняващи определена роля.

$$S = (SA, SR)$$

Тук SA представлява множеството абстракции (sa_1, sa_2, sa_3,…, sa_n), а SR множеството релации между тези абстракции (sr_1, sr_2, sr_3,…, sr_n). Според теорията, за даден проблем P може да съществуват повече от едно решение. Качеството на различните решения е пропорционално на степента, с която те покриват следните характеристики ***[13]***:

1. *Уместност на решението*

Решението S трябва да задоволява целите и ограниченията, поставени от проблема P. С други думи S трябва да бъде *уместно* и валидно за контекста, дефиниран от P.

2. *Каноничен вид*

Решението S трябва да включва нужни и достатъчни абстракции, за да предостави приложимо решение. Неуместни или излишни абстракции не трябва да бъдат включени в решението. С други думи решението S трябва да бъде с широко приложение, но въпреки това минимално по размер – т.е. в каноничен вид.

Уместността гарантира, че се търси правилното решение за дадения проблем. Каноничността на решението води до свеждане на ненужната сложност до минимум. Очевидно е, че за всеки архитектурен дизайн горните характеристики играят ключова роля. Компонентите на една архитектура трябва да бъдат уместни за решаването на дадения проблем, както и в каноничен вид, за да сме сигурни, че даденият проблем е решен по правилния начин.

Имайки предвид горният модел за разработка на софтуер, можем да дефинираме *отношение* ***[13]***:

Отношение: *Абстракция на решение в каноничен вид, уместна за дадения проблем.*

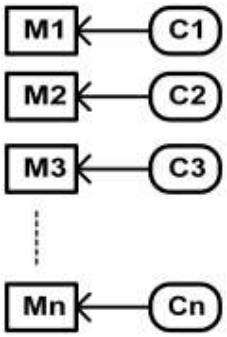
Трябва да отбележим, че от тази дефиниция следва, че отношенията не са абсолютни, а относителни по отношение на разглеждания проблем. Това, което може да бъде отношение за даден проблем може да не бъде за друг.

Генерализирането на отношението се нарича *концепция*. Концепция обикновено се дефинира като (умствено) представяне на категория от конкретни инстанции на отношения и се формира чрез обособяване на знанието за самите конкретни инстанции. Концепциите не са произволни абстракции или групи конкретни инстанции на отношения, а са добре дефинирани с консенсус от експерти в дадената област. Като такива концепциите са стабилни и добре дефинирани абстракции с богата семантика. В този контекст, считаме че отношенията са концепции, предоставящи канонични абстракции за решаването на даден проблем.

2.2 Основни концепции

Модулна декомпозиция

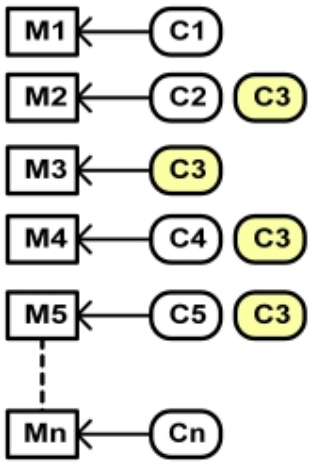
Принципът на разделяне на отношенията гласи, че за предпочитане е всяко отношение на даден дизайн-проблем да отговаря на един модул от системата. С други думи – проблемът трябва да бъде декомпозиран на модули, всеки един от които капсулира по едно отношение на проблема. Предимството в случая е, че отношенията са локализирани и съответно по-лесно разбираеми, разширяеми, многократно използвани, адаптирани и т.н. Този процес на декомпозиция е илюстриран на фигура 1. Дизайн задачата е декомпозирана до отношенията от C1 до Cn и всяко от тези отношения се съпоставя на отделен модул M. Модул е обособяването на модулна единица в даден език за програмиране (клас, функция, процедура и т.н.)



Фигура 1 – Съпоставяне на отношенията и модулите

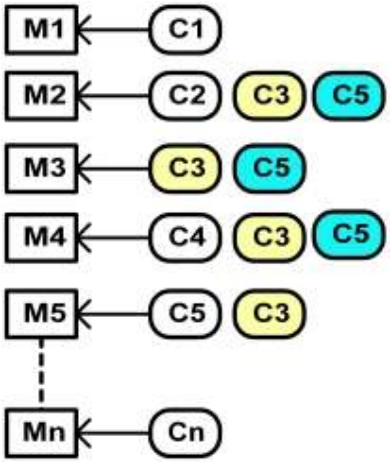
Пресичащи се отношения

Голяма част от отношенията могат да бъдат обособени в отделни модули. За съжаление, съществуват такива, които не могат лесно да бъдат отделени и използвайки езика, на който програмираме, съпоставяме всяко едно от тези отношения на множество модули. Такива отношения се наричат **пресичащи се отношения**. На фигура 2 може да се види как отношение C3 е съпоставено на модулите M2, M3, M4 и M5. Наричаме C3 пресичащо се отношение или аспект. Примери на аспекти са: следене, трасиране, синхронизация, балансиране на натоварването и т.н. Аспектите не са резултат на лош дизайн, а на далеч по-основни причини. Лош дизайн, включващ разпръснати отношения по различните модули може да бъде преобразуван до дизайн, в който всеки модул адресира дадено отношение. Ако обаче имаме работа с пресичащи се отношения това не е възможно – т.е. всеки опит за преобразуване на кода ще бъде неуспешен, а пресичането на отношенията запазено. Пресичащите се отношения са сериозен проблем, тъй като са по-трудни за разбиране, разширение, многократна употреба, поддръжка и т.н., защото са разпръснати на много места. Идентифицирането на местата, където пресичането е налично е един проблем, адаптирането на отношението без странични и непредвидени ефекти, е друг.



Фигура 2 – Отношението C3 пресича модулите M2, M3, M4 и M5

Проблемът се задълбочава ако имаме работа с множество пресичащи се отношения. Фигура 3 илюстрира 2 взаимопресичащи се отношения – C3 и C5.



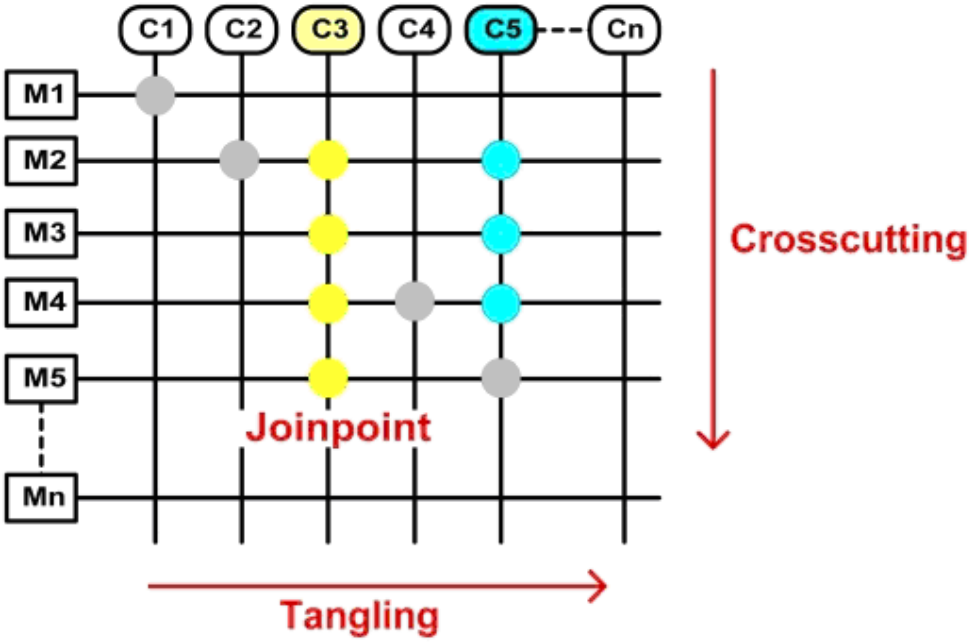
Фигура 3 – Пресичане на отношенията C3 и C5

Сплетени отношения

Тъй като не можем лесно да локализираме и разделим пресичащите се отношения, повечето модули ще обхващат по повече от едно отношение. В този случай казваме, че отношенията са сплетени в съответния модул. Например – на фигура 3 отношенията C2, C3 и C5 са сплетени в модул M2. Важно е да отбележим, че отношението C2 не е пресичащо. ***[21]***

Точки на свързване

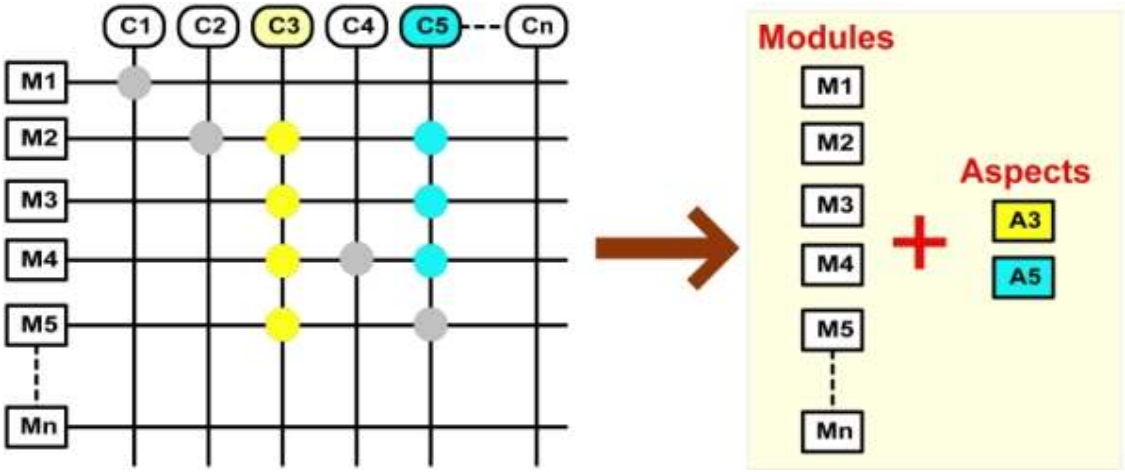
Фигура 4 илюстрира същата информация, като показаната във фигура 3. Тук модулите са подредени по вертикалната ос, а отношенията по хоризонталната. Цветните окръжности показват къде дадено отношение пресича даден модул. Това са така наречените точки на свързване. Точките на свързване могат да бъдат на нивото на модула (клас) или да са по-рафинирани и да взаимодействат с под-части на модула (атрибути, операции). Пресичането може да бъде идентифицирано лесно като се проследи вертикалата на дадено отношение. Сплитането може да бъде забелязано чрез проследяване на хоризонталата на даден модул.



Фигура 4 – Пресичане, сплитане и точки на свързване

2.3 Аспектна декомпозиция

Очевидно дадена дизайн-задача може да има пресичащи се отношения, с които конвенционалните механизми за абстракция не могат да се справят. AOCP предоставя изрични абстракции за представяне на пресичащи се отношения, наречени още аспекти ***[19]***. По този начин дадения дизайн-проблем се представя като множество отношения, локализирани в различни модули, плюс отношения, които пресичат дадено подмножество от модулите (фигура 5).



Фигура 5 – Разделяне на аспектите

За да се укажат точките, където аспектите се пресичат, се използва така наречената спецификация на пресечните точки. Спецификацията на пресечните точки е по същността си предикат върху цялото множество от точки на свързване, които даден аспект може да пресече. Спецификацията на точка на пресичане може да изброи точките на свързване или да предостави по-гъвкав начин за тяхното идентифициране.

Спецификацията на точките на свързване носи информация за това кои точки на свързване са пресечени от даден аспект, но не конкретизира по никакъв начин какво точно поведение е нужно в тези точки. За тази цел се въвежда концепцията за *съвет* (advice). Съветът е поведение, което може да бъде прикрепено преди (before), след (after) или вместо (around) дадена точка на свързване, като това изцяло зависи от спецификацията на точката на прекъсване.

Разгледаните по-горе концепции са вече широко утвърдени и общоприети от АОСР общността. Различни реализации на АОП разработки съществуват и въпреки факта, че всяка от тях представя по различен начин аспектите, точките на прекъсване и съветите, основните концепции остават същите.

2.4 Перспективи и проблеми пред АОП и АОСР

До момента разглеждахме основните концепции на АОП. Имайки предвид, че АОСР обхваща не само програмирането, а и целия производствен процес, то не е неразбираема и основната концепция на АОСР. Тя гласи, че можем да открием пресечени отношения и в другите фази на проекта, съответно – можем

да приложим вид аспектна декомпозиция, така че да модуляризираме успешно отношенията.

Самата област, която се занимава с идентифицирането на аспекти в другите фази на софтуерния процес, се нарича ‘Ранни аспекти’ (Early aspects). Основни цели и тема на изследвания са:

- идентифициране на пресечени отношения във всяка фаза на процеса – анализиране на изисквания, архитектура, дизайн, реализация, доставка (deployment), документация и т.н.;
- идентифициране на подходящи аспекти, с които може да се модуляризират откритите пресечени отношения в дадената фаза на проекта.

Разпространено схващане за момента е, че аспектите са предназначени само и единствено за решаване на проблеми по време на реализацията на проекта. Реалността обаче е съвсем различна. Дори да приложим аспектна декомпозиция в реализацията, обосновката на новите абстракции е нетривиална задача, защото:

- изискванията са произволно структурирани;
- обектният модел, резултат от дизайн фазата, няма връзка 1:1 със съответните изисквания;
- самите дизайн методологии (или поне масово разпространените) не поддържат основните абстракции – аспекти, съвети, точки на свързване, пресечни точки (pointcuts).

Постигайки основната цел, а именно – да сме открили аспектите във всички фази на проекта, то ще имаме изключително консистентен процес, в който проследяемостта на дадена функционалност от изискванията, през дизайн, реализация и документация ще бъде изключително проста и лесна задача. Всичко това води до повишаване на качеството (чрез намаляване на грешките, резултат от неправилно съпоставяне на елементите от различните фази на проекта) и много по-лесна поддръжка.

Друга област на изследвания са формалните методи по отношение на АОП семантиката. Въпреки кратката си история, вече има голям набор от АОП разработки за най-различни езици, среди, платформи и т.н. Поради липса на стандарти, всяка от тях дефинира своя собствена семантика по отношение на аспектите, точките на свързване и самия процес на свързване. Всичко това води

до сериозни проблеми от гледна точка на несъвместимост между тези разработки. Съвсем различен проблем, от много по-сериозно естество е, че въпреки някоя конкретно-дефинирана семантика на свързването на аспектната и базовата функционалност, не е възможно да се верифицира дали дадена програма е коректна спрямо конкретен набор от изисквания. Същия проблем допълнително се усложнява ако трябва да се добави нов аспект към дадена система – имаме ли гаранции, че той няма да повлияе на части от системата, които ние не сме желали да бъдат засегнати? Още по-лошо – възможно ли е новият аспект да повлияе на други, вече интегрирани аспекти? С решаването на всички тези проблеми са се заели група от изследователи, които се опитват с формални методи да дефинират подходяща семантика и да създадат модели, с помощта на които да може лесно да се верифицират програми, плод на АОП и АОСР.

2.5 Обобщение

Счита се, че в момента популярността на АОП и АОСР е същата, като популярността на ООП преди 20-30 години. Въпреки множеството разработки и напредък в областта, редовите програмисти и големите фирми гледат някак с недоверие на тази технология и все още избягват да я използват в индустриални проекти. Тази ситуация обаче се променя и няма да е далече моментът, когато АОП ще издържи теста на времето и ще се наложи като наследника на ООП на масовия, комерсиален пазар.

3 Microsoft .Net като платформа за АОСР

В тази глава ще разгледаме по-детайлно самия .Net фреймуърк и текущите .Net разработки в областта на АОП. Главата започва с кратък преглед и оценка на .Net и съпътстващите я средства и библиотеки, като среда за разработка. В последствие ще разгледаме текущите АОП разработки в .Net и ще ги съпоставим с Aspect .Net.

3.1 .Net Framework

Инициативата за появата на .Net датира от 1997, когато Майкрософт взима решението за разработка на платформа, която да улесни програмистите, предоставяйки хомогенна среда за разработка, даваща на софтуера свобода от гледна точка на операционната система (ОС) и езика, на който се програмира. Най-общо казано терминът *фреймуърк (framework)* представлява множество типове, класове, услуги, сървери, инструменти и др. Платформата .Net е проектирана така, че да разреши някои от проблемите и ограниченията на програмирането за операционната система Windows – най-вече сложността за разработка, поддръжка и разпространение на софтуера. Някои от целите на .Net са:

- нова хомогенна платформа за разработка – предоставя хомогенна среда за разработка, както на десктоп, така и на интернет и разпределени приложения;
- опростяване имплементирането на приложения – предоставя богат набор инструменти и библиотеки, които улесняват и ускоряват разработката на софтуер;
- подобряване на взаимодействието и интеграцията на бъдещи и налични системи.

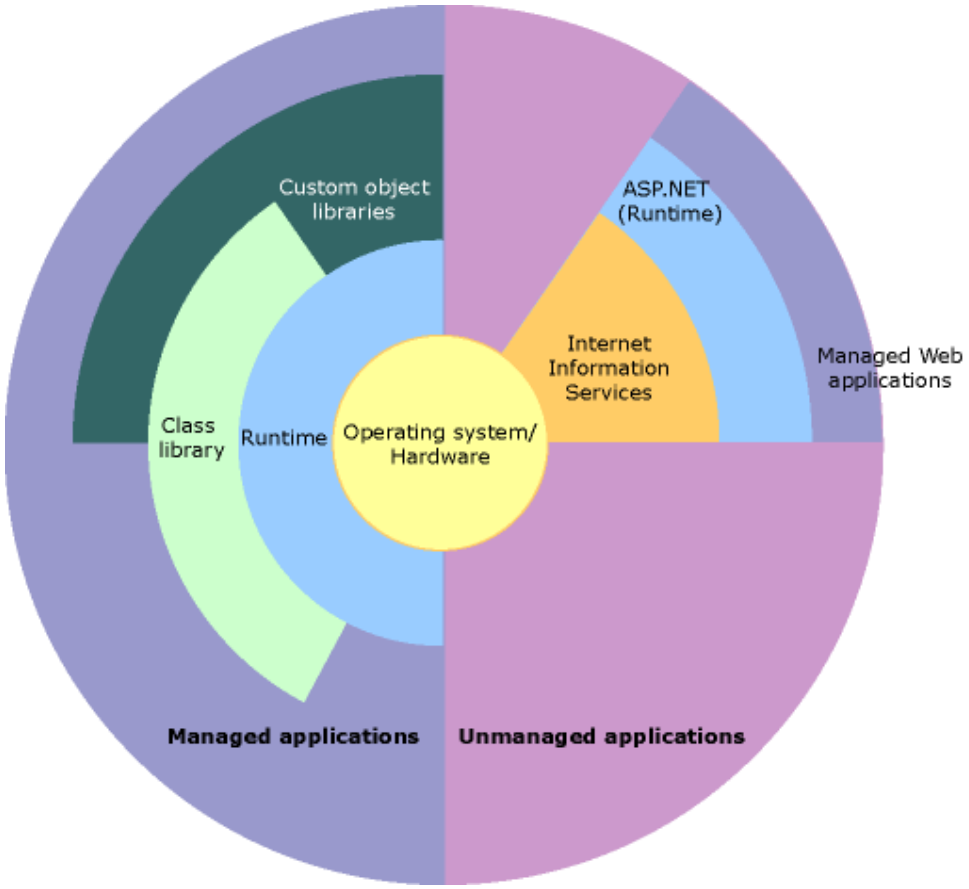
3.1.1 Архитектура

.Net фреймуъркът е изграден от два основни компонента: обща изпълнима среда (ОИС) (Common Language Runtime) и основна библиотека с класове (ОБК). ОИС лежи в основата на .Net. На него може да се гледа като на агент, който контролира кода по време на изпълнение – т.е. предоставя базови услуги като управление на паметта, нишките, разпределеност на изпълнението,

безопасност на типизацията, сигурност и оптимизираност на кода и т.н. Всъщност управлението на кода е основа на ОИС. Код, който ще бъде изпълним от ОИС се нарича *контролиран* (managed), докато този, който не е предназначен, *неконтролиран*.

ОБК, другият важен компонент на .Net фреймуърк, е изчерпателна обектно-ориентирана съвкупност от типове, които могат да бъдат многократно използвани за разработка на софтуер, вариращ от прости приложения, използващи командния ред, до сложни бизнес разработки, които са разпределени върху широк набор от мрежи и протоколи.

Диаграмата на фиг.6 показва връзката между ОИС и ОБК от една страна, и приложенията и цялостната система от друга. От диаграмата също може да се види как контролираният код се вписва в по-големи архитектури.



Фигура 6 – Връзка между ОИС и ОБК

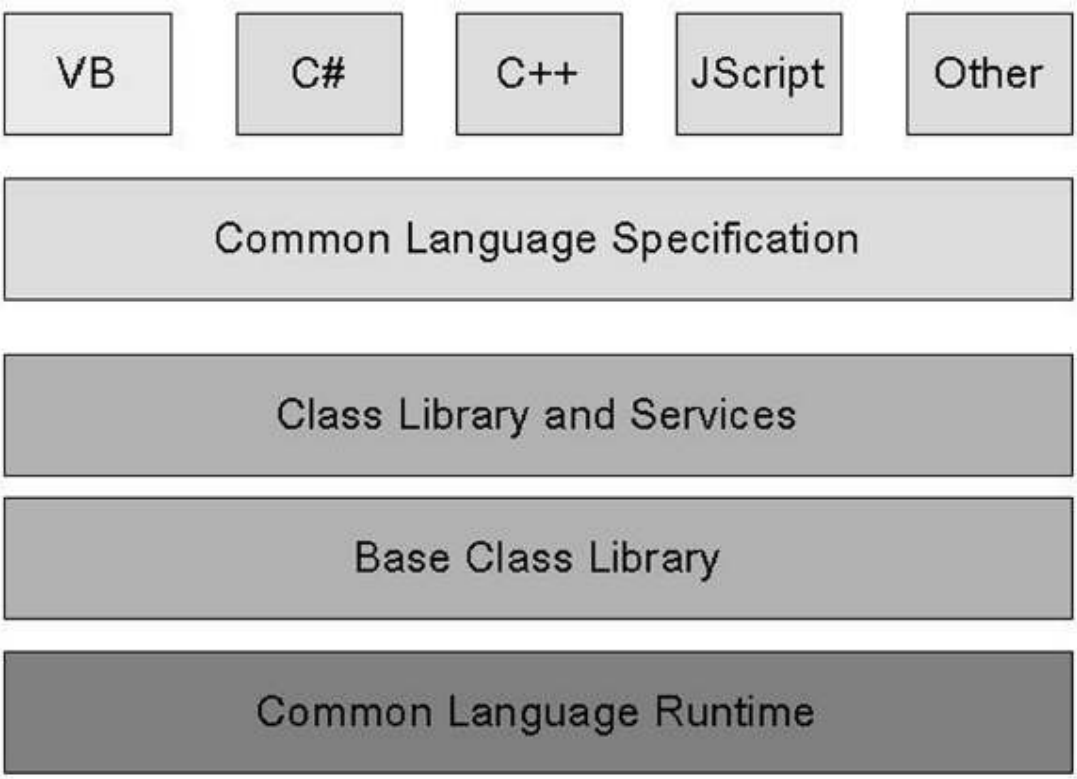
3.1.2 Езици за програмиране

Някои от текущо поддържаните програмни езици от .Net фреймуърка са:

- C# - най-популярният и употребяваният език в света на .Net;
- VB.NET – втори по популярност език, който въпреки, че носи името на предшественика си Visual Basic, всъщност няма много общо;
- C++ - разширение на C++, поддържащо .писане и компилиране на Net код;
- J# - вариант на Java, написан от Майкрософт за .Net платформата;
- Jscript – вариант на Jscript за .Net (повече информация може да се намери на адрес: http://www.gotdotnet.com/team/jscript/);
- Eiffel# – версия на популярния език Eiffel за .Net (повече информация може да се намери на адрес: http://archive.eiffel.com/doc/manuals/technology/dotnet/eiffelsharp/white_paper.html) ;
- CobolNet – разработка на Fujitsu на широко популярния в миналото език Cobol за .Net платформата (повече информация може да се намери на адрес: http://www.netcobol.com/products/windows/netcobol.html) ;
- #SmallTalk – преработка на езика SmallTalk, целяща платформата .Net (повече информация може да се намери на адрес: http://www.refactory.com/Software/SharpSmalltalk/);
- Perl – вариант на Perl за .Net (повече информация може да се намери на адрес http://www.activestate.com/Products/Perl_Dev_Kit/?_x=1);
- Pascal – вече има няколко разработки на езика Pascal за .Net. Една от тях може да бъде намерена на адрес: http://plas.fit.qut.edu.au/gpcp/NET.aspx;
- Python – разработката на Python за .Net може да бъде намерена на адрес: http://pythonnet.sourceforge.net/;
- Oberon – за повече информация по отношение на разработката на Oberon за .Net може да бъде намерена на адрес: http://www.bluebottle.ethz.ch/oberon.net/;
- Haskell – версия за .Net на небезизвестния език Haskell (повече информация може да се намери на адрес: http://www.cin.ufpe.br/~haskell/haskelldotnet/);
- Mercury – повече информация за тази разработка може да се намери на адрес: http://www.cs.mu.oz.au/research/mercury/dotnet.html;
- Scheme – версия за .Net на популярния функционален език Scheme (за повече информация: http://www.cs.indiana.edu/~jgrinbla/index.htm);
- Fortran – повече информация за версията на Fortran за .Net може да бъде намерена на адрес: http://www.lahey.com/dotnet.htm .

По-детайлен списък на всички поддържащи .Net към момента езици, може да бъде намерена на адрес: http://www.dotnetpowered.com/languages.aspx

Фигура 7 дава нагледна представа за това как се вписват някои от езиците в цялостната архитектура на .Net



Фигура 7 – Място на различните езици в. Net архитектурата

Както се вижда от фигура 7, това което позволява да имаме различни езици, които са взаимно съвместими в .Net е Общата Езикова Спецификация (ОЕС). Тя представлява множество правила, които гарантират гореспоменатата съвместимост. Останалата част от компонентите ще разгледаме в по-големи детайли в следващите секции.

3.1.3 Обща изпълнима среда (Common Language Runtime)

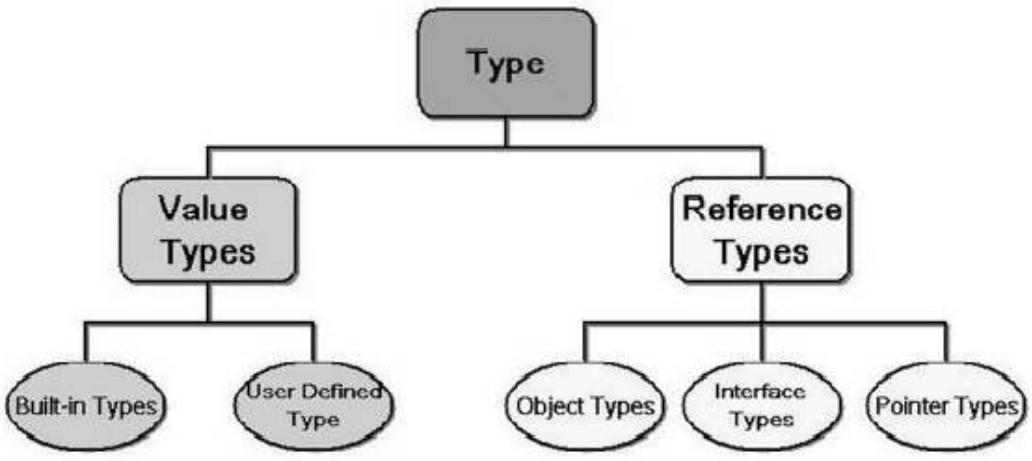
Общата Изпълнима Среда (ОИС) предоставя обща среда за изпълнение на програми, написани на .Net платформата. Тя е много наподобява Java Runtime Environment (JRE) в езика Java. Както вече споменахме, код който се изпълнява под контрола на ОИС, се нарича *контролиран*. ОИС изпълнява широк набор контролиращи дейности върху голям брой ресурси – памет, код,

своевременна компилация до натурален (за ОС) код, многонишковост и т.н. Кодът, който се изпълнява извън ОИС се нарича *неконтролиран*. Въпреки това неконтролираният код може да бъде интегриран и изпълняван от ОИС.

ОИС може да бъде разделена на 3 основни категории:

- Система на типовете (СТ)
- Система за изпъление (СИ)
- Система за метаданни (СМ)

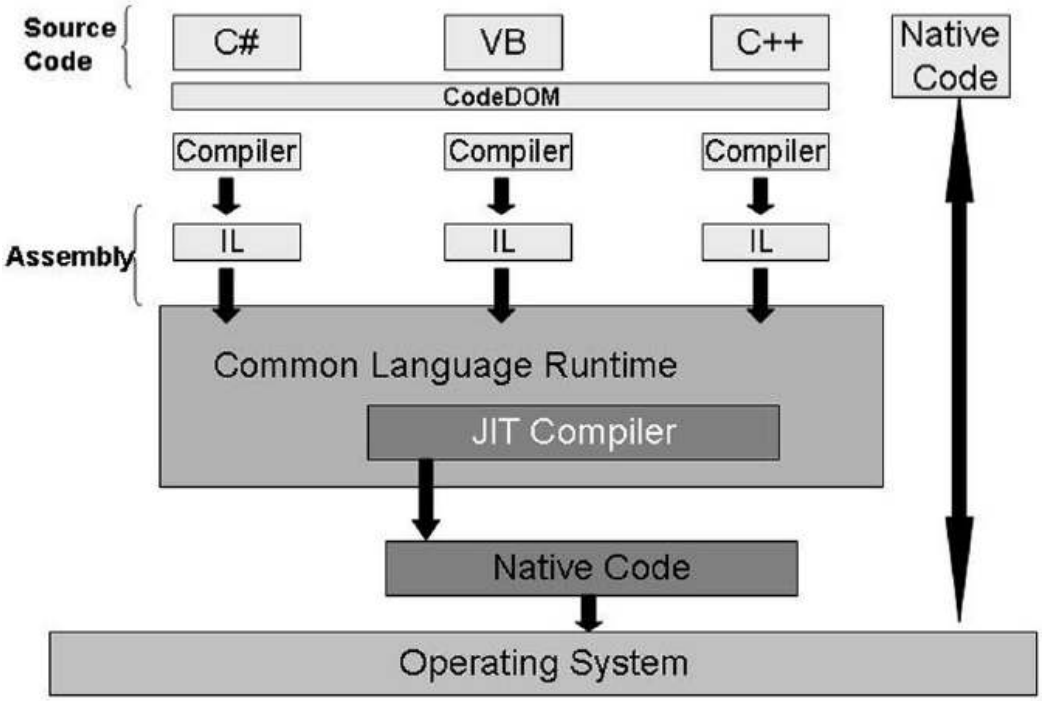
СТ е изградена от Обща Типова Система (ОТС) и Общоезикова Спецификация (ОЕС). Типовете в .Net са типове-стойности (value) и типове-референции (reference). Типове-стойности са вградените типове (int, double и т.н.) или дефинирани от потребителя – структури, изброими типове. Типове-референции са дефинирани от потребителя класове и интерфейси. Фигура 8 илюстрира йерархията на типовете в .Net. ОТС предоставя на програмиста стандартните типове-стойност и типове-референции, налични в .Net фреймуърка.



Фигура 8 – Йерархия на типовете в ОТС

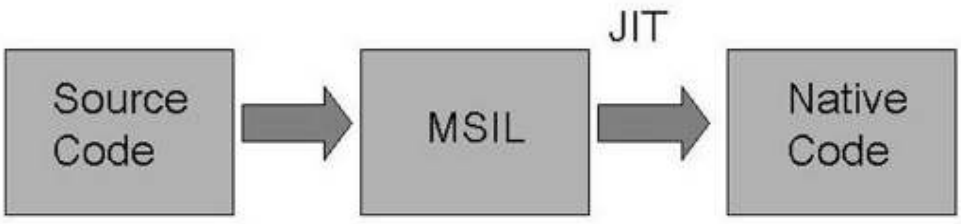
ОЕС изпълнява също много важна роля – дефинира множеството от правила, които даден език е задължен да удовлетворява за да могат програми, писани на него да бъдат изпълнявани от ОИС. Като допълнение, ОЕС предоставя набор от изисквания за съвместимост, които се използват за определяне дали даден език е съвместим с ОЕС. *Ако даден език е съвместим с*

ОЕС, то програми писани на него могат да бъдат изпълнявани от ОИС, както и от други съвместими с ОЕС .Net езици.



Фигура 9 – Модел на изпълнение в .Net

Системата за изпълнение е сърцевината на ОИС. Код, написан на който и да е от съвместимите с .Net езици се компилира до междинен език, наречен Microsoft Intermediate Language (MSIL). По време на изпълнение, MSIL кодът се компилира своевременно до машинен код. Този междинен код е факторът, който позволява на програмите, написани на .Net да „вървят" без ограничения от операционната система или езика, на който са написани (стига да е ОЕС съвместим). Подробният модел на изпълнение е илюстриран на фигура 9, докато фигура 10 показва опростена схема на компилирането в .Net.



Фигура 10 – Компилацията при .Net

Системата за метаданни (СМ) е неразделна част от .Net фреймуърка. Основната градивна единица на всяко .Net приложение е така нареченото асембли. В по-старите версии на Windows, по-голямата част от кода на програмите се намираше във файлове с разширение DLL (библиотека с динамично свързване). DLL файл, който съдържа в себе си контролиран код се нарича асембли. Асемблитата са аналог на JAR файловете в Java. Едно .Net асембли съдържа следните елементи:

- Манифест – описва името и версията на асемблито;
- Метаданни за типовете, дефинирани в асемблито;
- Модули – междинният код, който се компилира точно преди изпълнение;
- Ресурси – изображения, текст и т.н. – даващи възможност за пълнота и самодостатъчност на асемблито.

От гледна точка на самите приложения, асемблитата изпълняват следните важни функции:

- съдържат кода, необходим за изпълнение в ОИС;
- определят граници на сигурността (правата биват отдавани или отхвърляни на това ниво);
- определят граници на референтност (манифестът на всяко асембли се ползва за разрешаване на конфликти);
- определят граници на версията (асемблито е най-малката единица в .Net, която може да притежава версия);
- определя гранулярността на изходния код, която бива пакетирана и подготвяна за последващо разпространение (асемблито е най-малката единица на разпространение на програмен код).

Имайки предвид всичко изложено до момента, можем да направим извода, че .Net е подобрено продължение на Windows платформата, позволяващо лесна, бърза и удобна разработка, поддръжка и разпространение на приложения, които имат възможността много по-лесно да комуникират помежду си и да се интегрират.

3.1.4 Основна библиотека с класове

ОБК е колекция от типове за многократно използване, които тясно се интегрират с ОИС. ОБК е обектно-ориентирана, като по този начин

предоставяните типове могат директно да бъдат използвани в клиентски приложения с контролиран код. Това прави .Net класовете не само лесни за употреба, но и намалява времето необходимо за научаване на евентуални нововъведения във фреймуърка.

Както се очаква от една обектно-ориентирана библиотека, ОБК улеснява реализирането на широк набор тривиални задачи в програмирането като: управление на низове, връзка с релационна база данни, достъп до файлове и др. Като допълнение на класовете, реализиращи гореспоменатите тривиални задачи, са налични голям брой компоненти, предоставящи репшения на най-разнообразни сценарии. Като пример ще изброим различните видове приложения и услуги, които се реализират изключително лесно с .Net:

- конзолни приложения,
- Windows приложения с потребителски интерфейс,
- ASP .Net приложения,
- XML Web услуги,
- Windows услуги.

За всеки един от изброените типове съществува съответна под-библиотека, предоставяща огромен набор готови класове, които правят разработката на съответното приложение лесна и удобна, като по този начин оставя програмиста да се концентрира върху самата задача, вместо върху начина по който ще я реализира.

3.1.5 Модел за автоматизация на MS Visual Studio 2003

Въпреки, че Visual Studio 2003 предоставя на разработчиците много инструменти и способи да решат всяка една задача или проблем, съществуват някои дейности, които биха изискали по-голям контрол от страна на програмиста спрямо средата на разработка. За тази цел Visual Studio 2003 предоставя богат модел за автоматизация ***[5]*** на интегрираната среда за разработка (ИСР), познат като Модел за автоматизация (МА). МА улеснява разработването на разширения и автоматизирането на средата като цяло.

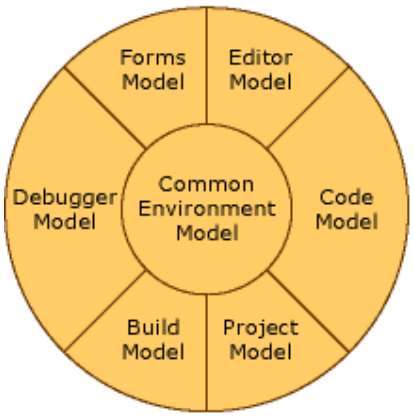
ИСР в Microsoft Visual Studio 2003 се състои от известен брой взаимосвързани инструментни прозорци като прозореца на решението (Solution Explorer) и редактора на код. МА се състои от богат набор обекти и интерфейси, които позволяват на разработчика да контролира, манипулира и автоматизира

тези прозорци, както и друга, свързана със средата информация – конфигурации за построяване, редактиране на код, дебъгване и др.

Пример за някои от предоставените обекти и интерфейси:

- обектът TaskList преставлява прозореца със списък от задачи;
- обектът Toolbox представлява кутията с инструменти;
- обектът ToolboxTab представлява tab в прозореца на кутията с инструменти;
- обектът ToolboxItem представлява елемент от кутията с инструменти.

МА е изграден от няколко подмодела, всеки един от които обхваща важна област от функционалността на средата. МА е илюстриран графично на фигура 11. Важно е да отбележим, че въпреки показаните разделения, самата сърцевина на модела и всички под-модели са общи и достъпни за всички .Net съвместими езици.



Фигура 11 – Модел за автоматизация на VS 2003

Таблица 1 предоставя описание за най-важните характеристики на всеки един от подмоделите на МА.

Име на подмодел	Описание
Common environment model	Обхваща по-голямата част от МА. Включва обекти за контролиране на ИСР (инструментни и документни

	прозорци, менюта и т.н.), събития и др.
Project model	Включва общи и специфични за .Net езиците обекти за контролиране на решението (Solution), проектите и елементите в тях. Общите обекти за решението/проекта работят с всеки VS .Net проект, докато специфичните обекти са предвидени да работят с конкретен .Net език.
Editor model	Включва обекти за манипулиране на текста в редактора на ИСР – добавяне, изтриване, форматиране, местене и т.н.
Code model	Включва обекти за намиране и редактиране на програмен код и елементи от проекта. Този модел предоставя класовете, интерфейсите, структурите и др., както и членовете на тези типове.
Forms model	Включва обекти за контролиране на Windows Forms (библиотека за потребителски интерфейс в .Net) и техните характеристики. Не включва обекти за Web Forms – библиотека за потребителски интерфейс в интернет приложения, базирани на ASP.Net.
Debugger model	Включва обекти за контролиране на дебъгера на VS 2003.
Build model	Включва обекти за контролиране на операциите по построяване на решението/проекта. Позволява програмно да се конфигурират дебъг настройките, както и да се създават автоматизирани компилации на продукта.

Таблица 1 – Подмодели на МА във VS 2003

Въпреки, че МА е изграден от подмодели, самият модел е сам по себе си интегриран, въпреки че някои от подмоделите са реализирани в отделни асемблита.

3.2 Текущи АОП разработки в .Net

В следващите няколко секции ще разгледаме в подробности няколко АОП разработки в областта на .Net ***[6]***. Описанието на начина, по който са реализирани и по който се ползват от разработчиците, ще ни помогне да идентифицираме изискванията, които Aspect.Net трябва да изпълнява, за да бъде максимално приложим и удобен за работа.

3.2.1 AspectSharp

AspectSharp е част от проекта Castle [9], който обхваща няколко разработки в различни направления, част от които зависят една от друга. Самият проект е реализиран без въвеждане на разширения към някой от .Net езиците. Аспектите се реализират като стандартни класове, които обаче трябва да реализират конкретен интерфейс, за да могат да осъществяват достъп до извикващия ги основен код. Конфигурацията на аспектите се извършва чрез допълнителен език със синтаксис, близък до този на езика Ruby. Ето как изглежда дефинирането на pointcut, който обхваща всички методи, казващи се RetrieveContent и приемащи всякакви параметри:

```
import YourCompany.CMS.ContentProviders in YourCompanyAssembly
import YourCompany.CMS.Aop.Interceptors

aspect SecurityAspect for RSSContentProvider
    include Mixins.SecurityResourceImpl in MyMixinsAssembly

    pointcut method(* RetrieveContent(*))
        advice(SecurityCheckInterceptor)
    end
end
```

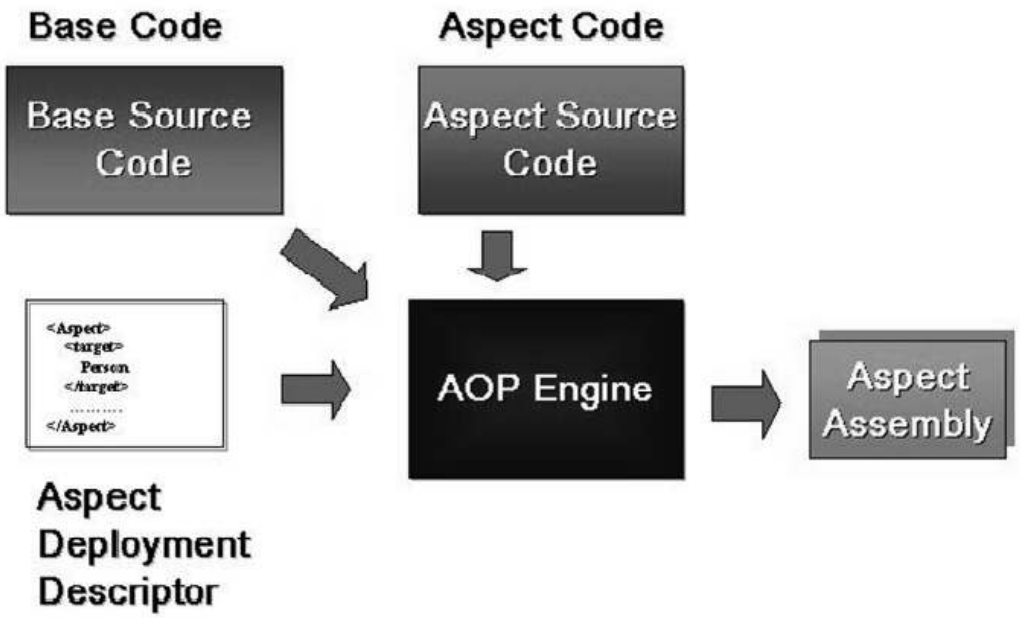
Свързването е на ниво метод – т.е. съветите биват извиквани преди и/или след метод, който отговаря на дефиницията на pointcut. AspectSharp не предлага компилатор и съответно статично свързване на аспектите. Извикването на аспектите се реализира непрозрачно за програмиста – т.е. чрез използване на

предоставени от AspectSharp класове, които динамично, посредством генериране на динамично прокси, вмъкват кода, който извиква съответните съвети, където и когато трябва.

3.2.2 AspectC#

AspectC# е АОП подход в C#, като в по-голямата си част се базира на аналогичния си Java еквивалент – AspectJ. Една от целите на проекта е АОП концепциите да бъдат използвани без да се налага въвеждането на допълнителни конструкции или езици, с които да бъдат описвани или конфигурирани аспектите / съветите. В момента AspectC# работи само с езика C# като се очаква бъдещите негови версии да поддържат всички .Net езици. Свързването се извършва само статично – т.е. по време на компилация. Конфигурирането на самото свързване се реализира в XML файл по подобие на AopDotNetAddIn.

Фигура 12 илюстрира архитектурата на AspectC#:



Фигура 12 – Архитектура на AspectC#

Основен компонент в AspectC# е AOP Engine, което е всъщност и ядрото, реализиращо свързването на базовата и аспектната функционалност. Тъй като AspectC# оперира на ниво сорс код, не случайно на диаграмата имаме 3 входа –

базовия сорс код, аспектния сорс код и XML файла, описващ свързването. Вече свързаният изход на AOP Engine е аспектирано и компилирано асембли.

3.2.3 Eos

Eos е аспектно разширение на програмния език C#, част от Microsoft .Net Framework. В основата си Eos е базиран на Aspect-J. Eos се стреми към подобряване на съществуващия АОП модел на езиците за програмиране в 3 основни направления ***[10]***. Първо – обобщава инстанцирането на аспектите и свързването на съветите с цел елиминиране нуждата от заобиколни решения (workarounds), които са неизбежни в повечето налични разработки. Второ – генерализира join point модела. Трето – цели елиминирането на разликите между класовите и аспектните конструкции в полза на един нов концептуален градивен блок, който комбинира експресивните възможности на класовете и аспектите, получавайки силно подобрена концептуална интеграция и унификация в дизайна на езика.

Eos поддържа както статично, така и динамично свързване на съветите. Езикът предлага набор от ключови думи, разширение на C#, с които се описват самите аспекти, начинът на свързване и т.н. След като аспектите са вече написани, самото свързване се извършва от Eos компилатора, който генерира съответния код, резултатен на свързването на съветите. След това изпълнимата среда на Eos има грижата да извършва динамично свързване на съвети, както и да поддържа стандартна информация – метаданни за точките на свързване и т.н.

По отношение на интеграцията на Eos в средите за разработка – единственият възможен начин за момента е чрез компилатора на Eos. Самият компилатор изпълнява освен рутинните си задължения, свързани с разпознаването и свързването на аспектния код и ролята на прокси към истинския C# компилатор на Microsoft. Това само по себе си е сериозно ограничение, тъй като само една малка част от опциите на стандартния компилатор се поддържат от компилатора на Eos. Съответно това ограничение е голям проблем за използването на Eos в реални условия, тъй като по-голямата част от настройките на компилатора няма как да се зададат. Друг текущ проблем на Eos е фактът, че поддържа само код, написан на C# - т.е. неупотребяем е за другите .Net базирани езици.

3.2.4 AopDotNetAddIn

AopDotNetAddIn представлява add-in за Microsoft Visual Studio 2003, който предоставя възможност за използване на АОП техники в ежедневните практики. AopDotNetAddIn не предоставя нововъведения в езика под формата на нови конструкции или ключови думи ***[7]***. За сметка на това аспектите се реализират като наследници на базов клас, а съветите - като член-функции на класа на аспекта. Информацията, необходима за свързването на съветите с останалата част от програмния код се реализира чрез XML файл, съдържащ цялата необходима за свързването информация.

В общи линии структурата на XML файла е:

```
<aspects>
        <aspect name="..." code="..." weave='true|false'>
                <advices>
                        <advice type="before|after|around">
                                <method name="..." />
                                <pointcut>
                                        <call>
                                                <method  name='...'
                                                        class='...'
                                                        access='...'
                                                        return='...' />
                                        </call>
                                </pointcut>
                        </advice>
                </advices>
                <introductions>
                        <member name="..." introduceto="..."/>
                        <method name="..." introduceto="..."/>
                        <interface name="..." introduceto="..." />
                </introductions>
        </aspect>
</aspects>
```

Както виждаме, писането на самия XML файл е бавна, не лесна и често извор на грешки дейност, която е често в основата на всички проблеми, свързани с AopDotNetAddIn.

3.3 Обобщение

В тази глава разгледахме основните концепции залегнали в основата на АОП. Основните компоненти и идеи на Microsoft .Net framework също бяха предмет на разглеждане и анализ. Представихме също и най-разпространените АОП разработки по отношение на .Net платформата. Най-общо можем да групираме тези разработки по следните основни характеристики:

- разширяват ли някой .Net език, въвеждайки нови ключови думи и/или конструкции;
- как е реализирано конфигурирането на аспектите, т.е. по какъв начин се указват самите joinpoints, pointcuts и самото свързване;
- поддържат ли всички .Net езици.

По отношение на първият фактор – повечето разгледани разработки са реализирани без допълнителни въведения в езика. Това разбира се не е без причина. Едно разширение към езика би изисквало съответен компилатор, който пък от своя страна в по-голямата част от времето би играл ролята на прокси към стандартния компилатор за дадения език, предмет на АОП. Това само по себе си ограничава ползването на АОП реализацията, а също и затруднява реализацията й и нейното ползване. Точно това е случаят с единствената .Net реализация, която въвежда допълнителни ключови думи и конструкции към C#, а именно Eos.

Според това как е реализирано конфигурирането на аспектите, можем да идентифицираме два основни подхода – чрез използването на XML или чрез въвеждане на нов език (най-често наподобяващ синтаксиса на AspectJ). Всеки от тези подходи има своите предимства и недостатъци. Използването на нов език обикновено налага дефинирането на граматика и реализирането на парсер, който разчита и верифицира тази граматика. Това само по себе си е сериозна задача, която все пак не е предмет на повечето АОП разработки. Употребата на XML пък води до неудобства по отношение на писането и поддържането му.

Друг важен факт, който трябва да отбележим е, че всички разгледани АОП разработки са реализирани единствено за езика C#.

Имайки предвид всичко казано по-горе и целите, които си поставя Aspect.Net, можем да изтъкнем следните негови преимущества пред останалите реализации:

- Aspect.Net не предлага разширения на никой от .Net езиците, което го прави независим от самите езици;
- Aspect.Net предлага само статично свързване на аспектите, което с други думи казано означава, че Aspect.Net работи единствено на нивото на MSIL. Това като следствие доказва, че Aspect.Net е наистина независим от конкретните .Net езици, което пък води до факта, че самите аспекти и кодът, който е предмет на АОП могат да бъдат написани на различни .Net езици;
- Конфигурирането на аспектите в Aspect.Net се извършва чрез XML файлове, което улеснява значително реализацията на проекта. Това предоставя възможност на програмиста сам и на ръка или с подходящи инструменти за автоматизация да модифицира конфигурацията на аспектите в приложението спрямо съответните нужди.

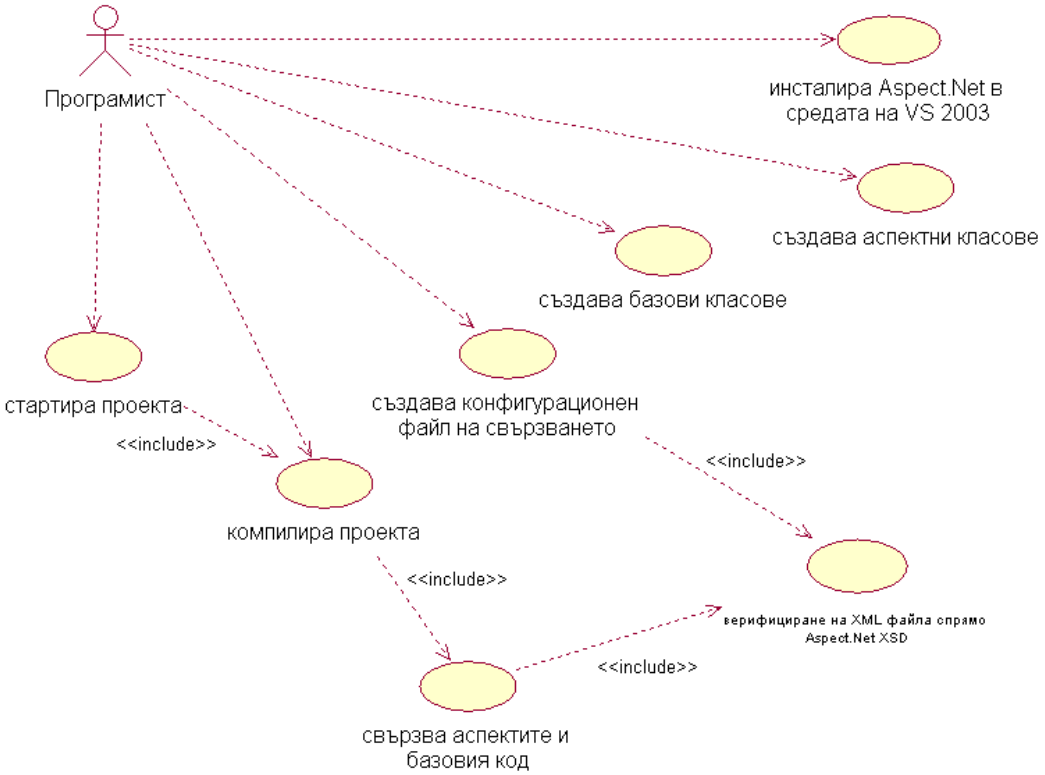
Като резултат можем смело да кажем, че Aspect.Net превъзхожда ако не всички, то поне повечето от разгледаните реализации.

4 Изисквания и дизайн

В тази глава ще разгледаме по-подробно изискванията, които Aspect.Net трябва да удовлетворява. Ще бъдат разгледани както функционалните, така и нефункционалните изисквания. Също ще опишем и подхода, който бе избран при разработката, както и мотивация за избора му. Като допълнителни материали към тази глава (Приложение 2), може да се намери кратко описание на библиотека, която ще се използва при разработката на проекта, както и обяснение на основните нейни характеристики и факторите, които налагат използване й. В Приложение 1 са представени UML диаграми на решението, съпроводени с описание на основните класове, компоненти и взаимодействия.

4.1 Изисквания

Функционалните изисквания могат да бъдат представени със следната диаграма (фигура 13), илюстрираща различните случаи на употреба на Aspect.Net от гледна точка на програмистите.



Фигура 13 – Диаграма на случаите на употреба (Use-Case diagram)

Вземайки предвид факта, че Aspect.Net ще представлява средство, ползвано от програмистите в ежедневната им практика, фигура 13 изброява минимално-нужната функционалност от гледна точка на програмист, ползващ системата, а именно:

- Aspect.Net трябва да предоставя инсталатор, с който автоматично да се осъществява интегрирането на продукта с MS Visual Studio 2003;
- Програмистът трябва да може да създава аспектни класове;
- Програмистът трябва да може да създава базови класове;
- Програмистът трябва да може да създава XML конфигурационен файл, с който да указва кои аспектни и кои базови функционалности ще бъдат свързани;
- Програмистът трябва да може да компилира проекта по добре познат за него начин – т.е. Aspect.Net не трябва да променя по никакъв начин стандартния процес във MS Visual Studio 2003;
- Като част от процеса на компилиране, Aspect.Net ще осъществи свързването на базовата и аспектната функционалност, използвайки XML конфигурационния файл;
- Както при компилирането, така и след създаването на XML конфигурацията, XML файлът ще удовлетворява стандартната XSD схема, която ще се разпространява като част от Aspect.Net.

Изхождайки от случаите на употреба, а също и като резултат от анализа на съществуващите разработки (силните и слабите им черти), ние достигнахме до следните функционални и нефункционални изисквания, които Aspect.Net трябва да удовлетворява:

Функционални изисквания

1. Aspect.Net трябва да улеснява програмиста по отношение на инсталирането и конфигурирането на самия Add-In. За удовлетворяване на това изискване, Aspect.Net трябва да предоставя на потребителите лесен и интуитивен за ползване инсталатор.
2. Aspect.Net трябва да позволява на програмиста да модуляризира пресичащи се отношения в .Net съвместимите езици.

3. Aspect.Net не трябва да прави никакви нововъведения в някой от .Net езиците. Това изискване се налага от гледна точка на:
   3.1. Улесняване на употребата на АОП от .Net програмистите, тъй като няма да им се налага да учат нови езикови конструкции.
   3.2. Улесняване на разработката на Aspect.Net – избягва се нуждата от разработването на допълнителни компилатори, т.е. ще може да се ползват стандартните.
4. Въпреки, че Aspect.Net ще бъде по-скоро демонстрация, доказваща приложимостта на АОП техниките в програмирането, и в частност в .Net платформата, Aspect.Net трябва да бъде написан по такъв начин, че да може да бъде разширяван и дообогатяван в последствие.
5. Системата не трябва да променя базовите класове. Вместо това Aspect.Net ще ползва тях и класовете на аспектите като входни данни, а самото свързване на аспектите ще става в системата, избягвайки каквито и да е промени по базовите класове. Идеята за това изискване в случая е, че базовите класове трябва да могат да се ползват и извън контекста на Aspect.Net.
6. Aspect.Net не трябва да изисква аспектите да бъдат реализирани по начин, различен от стандартните ООП похвати – т.е. в Aspect.Net аспектите се реализират като стандартни класове. Единственото, което ги различава от стандартните класове е използването на атрибут ‘Aspect’.
7. Единственото, което различава нормалните методи от съветите в даден аспектен клас е използването на атрибута ‘Advice’.
8. Аспектите и базовите класове са напълно независими компоненти – т.е. няма никаква зависимост от гледна точка на функционалност, интерфейс и т.н.
9. По отношение на дефиницията на pointcut и самото свързване на аспектите с базовата функционалност, изискванията са:
   9.1. Трябва да се поддържат трите основни типа свързване на аспектите, а именно: before, after и around.
   9.2. Идентифицирането на точките на свързване трябва да е възможно по следните критерии:
      9.2.1. Модификатор на достъп – public, private, protected;
      9.2.2. Име на метод. При търсенето по име, трябва да се поддържа използването на *. Дефиницията на използването на този знак е:

<search_string> := <plain_string> | <plain_string>”*” | “*”

<plain_string> := 1*<any character excluding “*”>

9.2.3. Тип на върнатия параметър

9.2.4. Параметри на метода. Всеки търсен параметър, трябва да може да бъде дефиниран чрез:

9.2.4.1.Име на параметъра;

9.2.4.2.Тип на параметъра.

9.3. Дефиницията на даден pointcut трябва да може да бъде пре-използвана в рамките на конфигурацията на свързването.

10. По отношение на формалната коректност на свързването:

10.1. Всяко свързване от тип before или after, трябва вътрешно да се представя като подходящо свързване от тип around.

10.2. Към произволна точка на свързване (метод) трябва да могат да бъдат свързани множество съвети (advices), които в общия случай принадлежат към различни аспекти.

10.3. Когато даден аспект има един или повече съвети, свързани към методи от някой клас, то само една инстанция на аспекта, обслужваща дадения клас и всички негови инстанции, бива създадена.

Нефункционални изисквания

Качеството на даден софтуер се определя не само от степента, с която са удовлетворени клиентските изисквания по отношение на функционалността. Дори приложение, което успешно покрива целия набор от функционални изисквания може да има сериозни проблеми с използваемостта, скоростта на работа, лекотата на поддръжка, количеството на нужните му ресурси и т.н. С други думи – нефункционалните изисквания за разлика от функционалните, определят множество ограничения върху крайния продукт или резултат.

По отношение на Aspect.Net, нефункционалните изисквания са:

11. Aspect.Net трябва да бъде разработен на Microsoft Visual Studio .Net, чрез езика C#.

12. Aspect.Net трябва да бъде реализиран като допълнение (Add-In) към самата среда на MS Visual Studio .Net.

13. Aspect.Net трябва да бъде реализиран в рамките на 3 месеца.
14. Aspect.Net трябва да бъде съвместим с Microsoft .Net Framework v1.1.
15. Aspect.Net трябва да бъде лесен за употреба и научаване.
16. Aspect.Net трябва да се разпространява безплатно, т.е. да не ползва комерсиални или платени библиотеки, които биха изисквали специален лиценз за употреба

4.2 Възможни подходи за дизайн

Във трета глава разгледахме няколко от най-известните АОП разработки в областта на .Net. Както видяхме, има няколко основни критерия, по които можем да групираме съответните подходи. Следващата таблица ги илюстрира:

Критерий	**AopDotNetAddIn**	**EOS**	**AspectC#**	**AspectSharp**
Езици	Всички .Net съвместими езици	C#	C#	C#
Експресивна мощ	Аспекти + XML конфигурация	Собствен език, унифициран подход – класпект (клас + аспект)	Конструиране и деклариране на аспекти + XML конфигурация	Ruby-базиран език за дефиниране на точки на свързване.
Механизми за композиция	Няма нововъдения в езика. Аспектите се реализират като класове, реализиращи базов аспектен клас	Класпект е нова унифицираща единица, енкапсулираща отношенията. Реализира се с нова конструкция в езика.	Конструкции за деклариране на аспекти, съвети и свързване с базовите класове	Аспектите се реализират като обикновени класове, реализиращи интерфейс. Само точките на свързване се реализират с език, базиран на синтаксиса на Ruby
Имплементационни механизми	Оперира върху MSIL. Поддържа само статично свързване на	Поддържа статично и динамично свързване на	Без нововъведения в езика. Аспектите се прилагат в/у	Поддържа само динамично свързване на аспектите чрез

	аспектите.	аспектите. Оперира върху сорс код.	класовете с дескриптор, описващ съпоставянето	динамично прокси. Оперира върху MSIL

Таблица 2 – Различни подходи при АОП инструментите

4.3 Подходът при Aspect.Net

Aspect.Net ще бъде реализиран и ще работи изцяло върху MSIL. Пряко следствие на това е фактът, че като резултат ще поддържа всички налични .Net езици. Аспектите и базовите класове могат да бъдат писани на различни езици.
С оглед на краткия срок, Aspect.Net ще поддържа само статично свързване на аспектите, т.е. веднага след компилация на основната функционалност, ще бъде извършено сливането с аспектния код. За тази цел ще трябва да имаме удобен инструмент, с който лесно и удобно да можем да манипулираме MSIL кода, в частност:

- намиране на всички точки на свързване (методи), отговарящи на условията, дефинирани в XML конфигурацията;
- заменяне на извикванията към методи на базовите класове;
- извличане и предоставяне на контекст за средата на изпълнение на съветите, които биват извиквани.

Както вече споменахме в обобщението на трета глава (т. 3.4), въвеждането на допълнителни конструкции в някой от .Net езиците води до следните неудобства и проблеми:

- придобиване на зависимост от конкретният език – самият АОП инструмент става зависим от даден .Net език, от неговия синтаксис и т.н., което като резултат го прави неприложим за всички .Net езици;
- нужда да се реализира компилатор за конкретният език (езици), за които е разработен инструментът. Това от своя страна води до неудобства, защото компилаторът на АОП езика играе ролята на прокси спрямо истинския компилатор за дадения език, което прави инструмента силно статичен и тромав по отношение на начина на използване – т.е. не поддържа дадени ключове или настройки, или пък трябва да се рекомпилира за всяка различна (нова) версия на компилатора;

- не на последно място, реализирането на компилатор не е никак лесна или тривиална задача, която освен това не е цел на текущата дипломна работа.

С оглед на гореспоменатите причини, Aspect.Net няма да въвежда нови конструкции в никой от поддържаните .Net езици. Аспектите ще бъдат реализирани като обикновени класове, с единствената разлика, че тези класове, както и методите (съветите), които реализират дадена логика, трябва да бъдат маркирани със специални атрибути по време на дефинирането на класовете/методите. По този начин избягваме от необходимостта да задължаваме програмистите, които реализират аспекти, да наследяват от даден клас. Спестяването на това ограничение не е никак малко, имайки предвид факта, че в .Net няма множествено наследяване.

Свързването се реализира непосредствено след успешното завършване на компилирането, както на основния код, така и на аспектите. Свързващата функционалност се ръководи изцяло от конфигурацията на аспектите, описана в съответните XML файлове. Като вход на процеса се явяват:

- асемблитата на основния код (MSIL);
- асемблитата на аспектния код (MSIL);
- конфигурационните файлове, описващи свързването на аспектите и основните класове.

Самият процес на свързване се свежда до:

- първоначална обработка на всички асемблита, съдържащи аспекти;
- идентифициране на всички точки на свързване, отговарящи на конфигурационните файлове;
- дефиниране на краен брой трансформации върху основния код, които водят до аспектирането му;
- изпълняване на трансформациите;
- записване на резултатния код.

След извършване на процеса на свързване, като резултат се получава код, който включва както основните класове, така и приложените трансформации, свързани със самото аспектиране на кода.

4.4 Обобщение

В тази глава разгледахме случаите на употреба на Aspect.Net. Като резултат от анализ на текущите разработки в областта и случаите на употреба идентифицирахме функционалните и нефункционалните изисквания, които Aspect.Net трябва да удовлетворява. С оглед на изискванията, в края на главата разгледахме общия дизайн подход, който бе реализиран в Aspect.Net.

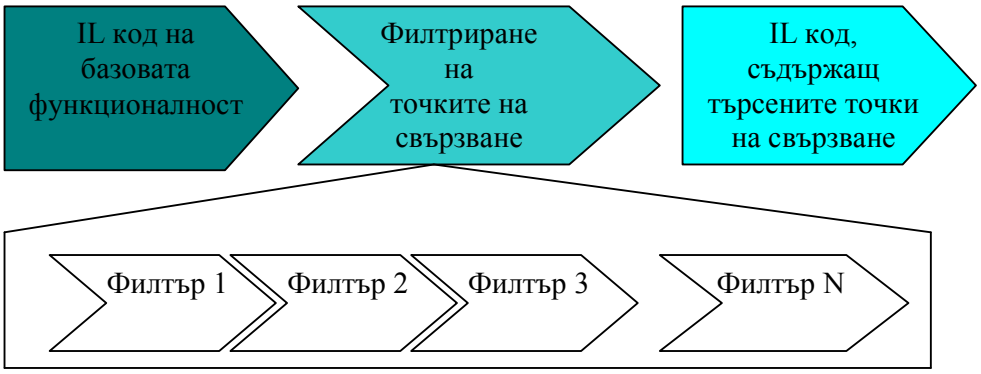
5 Реализация

В тази глава ще разгледаме по-подробно някои ключови части от реализацията на Aspect.Net, както и ще покажем къде и как са реализирани изискванията, изброени в глава 4.

5.1 Идентифициране на точките на свързване

Една от ключовите функционалности във всяка АОП разработка е идентифицирането на точките на свързване. По принцип, този процес се свежда до намирането на всички точки на свързване, които отговарят на дадена дефиниция на разрез (pointcut).

Фигура 14 илюстрира процеса:



Фигура 14 - Идентифициране точките на свързване

От фигурата се вижда, че самото идентифициране на точките на свързване е реализирано посредством верига филтри, всеки от които реализира конкретна логика, определяща дали даден елемент принадлежи на множеството от точки на свързване, които търсим.

На всеки филтър отговаря елемент от XML конфигурационния файл (наследник на абстрактния елемент predicate), в който са указани параметрите, нужни за нормална работа на съответния филтър.

В общия случай един .Net проект има повече от едно асембли, а самото аспектиране на асемблитата се извършва последователно. От друга страна, за всеки дефиниран в конфигурационния XML файл pointcut, ние търсим съответните точки на прекъсване чрез последователни извиквания към

съответния метод на филтъра. Именно поради тази причина, базовият интерфейс на филтрите съдържа метод (PreCacheAssembly), който бива извикван автоматично от Aspect.Net, за да може съответният филтър да кешира нужната му информация и повторното търсене в даденото асембли да става с много по-висока скорост.

В Aspect.Net има реализиран само един филтър – MethodMatcher. Той служи за идентифициране на методи, като точки на свързване. Параметрите за филтриране, които поддържа са:

- име на метода (позволява използване на '*');
- модификатор на достъп;
- тип на върнатия параметър;
- параметри на метода.

Имайки предвид, че основната тежест пада на търсенето по име, MethodMatcher филтърът реализира кеш, базиран на префиксно дърво. Изграждането на префиксното дърво се случва в тялото на метода PreCacheAssembly на филтъра, който (метод) се извиква само веднъж за всяко асембли, което е обект на търсене.

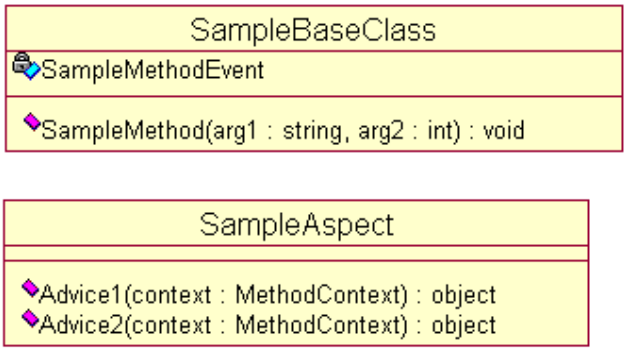
Класът, който енкапсулира по-голямата част от функционалността за идентификация на точките на свързване, а също е и входна точка на търсенето, се казва PointCutManager. За повече информация – виж Приложение 1.

Както вече видяхме, всички изисквания, свързани с дефинирането на pointcuts (т.е. идентификацията на точките на свързване), описани в глава 4.1, точка 8.2; са напълно реализирани.

5.2 Свързване на аспектите и базовата функционалност

След като точките на свързване са идентифицирани е време аспектният код да бъде интегриран в базовия. В следващите редове ще покажем как е реализирано това в действителност, така че всички изисквания, изброени в глава 4, да бъдат удовлетворени.

Нека имаме следните 2 класа, илюстрирани на фигура 15:



Фигура 15 – Пример, стъпка 1

Ще приемем, че първият е базов клас, а вторият – аспектен клас, съдържащ два примерни съвета – SampleAdvice() и SampleAdvice2(). Искаме да свържем SampleAdvice с SampleMethod, така че:

- SampleAdvice() да бъде извикан преди базовия метод – т.е. before свързване;
- SampleAdvice2() да бъде извикан вместо базовия, като той трябва да има възможност да стартира в произволен момент базовия метод – т.е. around свързване.

Изискванията, които трябва да удовлетворим при свързването са:

1. Съветите трябва да имат достъп до контекста на базовия метод. Този контекст ще съдържа:
    a. типа на свързването на съвета;
    b. стойностите на формалните параметри на базовия метод;
    c. делегат към базовия метод.
2. Свързванията before и after да се представят вътрешно чрез around свързване;
3. Поради семантиката на around свързванията, трябва да предоставим средство, с което разработчика на аспекти да може да извиква базовата функционалност във произволен момент;
4. Аспектите и базовата функционалност трябва да бъдат независими едни от други – т.е. трябва да е възможно даден съвет да може да бъде свързван around към множество методи и извикването на базовия метод не трябва да зависи от конкретното свързване;

5. Всичко по-горе трябва да бъде реализирано **без** да се добавя нищо ново към нито един от .Net езиците

Повечето от тези изисквания са описани в глава 4, като тези, които не са (основно изискване 1 и 3) се формираха по време на реализацията на проекта.

Първото изискване бе реализирано най-лесно чрез просто дефиниране на клас, съдържащ необходимата информация. Далеч по-интересно стои въпросът с останалите изисквания. Нека ги разгледаме по-подробно, както и проблемите свързани с тях.

Принципно има два подхода за самото свързване – чрез променяне на базовата функционалност, така че в съответните методи да бъдат 'инжектирани' извиквания към съответните аспекти; и втория подход – чрез създаване на прокси клас, който капсулира базовия и контролира кога се извиква базовият и кога аспектният код. Вторият начин е далеч по-проблемен от първия с оглед на факта, че той би изисквал всички референции към оригиналния клас да бъдат заменени с референции към новия прокси клас. Това от своя страна води до следните проблеми:

- целият базов код трябва да бъде сканиран и редактиран, за да бъдат опреснени референциите;
- подобно сканиране би увеличило драстично времето на свързването;
- няма никаква гаранция, че всички референции могат да бъдат намерени или могат да бъдат сменени:
    - в типичен фреймуърк, където с reflection се използва даден клас (първоначално създаден по име), референциите не биха могли да бъдат хванати, тъй като те се създават динамично по време на изпълнение на продукта;
    - в друг типичен случай, където имаме стандартна plug-in архитектура, референциите към даден клас от базовата функционалност, които се намират във вече написан от трета страна plug-in не могат да бъдат сменени без да бъдат нарушени авторските или лицензните права.

Поради всички тези причини, единственият валиден подход е първият – т.е. базовият клас да бъде модифициран по такъв начин, че от една страна да удовлетворява всички изисквания, дефинирани по-горе, а от друга да изглежда непроменен за външния свят – т.е. за ползващите и зависещите от него класове.

5.2.1 Подготовка на базовата функционалност

Навлизайки в подробностите на свързването, вече ни е ясно, че все пак някакви промени ще трябва да бъдат направени в базовия клас. Затова тук е мястото да дефинираме понятието „непроменен за външния свят". С оглед на ООП концепциите и факта, че базовата функционалност е реализирана с ООП средства, то външният свят се свежда до public и protected секциите на съответния базов клас. Те са единствените, които се виждат 'отвън' – т.е. от другите класове и от евентуалните наследници на класа. С други думи, за да запазим семантиката Aspect.Net гарантира, че:

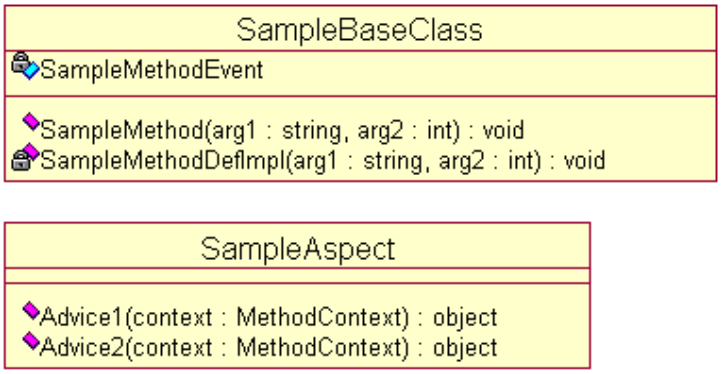
- повечето промени, както и всички нововъведения в класа ще бъдат реализирани в private секцията на съответния базов клас. Освен това те ще са дефинирани като static, тъй като свързването трябва да влияе върху всички инстанции на дадения клас;
- в public и private секциите единствено тялото на базовия метод, който бива свързван, е обект на промяна. Тази промяна не влияе на сигнатурата на метода, така че след промяната той ще продължава да изглежда по същия начин за всички, които го ползват статично или динамично.

След като изяснихме къде ще правим промени в базовия клас, то е време да поговорим за това и какви точно промени ще са нужни. Едно от изискванията гласи, че множество съвети трябва да могат да бъдат свързани към съответен базов метод. В .Net има реализиран механизъм за:

- регистриране на множество методи с еднаква сигнатура в дадена точка на извикване;
- лесно извикване на всички регистрирани методи.

Това е така наречените модел на събитията (event model). Точно този модел напълно задоволява нашите нужди от гледна точка на нужните промени. Ако за всеки един метод, който се явява точка на свързване, можем да дефинираме скрито (private) и статично (static) събитие, то всички свързани в тази точка съвети просто трябва да бъдат регистрирани, а в последствие извикани чрез възникването на това събитие. Взимайки под внимание, че ще трябва да променим базовия метод, който е обект на свързване, то той е и естественото място за възникване на събитието. Освен това, преди възникване на събитието, контекстният обект трябва да бъде създаден и инициализиран, така че да може

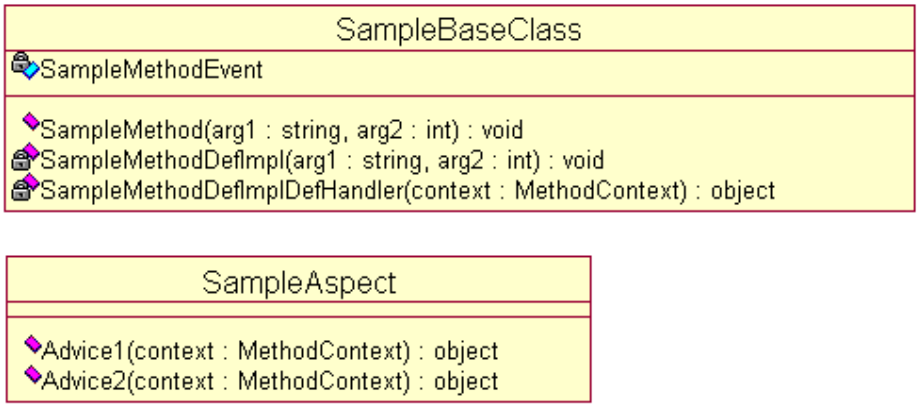
да бъде предаден като параметър при възникването на събитието. Не трябва да забравяме обаче, че оригинално базовият метод е имал съвсем друг изпълним код в тялото си. Затова преди да правим каквито и да са промени по него, ние трябва да го копираме като частен (private) метод на класа със същата сигнатура. Фигура 16 отразява промените до момента, по отношение на базовия клас:



Фигура 16 – Пример, стъпка 2

Копираният оригинален метод се казва SampleMethodDefImpl(), а събитието SampleMethodEvent. Както виждаме имената и на двете зависят от името на метода, който е обект на свързване. По този начин гарантираме уникалността на имената на новосъздадените обекти.

Друг private метод, който трябва да дефинираме в базовия клас има сигнатурата на съвет. Това е с цел да избегнем създаването на делегатен клас с конкретната сигнатура на оригиналния метод (SampleMethodDefImpl). Този private метод играе ролята на адаптер към интерфейса на оригиналния метод. Фигура 17 илюстрира новия метод (SampleMethodDefImplDefHandler):



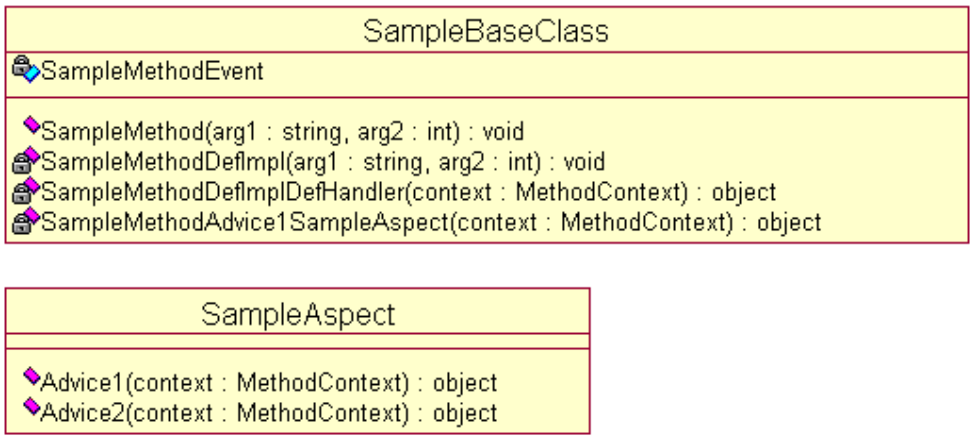
Фигура 17 - Пример, стъпка 3

Последният проблем, свързван с възникването на събитието, касае предаването на контекстния параметър за типа на свързване. Какъв по-точно е проблемът? Самият факт, че един и същи съвет може да бъде свързан с няколко различни базови метода, логично води до извода, че типът на свързване на конкретния съвет с конкретния метод трябва да бъде съхранен по някакъв начин в базовия клас, а не в аспектния. Първата възможна идея е това да бъде статично записано при конструирането на контекста в променения вече базов метод, където става възникването на събитието. Това, обаче, не е правилното място, тъй като може да имаме повече от един съвет свързан с метода, които са свързани по различен начин. Този факт прави невъзможно статичното записване на начина на свързване в базовия метод. Също знаем, че това трябва да се случи преди извикването на съвета, тъй като съветът очаква контекстният му обект да е правилно инициализиран, вкл. със стойността на типа на свързване. Решението на проблема, което Aspect.Net предлага е:

- за всеки свързан към метода съвет трябва да добавим нов private static метод към класа, който статично инициализира съответния параметър в контекстния обект и след това предава изпълнението към съвета;
- вместо съветите, тези методи ще са всъщност тези, които ще се регистрират за събитието, като по този начин ще позволим:
    - общ код за всички свързани съвети в базовия метод (попълване на контекст и възникване на събитието);

- различна инициализация на контекстния обект, в зависимост от вида свързване на съответния съвет и базовия метод.

Фигура 18 илюстрира промените до момента:



Фигура 18 - Пример, стъпка 4

Както се вижда, името на метода се формира от конкатенирането на следните низове:

- името на базовия метод;
- името на съвета;
- името на аспекта.

По този начин гарантираме уникалността му, имайки предвид неговата семантика, дефинираща релация 1:1 между даден съвет и базов метод.

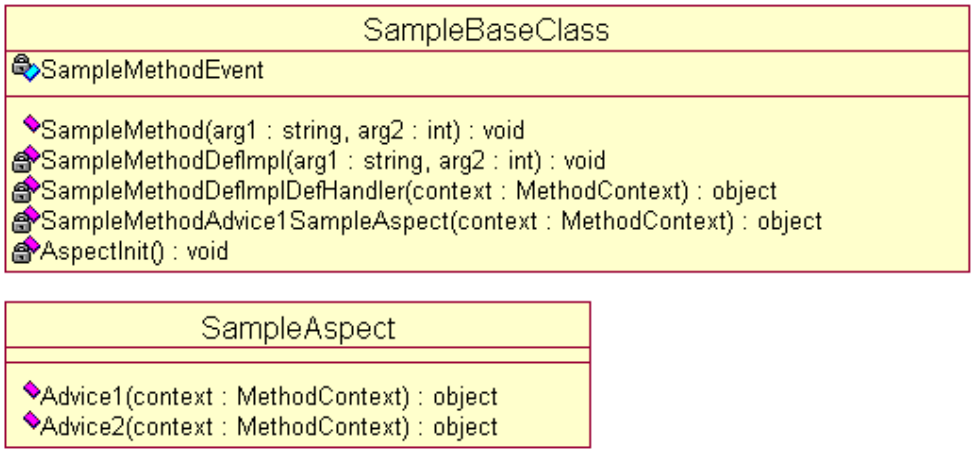
Вече покрихме всичко по отношение на нужните модификации за извикването на свързаните съвети, както и съдбата на оригиналния метод. Това, което не е все още ясно е как точно ще реализираме регистрирането на новосъздадените методи към събитието, дефинирано за всеки един метод, представляващ точка на свързване. Мястото, където това регистриране се случва, трябва да е общо за класа и в него да бъдат регистрирани всички методи, които са прокси към съответно свързаните съвети. За целта ще добавим нов private static метод (AspectInit) към класа, в който:

- за всеки един свързан (към метод на класа) съвет ще има код, който регистрира съответния прокси метод към съответно дефинираното събитие;

- за всички аспекти, имащи съвети, свързани към някой от методите на класа, ще бъде създадена и записана в предварително дефинирана статична променлива от типа на съответния аспект инстанция.

С въвеждането на AspectInit метода решихме проблема с регистрирането на прокси методите към съветите и централизирането на това регистриране. За да заработи подходът, обаче, е задължително този метод да бъде извикан преди който и да е от конструкторите на съответния клас. Това е нужно поради факта, че в съответните конструктори може да има извиквания към методите, които са свързани с аспекти. Това води до още една модификация, която Aspect.Net трябва да извърши – да обходи всички конструктори и да промени телата им по такъв начин, че първата инструкция в тях да бъде извикване на AspectInit метода.

Фигура 19 показва как изглежда диаграмата след последната промяна:



Фигура 19 - Пример, стъпка 5

Вече описахме всички по-важни промени в базовия код, както и причините и нуждите, които ги налагат. Затова нека обобщим какви са нужните промени по даден базов клас X с метод M, ако той бива свързван със съвет A на аспект T.

1. Дефинираме в X private static събитие AEvent;
2. Копираме оригиналния метод X.M в X.MDefImpl, който е private;
3. Създаваме private static метод X.MDefImplDefHandler, който е със сигнатурата на делегата на събитието и е прокси към съответния X.MDefImpl;

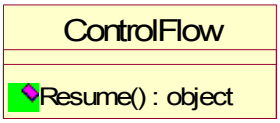
4. Модифицираме X.M по такъв начин, че новото му тяло извършва само:
    a. Създаване и инициализиране на контекстния обект;
    b. Възниква събитието AEvent.
5. Създаваме private static метод X.MAT, който статично попълва в контекстния обект (параметър на метода) вида на свързването между X.M и T.A;
6. Проверяваме дали има вече създаден private static AspectInit метод и ако няма – създаваме такъв с празно тяло и модифицираме всички конструктори в класа да го извикват с първата си инструкция;
7. Модифицираме AspectInit метода, като регистрираме X.MAT към събитието AEvent. Ако това е първото свързване на съвет от аспекта T, добавяме и код за създаване на инстанция на T, която съхраняваме в предварително дефинирана статична променлива на класа. По този начин една инстанция на T обслужва всички инстанции на X.

5.2.2 Подготовка на аспектите

След като разгледахме какво е нужно като промяна по отношение на базовата функционалност, нека видим дали и какво е нужно да се промени и по отношение на съветите и аспектните класове.

Както вече споменахме различните свързвания (before, after и around) трябва вътрешно да се представят като around свързване, за да е коректно самото свързване. От друга гледна точка, програмистите, реализиращи аспекти и съвети, трябва да имат средство, с което да могат в произволен момент да извикат базовия метод, в случай на around свързване. Преди да решим как ще подходим, трябва да вземем под внимание и още едно ограничение, което трябва да спазим – не трябва да добавяме нови синтактични конструкции в никой .Net език, така че въвеждане на нова ключова дума (например resume) не е възможно, а използването на съществуваща би довело до редица двусмислици и проблеми.

Проблемът с извикването на базовия (оригиналния) метод ще решим като създадем клас ControlFlow с метод Resume, който реално няма реализирано тяло. Самият клас изглежда така:



Когато програмистът на даден съвет, който ще бъде свързан around, иска да извика базовия метод, той просто поставя извикване към ControlFlow.Resume(). Вече е ясно, че без допълнителни промени на аспектния код, извикването на ControlFlow.Resume() не би довело до нищо, имайки предвид че самият метод Resume няма реализация.

Промените, които трябва да се направят по отношение на проблема по-горе са относително тривиални. Свеждат се до заместването на извикването към ControlFlow.Resume() с динамично извикване на подадения делегат в контекстния обект с параметрите, които също могат да бъдат намерени в контекстния обект. По този начин за програмистите на аспекти всичко това остава скрито, оставяйки им по-голяма възможност да се концентрират върху самата аспектна функционалност.

Другият проблем, който изисква промени по самите аспекти е, че before и after свързванията, трябва да се представят чрез around. От гледна точка на реализацията това се свежда до изискването:

- без значение от свързването, винаги трябва да се извиква съветът;
- в зависимост от свързването (before/after), съвета извиква базовия метод когато е нужно (след/преди изпълнението на самия съвет).

Друга важна характеристика, която трябва да бъде отчетена е, че един съвет може да бъде свързан с множество методи, като начинът на свързване е произволен. Това означава, че каквито и промени да правим на даден съвет, с оглед на горните изисквания, те не трябва да включват статична информация за конкретно свързване.

С оглед на всичко казано, решението, което Aspect.Net предлага е следното:

1. Информацията за типа на свързване не се добавя като статични данни към самия аспект/съвет, а ще се съхранява в базовия метод, т.нар. точка на свързване (виж 5.2.2 за повече информация);
2. Типа на свързване ще се подава към съвета в контекстния обект;
3. Тялото на всеки съвет се променя по такъв начин, че в началото и края на метода на съвета се слагат следните два блока:

a. *в началото на съвета* : ако параметърът за типа на свързване (в контекстния обект) е after, аналогично на заместването на ControlFlow.Resume() се извиква базовият метод;
b. *във всяка изходна точка на съвета* : ако параметърът за типа на свързване (в контекстния обект) е before, аналогично на заместването на ControlFlow.Resume() се извиква базовият метод.

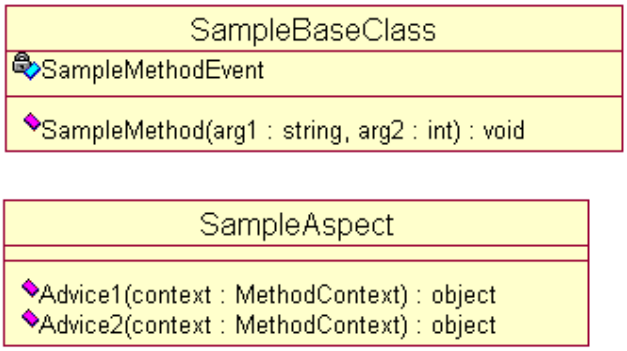
Нека сега обобщим всички промени, които се извършват спрямо аспектния код:

1. В началото и всеки край, т.е. всяка изходна точка, на всеки съвет (на всеки аспект) се слагат гореспоменатите 2 блока, които свеждат свързване от тип before или after до свързване от тип around;
2. Тялото на всеки съвет се претърсва за извиквания към ControlFlow.Resume(). Кодът на всяко такова срещане се подменя с код, извикващ оригиналния метод с параметрите, съхранени в контекстния обект.

5.2.3 Процеса на свързване

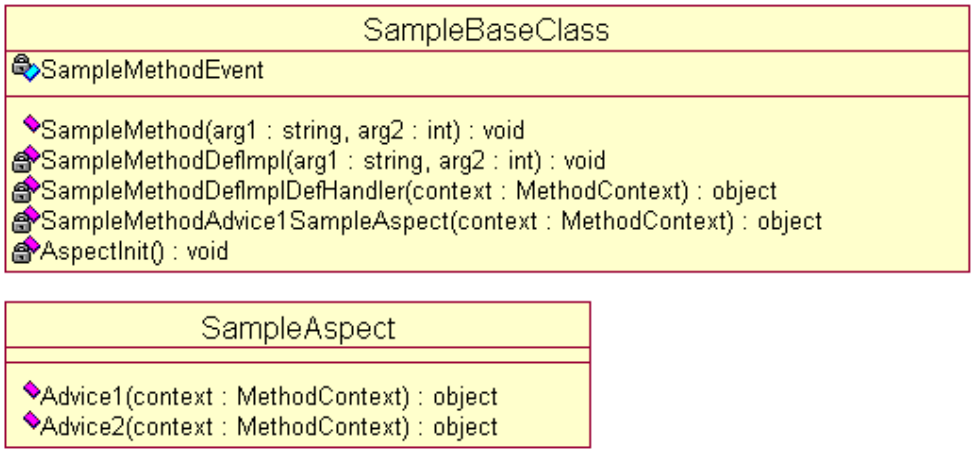
В предните две секции разгледахме подробности в какво се състои подготовката на аспектната и базовата функционалност, нужна за реализиране на изискванията, описани в глава 4. Въпреки, че нужните промени бяха обяснени в достатъчно ниво на детайлност, в тази секция ще предоставим визуална представа (посредством диаграми) за това как протича изпълнението на оригиналния и на свързания метод. Всичко това ще спомогне за допълване на цялостното обяснение и даване на нагледна представа за начина на работа на свързаната вече функционалност.

Нека се върнем на примера от началото на тази глава. Фигура 20 илюстрира как изглеждаха класовете преди, каквато и да е намеса от Aspect.Net:



Фигура 20 - Пример, стъпка 1

След свързването на съвета Advice1 с метода SampleMethod, класовете изглеждат по следния начин:

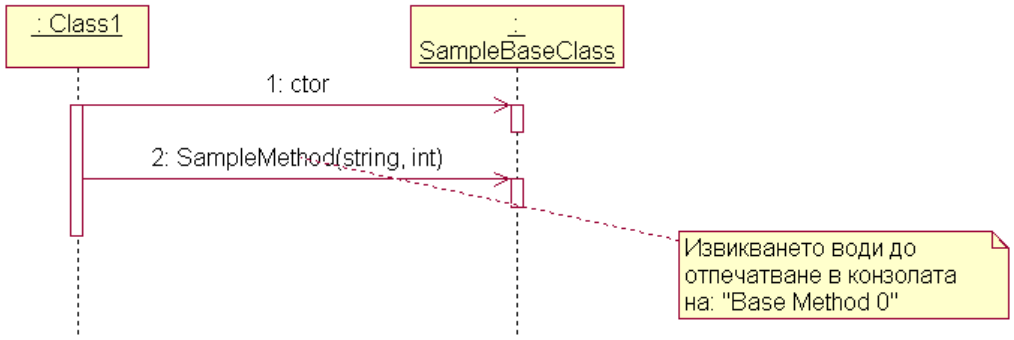


Фигура 21 - Пример, стъпка 5

Нека приемем, че свързването е от тип around. Нека класът Class1 има референция към SampleBaseClass и ползва метода SampleMethod. Нека логиката на метода SampleMethod е да изписва в конзолата низ, който се получава от конкатенацията на 2та аргумента.

Логиката на Advice1 е да промени първия аргумент, като добавя като префикс низа „Advice1->”; и втория, добавяйки единица към параметъра. След това предава изпълнението на базовия метод, посредством ControlFlow.Resume()

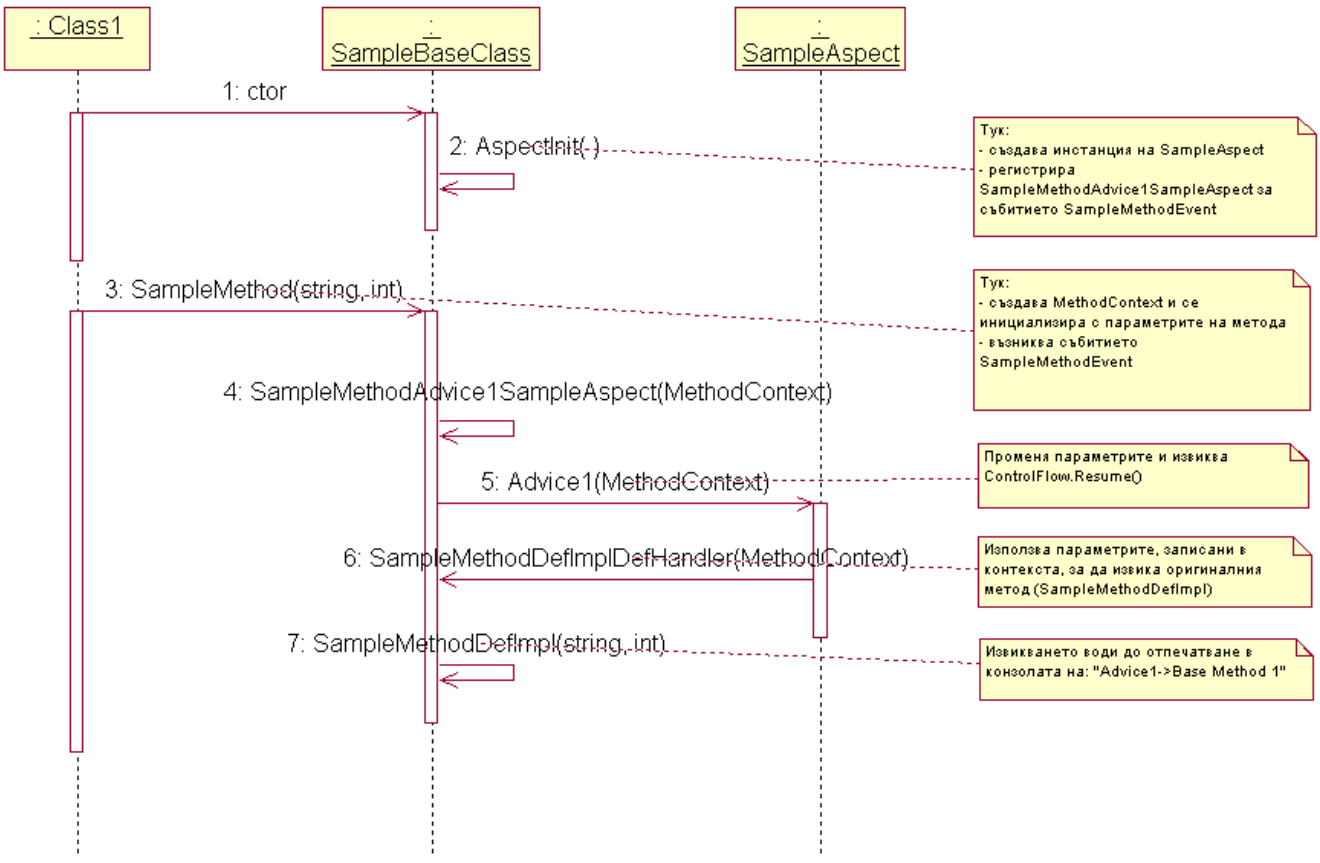
Фигура 22 по-долу илюстрира изпълнението на извикването на метода SampleBaseClass.SampleMethod() с параметри „Base Method” и 0:



Фигура 22 - Пример, оригинална диаграма на извикването

Както се вижда, изпълнението е изключително просто – създава се инстанция на SampleBaseClass, изпълнява се конструкторът, а последващото извикване на SampleMethod предизвиква отпечатване в конзолата на „Base Method 0".

Ето каква е последователността на изпълнение при свързване от тип around на Advice1 към SampleMethod:



Фигура 23 - Пример, диаграма на извикването след свързването

Детайли по отношение на различните стъпки от изпълнението, описани на фигура 23, могат да бъдат намерени в кутийките в дясно. Резултатът от цялото изпълнение е отпечатването на следния низ в конзолата: „Advice->Base Method 1”, което всъщност беше и очаквания резултат.

5.3 Обобщение

В тази глава разгледахме ключовите елементи от реализацията на Aspect.Net, а именно: идентифицирането на точките на свързване, подготовката на базовата и аспектната функционалност, както и самия процес на свързване. Описаме и мотивацията ни за избора на конкретните реализационни решения. Множеството примери и диаграми имаха за цел да дадат на читателя още по-ясна представа за това какво, защо и как се случва по време на работа на Aspect.Net, както и по време на изпълнение на вече свързания код. Показахме и как изискванията в глава 4 са удовлетворени и какво влияние оказват на самата реализация на Aspect.Net.

6 Оценка на решението

В тази глава ще разгледаме Microsoft .Net като платформа за Аспектно Ориентирана Разработка на Софтуер. В последствие ще опишем методите ни за оценка и ще ги използваме, за да оценим резултата на дипломната работа – Aspect.Net. Накрая ще предоставим един пример за междуезиково свързване, посредством Aspect.Net.

6.1 Microsoft .Net като платформа за AOPC

"Платформа е всяка база от технологии, върху която други технологии или процеси са построени. По отношение на компютрите, платформа е основна компютърна система, на която могат да се изпълняват приложни програми". AOCP е нова технология, позволяваща разделянето на отношенията при производството на софтуер. Въпросът, на който търсим отговор е: позволява ли .Net платформата AOCP методи или техники? Аз вярвам, че .Net позволява AOCP, най-малкото поради факта, че е възможно съществуването на Aspect.Net, който позволява модуляризиране на пресичащи се отношения, без да се налагат разширения на който и да е от .Net езиците. Платформата .Net предлага повече средства за AOCP от други подобни платформи, особено по отношение на следните области: Мета-данни, Рефлекция (Reflection), Генериране на код и Междинен език (MSIL).

6.1.1 Мета-данни

Мета-данните представляват информация за данните, сорс-кода на дадена програма, наличните типове, асемблита и т.н., която се съхранява в програмата. В .Net метаданни могат да бъдат дефинирани чрез използване на атрибути. Атрибутите могат да бъдат вградени ([Webmethod], [Serializable] и др.) и потребителски. Потребителските се създават от програмиста – в Aspect.Net използваме [AspectAttribute] и [AdviceAttribute], за да придадем семантика на аспекти или съвети към съответните класове/методи. Метаданните и в частност потребителските атрибути, са огромно предимство, което .Net предоставя на програмистите.

6.1.2 Рефлекция (Reflection)

Рефлекцията е тясно свързано с мета данните. Тя представлява процеса, по който програмата може да прочете собствените си мета-данни. В Java механизмът на рефлекция позволява да се виждат данни, да се откриват типове, динамично да се изпълняват характеристики и методи и т.н. В .Net поддръжката на рефлекция е далеч по-развита – реализирана е основната цел на рефлекцията, а именно – генерирането на типове по време на изпълнение, познато още под името рефлекционна емисия. Огромното множество налични АОП разработки имат своите корени в базирани на рефлекцията подходи за реализация.

6.1.3 Генериране на код (CodeDOM)

Възможността да се моделира сорс-код като езиково-неутрално абстрактно дърво (ЕНАД) е от огромна полза за голяма част от разработките в областта. Някои такива разработки, които реализират свързване на нивото на програмния код са AspectJ, AspectC# и др. Възможността да се моделира кодът, използвайки ЕНАД, би позволила междуезиково свързване на базова и аспектна функционалност. Ако даден език може да бъде моделиран като CodeDOM граф, то междуезиково свързване би било относително лесно за реализация, тъй като целият код би бил достъпен в графа. За съжаление обаче, текущата версия на CodeDOM не поддържа напълно Общата Езикова Спецификация (Common Language Specification) и междуезиковото свързване не е възможно за момента, използвайки CodeDOM.

6.1.4 Междинен език (MSIL)

Междинният език (МЕ) на Майкрософт се използва най-често при АОП разработките за .Net. Самият факт, че C#, VB.Net и другите .Net езици се свеждат до МЕ след компилация, позволява междуезиково свързване. Точно това е и подходът, избран и реализиран от Aspect.Net. JIT компилаторът на .Net компилира МЕ преди зареждане и изпълнение на МЕ кода. Това способства лесната интеграция и реализиране на динамично аспектиране на функционалността.

6.2 Оценка на решението Aspect.Net

За да оценим Aspect.Net, ще го сравним с други подобни инструменти – AspectJ и AspectC#. Методите за оценка на решението са следните:

1. Начин на дефиниране на аспектите,
2. Средства за композиция,
3. Начин на реализация,
4. Обособеност на аспектите и базовата функционалност,
5. Софтуерен процес,
6. Ефективност на употребата (usability).

Преди да продължим нататък трябва да отбележим, че разглежданите разработки са имали значително по-голям период на разработка, в който са взели участие голям брой програмисти.

6.2.1 Начин на дефиниране на аспектите

Тази оценка е свързана с:

1. Как се дефинират точките на свързване?
2. Към какво могат да бъдат прилагани аспектите? Могат ли да бъдат прилагани на ниво променливи, методи или класове?
3. Степента, до която аспектите могат да бъдат адаптирани към конкретна употреба в системата. Могат ли да бъдат общи или специфични аспектите?

Инструмент	**Начин на дефиниране на аспектите**
AspectJ	AspectJ притежава модел на точките на свързване, при който: - Точка на свързване е добре дефинирана, изпълняема част от програмата (Приложение 3); - Pointcut е множество от точки на свързване. Аспект е единица, модуляризираща пресичащо отношение. Аспектите могат да бъдат, както специфични, така и общи (и съответно преизползваеми), свързани към различни области ог системата Точките на свързване се дефинират посредством специфичен синтаксис, близък до този на езика Java.

AspectC#	Точка на изпълнение в AspectC# може да бъде единствено изпълнението на метод. Аспект е единица, модуляризираща пресичащо се отношение. Аспектите могат да бъдат, както специфични, така и общи (и съответно преизползваеми), свързвани към различни области от системата. Точките на свързване се дефинират посредством XML файл.
Aspect.Net	Точка на изпълнение в Aspect.Net може да бъде единствено изпълнението на метод. Аспект е единица, модуляризираща пресичащо се отношение. Аспектите могат да бъдат, както специфични, така и общи (и съответно преизползваеми), свързвани към различни области от системата. Точките на свързване се дефинират посредством XML файл.

Таблица 3 - Начин на дефиниране на аспектите

6.2.2 Средства за композиция

Тази оценка е свързана с:

1. Има ли доминираща декомпозиция, която обхваща целия език? Дали има един вид декомпозиция, който се прилага само върху аспектите или всички отношения се третират еднакво?
2. Съдържа ли системата разширения на езика, нужни при дефинирането на аспекти?
3. Каква е връзкава и има ли зависимост между различните аспекти? За да могат да се използват многократно, подобна връзка трябва да се разгледа в детайли.

Инструмент	**Средства за композиция**
AspectJ	Доминантната декомпозиция в Java е ООП, а по отношение на AspectJ – всички аспекти се прилагат към структурата на класовете. AspectJ добавя към езика Java синтактични разширения. Аспектите в AspectJ могат да влияят на базовата

	функционалност чрез използването на ключовата дума proceed. AspectJ предоставя правила за разрешаване на конфликти между аспектите.
AspectC#	AspectC# не въвежда разширение към езика C#. Доминиращата декомпозиция е ООП и всички аспекти се прилагат към структурата на класовете. AspectC# има сходни конструкции като AspectJ и може динамично да влияе на изпълнението на базовата функционалност посредством ключовата дума proceed.
Aspect.Net	Aspect.Net не въвежда разширение към никой език, тъй като е езиково независим. Доминиращата декомпозиция е ООП, която се прилага, както към базовата, така и към аспектната функционалност. Aspect.Net може да влияе на базовата функционалност, посредством празния метод ControlFlow.Resume, който бива инструментиран по време на компилация.

Таблица 4 – Средства за композиция

6.2.3 Начин на реализация

Оценката на реализацията включва следните критерии:

1. Какъв тип е свързването на базовата и аспектната функционалност – динамичен или статичен?
2. Има ли ясно разграничение между аспектите и базовата функционалност? Може ли компилирането на двете да става по отделно?
3. Какъв е входът за свързващата функционалност? Ако е сорс код, програмистът трябва да има достъп до кода на програмата. Ако пък е междинен език, той има нужда само от готовия файл или асембли.
4. Може ли системата да бъде приложена към съществуващи приложения?
5. Има ли начин за верифициране на свързването? Възможно ли е да преценим дали свързването на даден аспект към базова функционалност е правилно?
6. Системата поддържа ли аспектен полиморфизъм?

Инструмент	Начин на реализация
AspectJ	В AspectJ свързването е реализирано статично, т.е. по време на компилация. Има ясно разграничение между базова и аспектна функционалност и е възможно да се компилира и изпълнява самостоятелно базовата функционалност. AspectJ може да бъде приложен, както върху съществуващи, така и върху нови приложения. AspectJ използва сорс код като вход за свързването, а верифицирането на самото свързване се реализира от компилатора. AspectJ не поддръжа на аспектен полиморфизъм.
AspectC#	Свързването в AspectC# е статично. Има ясно разграничение между аспектна и базова функционалност. AspectC# може да бъде прилаган, както върху съществуващи, така и върху нови приложения. AspectC# използва сорс код като вход за свързването. Липсва поддръжка на верифициране на свързването, както и поддръжка на аспектен полиморфизъм.
Aspect.Net	Свързването в Aspect.Net е статично. Разграничението между аспектната и базовата функционалност е чисто декларативно и може да бъде пренебрегнато. Като резултат, аспектите също могат да играят ролята на базова функционалност за други аспекти. Aspect.Net може да бъде прилаган, както върху съществуващи, така и върху нови приложения. Aspect.Net използва ME като вход за свързването, което позволява реализирането на междуезиково свързване. Липсва поддръжка на верифициране на свързването, както и на аспектен полиморфизъм.

Таблица 5 - Начин на реализация

6.2.4 Обособеност на аспектите и базовата функционалност

Обособеността на аспектите и базовата функционалност се определя от това дали програмистът на базовата функционалност трябва да е наясно с факта, че към неговия код ще бъдат свързвани аспекти, т.е. дали той ще трябва да извърши някакви приготовления, за да може свързването на аспектите да не бъде възпрепятствано. Това се нарича неосъзнатост (oblivousness) ***[1]***. Като

следствие на въпроса – под каква форма ще е възможно прилагането на аспектите към системата – локално, глобално или и двете? Най-добре би било ако е възможно това да зависи от преценката на програмиста.

Инструмент	Обособеност на аспектите и базовата функционалност
AspectJ	Степента на обособеност е оставена на програмиста. Възможно е да се наложи програмистът да приготви базовата функционалност за някои аспекти, както например в случая с изпълнението в транзакция. Аспектите могат да влияят на програмата локално и глобално.
AspectC#	Степента на обособеност е оставена на програмиста. В определени случаи базовата функционалност може да се наложи да бъде подготвена за някой аспект. Аспектите могат да влияят на програмата локално и глобално.
Aspect.Net	Степента на обособеност е оставена на програмиста. В определени случаи базовата функционалност може да се наложи да бъде подготвена за някой аспект. Аспектите могат да влияят на програмата локално и глобално.

Таблица 6 – Обособеност на аспектите и базовата функционалност

6.2.5 Софтуерен процес

Оценката по отношение на софтуерния процес включва:

1. Каква методология или фреймуърк се използват от системата?
2. Какви аспектни механизми способстват преизползването?
3. Възможно ли е да се анализира ефективността на дадена аспектна система?
4. Има ли механизъм за проследяване на грешки (debugging mechanism)?

Инструмент	Софтуерен процес
AspectJ	AspectJ използва АОП с ясно разделение между аспектна и базова функционалност. Аспект в AspectJ е модуляризирано пресичащо отношение – тази структура позволява преизползване. В момента няма механизъм за проследяване на

	грешки.
AspectC#	AspectJ използва АОП с ясно разделение между аспектна и базова функционалност. Аспект в AspectJ е модуляризирано пресичащо отношение – тази структура позволява преизползване. В момента няма механизъм за проследяване на грешки.
Aspect.Net	AspectJ използва АОП с ясно разделение между аспектна и базова функционалност. Аспект в AspectJ е модуляризирано пресичащо отношение – тази структура позволява преизползване. В момента няма механизъм за проследяване на грешки.

Таблица 7 – Софтуерен процес

6.2.6 Ефективност на употребата (usability)

Използваемостта и ефективността на употребата зависят пряко от това до колко лесна е тя за научаване и реално ползване. Има ли общество на потребителите, което да споделя информация, предложения за подобрения и прочие за системата.

Инструмент	Ефективност на употребата
AspectJ	AspectJ е лесен за употреба с множество допълнения към различни среди за разработка на Java. В момента има много разработки в областта на програмирането с AspectJ, както и съществуват няколко големи общности на потребителите му.
AspectC#	Въпреки надеждите за интерес към разработката поради близкия до AspectJ синтаксис, така и не се появиха заинтересувани групи, които да ползват инструмента.
Aspect.Net	Въпреки надеждите ми, че поради интеграцията на инструмента с Microsoft Visual Studio 2003, ще се появят заинтересувани групи от програмисти, това според мен е невъзможно в момента. Имайки предвид текущото състояние на другите инструменти (AspectJ например), доста подобрения трябва да се направят преди Aspect.Net да стане равностоен по

	функционалност и популярност на другите разработки (виж глава 7).

Таблица 8 – Ефективност на употребата

6.3 Обобщение

След реализацията на Aspect.Net имахме преките наблюдения и описахме потенциала на платформата .Net за АОСР. Също дефинирахме критериите за оценка на Aspect.Net:

1. Начин на дефиниране на аспектите
2. Средства за композиция
3. Начин на реализация
4. Обособеност на аспектите и базовата функционалност
5. Софтуерен процес
6. Ефективност на употребата (usability)

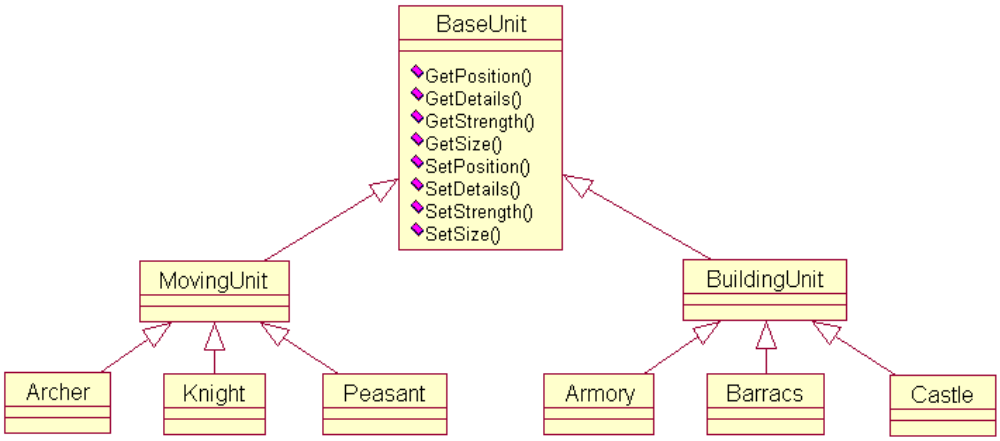
В останалата част от тази глава реализирахме оценката на Aspect.Net спрямо другите .Net АОП разработки по отношение на гореизброените критерии. Направените изводи могат да бъдат намерени в глава 8, където в допълнение ще преразгледаме поставените цели и ограниченията на решението. Но нека преди това разгледаме един реален пример на употреба на Aspect.Net, който ще ни подготви напълно за заключителната част на дипломната работа.

7 Пример на употреба на Aspect.Net

В тази глава ще демонстрираме един проблем, който Aspect.Net решава по изключително елегантен начин от дизайна до реализацията на решението. Примерът не е от множеството на стандартните примери, свързвани с АОП, а съм се постарал да е близък до реален проблем от ежедневната практика на разработчиците на софтуер.

7.1 Дефиниция на проблема

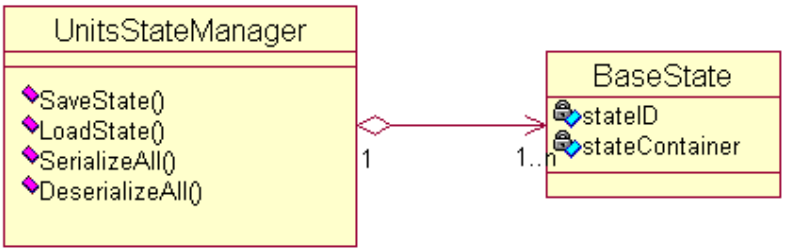
Нека предположим, че в момента работим по дадена компютърна игра, която е вече почти готова по отношение на реализацията. Месец преди официалното откриване на играта се получава изискване от продуцента, играта да предоставя възможност на играчите да записват филмчета, докато играят. Нека имаме следната йерархия от класове, които представляват малка и значително опростена част от целия обектен модел на играта:



Фигура 24 - Примерна йерархия от обекти

Искам да подчертая, че въпреки целта ми примерът да е максимално реален, обикновено в една игра обектният модел е много по-различен и далеч по усложнен поради редица фактори. С оглед на това горният модел е изключително опростен и хипотетичен, но все пак изпълнява целта, а именно – да изгради представа за начина на употреба на Aspect.Net в една относително реална ситуация съдържаща съвсем конкретен проблем.

За да решим проблема със записването на игра с цел преглеждане в последствие, ние въвеждаме нов клас UnitsStateManager, чиято отговорност е да съхранява и сериализира/десериализира BaseState обекти . Фигура 25 илюстрира тези два класа и връзката между тях:



Фигура 25 - Съхранение на състоянието

В следващите две секции ще разгледаме два подхода за довършването на решението на поставения проблем, а именно подхода на обособяване на отношението за сериализиране на гореспоменатата йерархия, използвайки тези два класа.

7.2 Стандартно решение

Стандартното решение е подчинено на ООП техниката. Дори само в тази категория има голям брой възможни подходи за дизайн на решението. Като цяло обаче, всички се свеждат до:

- Въвеждане на зависимост в класовете от йерархията към класа за съхранение на състоянията (StateManager) и базовия клас за състояние BaseState
- Промяна на всички методи, които влияят на състоянието на обектите, така че да са възможни:
    - Записване на мо;ментното състояние в нова инстанция на BaseState обект
    - Извикване на метода StateManager.SaveState(), подавайки съхраненото състояние.

Принципно е възможно да не се променят методите на класовете от йерархията, но в такъв случай интеграцията просто ще се премести на друго място – в трети клас, който ще трябва да следи кога се променят обектите от

йерархията и в съответствие с това да се погрижи за правилното сериализиране на състоянието на обектите. За съжаление този подход е значително по-сложен, а освен това и не винаги напълно възможен. С оглед на това, тук ще демонстрираме систематизирания по-горе подход, като по-прост и по-често срещан.

Както вече казахме, всички методи променящи състоянието на обектите от йерархията класове, изобразена във Фигура 24, трябва да бъдат модифицирани. Тъй като промените ще са идентични навсякъде, ще разгледаме само метода Knight.SetPosition(). Листинг 1 показва кода на метода в оригиналния му вид:

```
/// <summary>
/// Changes the position of a Knight object
/// </summary>
public void SetPosition( Position newPosition )
{
	// make sure we are given valid position
	if ( !this._currentPosition.IsValidInMap( this._currentMap ) )
	{
		throw new ApplicationException( "InvalidPosition set " +
					this._currentPosition.ToString() );
	}

	// update state
	this._currentPosition = newPosition;

	// raise the event, so that the engine redraws the object
	// in next render cycle
	NeedRefresh( this );
}
```

Листинг 1 – Оригинално тяло на метода

След промените, нужни за интегриране на сериализиращата функционалност, кодът изглежда по следния начин:

```
/// <summary>
/// Changes the position of a Knight object
/// </summary>
public void SetPosition( Position newPosition )
{
	try
	{
		// make sure we are given valid position
		if ( !_currentPosition.IsValidInMap( this._currentMap ) )
		{
			throw new Exception( "InvalidPosition set " +
					this._currentPosition.ToString() );
		}

		// create and initialize the state object
		BaseState currState = new BaseState();

		currState.ID = this.ID;
		Hashtable sc = currState.StateContainer;

		sc.Add( "position", this._currentPosition );
		sc.Add( "size", this._size );
		sc.Add( "strength", this._strength );

		// now save the state for serialization
		UnitsStateManager.Instance.SaveState( currState );

		// update state
		this._currentPosition = newPosition;

		// raise the event, so that the engine redraws the object
		// in next render cycle
		NeedRefresh( this );
	}
	catch
	{
		// in case of exception we should remove the
		// persisted state if we succeeded storing it already
		UnitsStateManager.Instance.DeleteState( this.ID );

		// rethrow the error
```

```
            throw;
        }
}
```

Листинг 2 – Тяло на метода след интегриране на сериализиращата функционалност

Допълнителният код в листинг 2 извършва следното:

- създава и инициализира обект, съдържащ състоянието на текущата инстанция на Knight класа;
- извиква метода UnitsStateManager.SaveState, който съхранява подадения обект за последваща сериализация;
- капсулира целия код в try-catch блок, така че ако възникне изключение, евентуално записаното състояние да бъде премахнато от UnitsStateManager.

7.3 Aspect.Net решение

Използвайки АОП подход, ние можем да енкапсулираме функционалността, която в предната секция ‘вградихме’ в оригиналния метод. Ще създадем аспект, който реализира и енкапсулира нужната функционалност. Базовата функционалност остава абсолютно непроменена – т.е. същата като показаната в листинг 1. Аспектът, който създаваме има следната реализация:

```
[Aspect( Name = "StatePersisterAspect" ) ]
public class StatePersisterAspect
{
        /// <summary>
        /// Persists old state of Knight objects
        /// </summary>
        [Advice( Name = "SaveStateKnight" )]
        public object SaveStateKnight(object context)
        {
                Knight unit =
                        ((MethodContext)context).ctxObject as Knight;

                // create and initialize the state object
                BaseState currState = new BaseState();

                currState.ID = unit.ID;
```

```
            Hashtable sc = currState.StateContainer;

            sc.Add( "position", unit.GetPosition() );
            sc.Add( "size", unit.GetSize() );
            sc.Add( "strength", unit.GetStrength() );

            try
            {
                  // perform standard base coding
                  ControlFlow.Resume();
                  // store the old state now
                  UnitsStateManager.Instance.SaveState( currState );
            }
            catch
            {
                  // remove the eventually stored old state
                  UnitsStateManager.Instance.DeleteState( unit.ID );
                  // rethrow the error
                  throw;
            }

            // we have to return something
            return null;
      }

}
```

Листинг 3 – Реализация на аспекта, енкапсулиращ пресичащото се отношение

Както се вижда в листинг 3, създали сме съвет SaveStateKnight(), който е специфичен съвет, свързван само към обекти от тип Knight. Съветът съдържа само и единствено функционалност, свързана със сериализирането на състоянието обекти от тип Knight. Последното ни позволява да се възползваме от знанието ни за типа на обекта, записан в контекста, така че можем да се възползваме напълно от интерфейса му.

За решаването на останалата част от задачата, би трябвало да добавим по един съвет и за всеки тип, изискващ специфично поведение по време на сериализация. Възможно е да реализираме съвет, който е общ и може да бъде свързван към широк набор обекти от йерархията.

XML конфигурацията, с която дефинираме свързването на аспектната и базовата функционалност изглежда така:

```
<?xml version="1.0" encoding="utf-8" ?>

<configuration>
      <pointcut name="KnightSetters">
            <combine>
                  <matchMethod>
                        <access_modifier value="public" />
                        <class value="Knight" />
                        <name value="Set*" />
                  </matchMethod>
            </combine>
      </pointcut>

      <weave
            pointcut="KnightSetters"
            aspect="StatePersisterAspect"
            advice="SaveStateKnight"
            bindOptions="bindAround" />

</configuration>
```

Листинг 4 – XML конфигурацията, нужна за свързването на аспекта

7.4 Резултати

Разликата между двата подхода е очевидна. При стандартния подход, имаме следните (стандартни) проблеми:

1. ‘Разхвърляност’ на кода - нужните редове код за създаване на BaseState и самото извикване на сериализиращия метод на UnitsStateManager са пръснати из всички Set методи на всички класове от йерархията.
2. ‘Оплетеност’ на кода – всеки Set метод и в частност разгледания в Листинг 2, след интегриране на новата функционалност съдържа код, който касае повече от едно отношение. В случая с показания в Листинг 2, отношенията са:
    - Валидация на позицията спрямо картата;

- Сериализация на състоянието – инстанциране на BaseState обект, извикване на `UnitsStateManager.SaveState` и обработка на изключения;
- Промяна на състоянието на обекта от тип Knight;

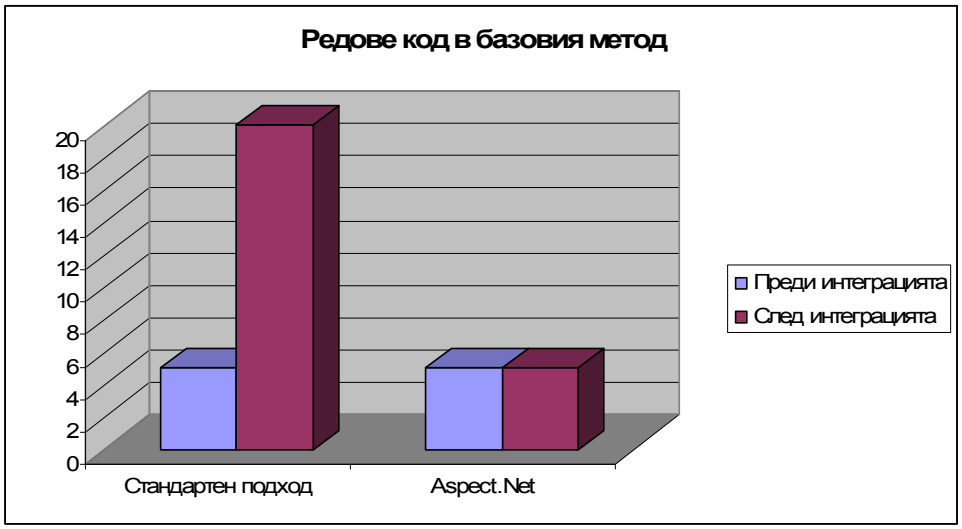
3. Повторение на код – като резултат от проблем 1, имаме големи конструкции от код, които се повтарят на множество места.
4. Увеличение на сложността - добавянето на новата функционалност (свързана със второто отношение) измени и увеличи многократно сложността на иначе простия оригинален метод. Това влияе негативно на времето за поддръжка и качеството на самия код.

За съжаление ООП не предлага механизъм, по който адресираните по-горе проблеми да бъдат решени напълно.

За разлика от ООП, Aspect.Net предоставя изразни средства и абстракции от света на АОП. Чрез дефинирането и обособяването на аспекта StatePersisterAspect, ние постигаме следните подобрения:

- модуляризацията премахва причините за проблем 1 – т.е. нямаме почти един и същ код на множество места;
- като следствие се разрешава и проблем 3;
- изнасянето на кода, реализиращ отношението ‘Сериализация на състоянието’, в отделен модул, намалява ‘оплетеността’ на кода, елиминирайки проблем номер 2;
- като следствие постигаме и драстично намаление на сложността, което само по себе си премахва причините за проблем номер 4.

За да получим по-формална представа за размера на подобренията, нека разгледаме следните две диаграми:



Фигура 26 - Редове код в базовия метод SetPosition()

Фигура 26 илюстрира графично промените по базовия метод, по отношение на броя редове код, нужни за интегрирането на новото отношение. Както се вижда, при стандартния подход имаме значително увеличаване на тялото на метода, докато при Aspect.Net броят на редовете остава същия. Имайки предвид, че новосъздадения съвет може да се приложи върху всички методи на класа Knight, количеството новонаписан код, може да бъде намален драстично, само и единствено поради употребата на Aspect.Net.

Фигура 27 - Процентно съотношение на отношенията в базовия метод SetPosition() след интеграцията

Фигура 27 показва съотношението на кода, реализиращ новото отношение (сериализация) и старото отношение - смяна на позицията на обекта. Както се вижда от диаграмата, следствие интеграцията се получава така, че кодът реализиращ новото отношение е значително повече от оригиналния код в тялото на метода. Използвайки Aspect.Net, новото отношение е обособено в изцяло нов модул (аспект), така че базовият метод остава непокътнат, без да бъдат добавяни нови отношения към него. Всичко това води до много по-лесна поддръжка и разбираемост на кода на приложението.

Взимайки предвид горните диаграми, може да си представим какъв резултат би дал Aspect.Net и АОП техниките като цяло, ако се използват в реален комерсиален проект. Именно и това беше целта на тази дипломна работа – да реализира инструмент, с който използването на АОП техники би било възможно и в .Net платформата. Цел, която смятам е напълно удовлетворена.

7.5 Обобщение

В тази глава предоставихме един реален пример на употреба на Aspect.Net. Разгледахме също и алтернативния (стандартния ООП) начин за реализация на дефинираните изисквания. Сравнихме двата подхода по голям брой показатели, описани в точка 7.4, като изтъкнахме и редицата проблеми, които Aspect.Net разрешава чрез използването на абстракции и концепции от света на АОП.

8 Изводи и заключения

В тази глава ще разгледаме изводите, до които достигнахме след завършване на реализацията на Aspect.Net. Ще започнем с кратка дискусия по отношение на целите, поставени в дипломната работа и това как и в каква степен тези цели бяха удовлетворени. Ще разгледаме също и текущите ограничения на системата, както и бъдещите насоки за развитие на Aspect.Net.

8.1 Преглед на поставените цели

В първа глава разгледахме и поставихме целите на дипломната работа. Вярвам, че успях да удовлетворя в голяма степен всяка една от тях. Нека покажем това:

1. **Дизайн и реализация на АОП инструмент за Microsoft .Net Framework, позволяващ използването на АОП механизми в .Net**

   В глава 4 и 5 представихме най-важните решения и подходи, както в дизайна, така и по отношение на реализацията на Aspect.Net. Смятам, че резултата удовлетворява всички изисквания в глава 4, както и че предоставя възможност за бъдещи разширения и подобрения. Подходящо бяха описани всички проблеми, възникнали по време на реализацията, начините по които бяха разрешени и ефекта, който оказаха на окончателната версия на Aspect.Net

2. **Дизайн и реализация на разширение за Microsoft Visual Studio 2003, което улеснява и облекчава работата на разработчиците по отношение на АОП в .Net платформата**

   Функционалността, която Aspect.Net реализира е предоставена на потребителите като разширение на средата за разработка Microsoft Visual Studio. Повече информация за Microsoft .Net, както и за модела на автоматизация Microsoft Visual Studio бе предоставена в глава 3. В Приложение 1 може да бъде намерена информация по отношение на конкретната реализация на самото разширение.

3. **Оценка на разработката по следните показатели – начин на реализиране на аспекти, методи за композиция, възможности за модуляризация, интегриране в софтуерния процес, използваемост.**

В глава 6 предоставихме оценка на Aspect.Net, сравнявайки го с други разработки в областта. Показахме, че въпреки краткото време за реализация на инструмента, той притежава завидно количество характеристики, които го нареждат близо, а понякога и преди, другите далеч по-стари, университетски разработки.

4. **Изследване на MS .Net Framework с цел установяване на възможностите за успешна реализация на AOCP в рамките на фреймуърка**

   Microsoft .Net беше разработен като наследник на Windows платформата, като една от многото цели беше по-лесна разработка на приложения. Но дали Microsoft .Net предоставя повече от останалите платформи за AOCP? В глава 6 разгледахме области от фреймуърка, които смятаме, че могат да предоставят или поддържат AOCP техники.

8.2 Ползите от Aspect.Net

Смятам, че създадох инструмент, който позволява модуляризиране на пресичащи се отношения в .Net, който не изисква специфични синтактични конструкции. Посредством разширението на средата на разработка Microsoft Visual Studio и поддръжката на междуезиково свързване, инструментът става максимално лесен и удобен за употреба за програмистите. Всичко това се надявам да доведе до по-голяма популярност на инструмента сред програмистките среди.

8.3 Ограничение на системата

Въпреки практическата си насоченост, Aspect.Net е само изследователски прототип и като такъв има множество ограничения. Ограниченията ще ги категоризираме и опишем в следните секции:

8.3.1 Ограничения при идентифициране на точките на свързване

Една от областите, където има ограничения е идентифицирането на точките на свързване. Въпреки, че реализацията на Aspect.Net покрива всички изисквания, далеч по-гъвкав и експресивен механизъм за идентифициране би могъл да бъде реализиран. Един такъв механизъм би улеснил в голяма степен

дефинирането на условията, на които трябва да отговаря дадена точка на свързване, за да бъде включена към даден pointcut.

8.3.2 Семантични ограничения

Това е може би областта на Aspect.Net, където има най-много ограничения, сравнено с някои от другите комерсиални или университетски разработки. Семантиката на Aspect.Net е подмножество на тази на AspectJ. Най-сериозните ограничения са:

- Липса на богат модел на точките на свързване – в момента Aspect.Net поддържа точки на свързване при извикване на метод. За разлика, AspectJ поддържа точки на свързване при възникване на изключение, статична инициализация на класовете и т.н.
- Липса на богат набор предикати, които се използват при дефинирането на pointcut – cflow, cflowbelow, within и т.н.
- Липса на поддръжка на динамично свързване на аспектите по време на изпълнение.
- Липса на поддръжка на аспектен полиморфизъм. Това ограничение е пряко следствие от липсата на поддръжка на динамично свързване на аспектите.

8.4 Насоки на развитието на Aspect.Net

Вярвам, че Aspect.Net постигна основната си цел – да предостави инструмент, с който могат да се модуляризират пресичащи се отношения. За да стане обаче наистина богат на функционалност инструмент, който може да се наложи като стандарт в общността на разработчиците на .Net платформата, трябва да се извършат редица подобрения в следните области:

8.4.1 Семантика

Както вече споменахме Aspect.Net реализира семантика, която е подмножество на семантиката в други разработки – AspectJ например. Това не е проблем за конкретния прототип, но определено трябва да се обърне внимание и да се реализира семантика, която е не само идентична на тази на AspectJ, а също взима под внимание и спецификите на Microsoft .Net.

8.4.2 Механизъм на свързване

В момента Aspect.Net поддържа само статична форма на свързване на аспектите и базовата функционалност. Определено това е сериозно ограничение и следващите версии би трябвало да позволяват също динамично свързване. От реализационна точка това не е проблем, а дизайнът на Aspect.Net би позволил относително лесна реализация на подобно разширение.

8.4.3 Разширението на MS Visual Studio 2003

За целите на дипломната работа, текущото разширение на MS Visual Studio 2003 представлява просто входна точка за самото свързване на аспектите и базовата функционалност. От гледна точка на реално ползване на инструмента, редица подобрения могат да се направят по отношение на тази точка на интеграция, някои от които са:

- Разработване на графичен дизайнер, с който лесно и удобно може да се дефинира свързването на базовата и аспектната функционалност. Наличието на такъв дизайнер би улеснило всички потребители на Aspect.Net, тъй като не би им се налагало да пишат на ръка конфигурационния XML файл.
- Разработване на диалог за конфигуриране на множеството настройки на самото свързване, реализирано от Aspect.Net.

8.4.4 Тестване

Единствената техника за оценяване на нова техника за програмиране е построяването на системи, които я използват. Определено скромният пример в глава 6, секция 6.3, не е достатъчен и има нужда от повече експерименти и реални случаи на употреба, за да може да се оцени полезността на инструмента.

8.4.5 Общи подобрения

Оптимизация – има много текущи места в реализацията на Aspect.Net, където могат да се приложат широк набор оптимизации. До момента основният фокус не е паднал върху това, тъй като оптимизациите бяха извън обхвата на дипломната работа.

Друга област за подобрение е библиотеката, която ползваме – RAIL. В процеса на реализация на дипломната работа открих, че тази библиотека има много сериозни проблеми, а именно:

- липса на нужна функционалност;
- множество проблеми с очаквани, но неработещи функционалности;
- проблеми с производителността.

Повечето от критичните проблеми, които срещнах ги разреших сам, тъй като разполагам със сорс кода на библиотеката. За съжаление обаче, сравнително късно открих, че използването на RAIL доведе до повече проблеми, отколкото ползи. Така че в бъдеща версия на Aspect.Net трябва да се помисли за премахването на RAIL и реализирането на собствена, компактна и ефективна библиотека за инструментиране на .Net код.

8.4.6 Съвместимост с Microsoft .Net 2.0 и Microsoft Visual Studio 2005

Конкретният инструмент адресира и е съвместим с Microsoft .Net 1.1 и Microsoft Visual Studio 2003. През изминалата година на пазара се наложиха по-новите версии на фреймуърка и средата за разработка, с които бъдеща версия на Aspect.Net би трябвало да бъде съвместима. По отношение на средата на разработка, почти никакви усилия не са нужни за адаптация на текущото решение. Не по същия начин стои въпросът с адаптацията към .Net 2.0. В новата версия има някои семантично нови елементи и нововъведения, които трябва да бъдат анализирани на MSIL ниво и да бъде преценено какви и евентуално дали въобще биха били нужни промени в Aspect.Net, за да може да бъдат поддържани тези нови парадигми.

Използвана литература

[1] **Elrad T, Aksit M, Kiczales G, Lieberherr K, and Ossher H, 2001**.Discussing aspects of AOP. Communications Of the ACM, 44(10):33–38, 2001.

[2] **Kiczales G, 2001**. An overview of AspectJ. Proceedings of the 5th European Conference on Object Oriented Programming (ECOOP), 2001.

[3] **Kiczales G, 1997** Aspect oriented programming. Proceedings of the European Conference on Object Oriented Programming (ECOOP), 1997.

[4] **Ossher H, Tarr P, 2001**. Multi-dimensional separation of concerns and the hyperspace approach. Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, 2001.

[5] **MSDN** Visual Studio 2005 Automation Model, http://msdn2.microsoft.com/en-us/bb187340(vs.80).aspx

[6] **AOSD** Homepage. http://www.aosd.net .

[7] **AopDotNetAddin** Homepage http://www.geocities.com/m_mesalem/aop.html

[8] **AspectJ Team**. AspectJ Homepage. http://aspectj.org .

[9] **AspectSharp** Homepage http://sourceforge.net/projects/aspectsharp/

[10] **EOS** Homepage http://www.cs.virginia.edu/~eos

[11] **PARC 1997,** http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-ECOOP97/for-web.pdf

[12] **RAIL** Project Homepage http://rail.dei.uc.pt/

[13] **Taosad Homepage** http://trese.cs.utwente.nl/taosad/separation_of_concerns.htm

[14] **Ohad Barzilay, Schmuel Tyszberowicz** – Call and execution semantics in AspectJ

[15] **T. Elrad, R. E. Filman, and A. Bader**. Aspect-oriented programming: Introduction. Comm. ACM, 44(10):29–32, 2001

[16] **J. D. Gradecki and N. Lesiecki**. Mastering AspectJ: Aspect-Oriented Programming in Java. John Wiley & Sons, 2003.

[17] **G. Kiczales.** Aspect-oriented programming. In ECOOP '97: European Conference on Object-Oriented Programming, 1997. Invited presentation.

[18] **R. K. Keller, R. Schauer, S. Robitaille and P. Pagé**. Pattern-Based Reverse-Engineering of Design Components. In Proceedings of the 21st International Conference on Software Engineering (ICSE'99), pages 226–235, May, 1999.

[19] **H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal**. Specifying subjectoriented composition. Theory and Practice of Object Systems, 2(3):179–202, 1996.

[20] **M. Mezini and K. Lieberherr.** Adaptive plug-and-play components for evolutionary software development. In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications, October, 1998.

[21] **Doug Kimelman, Multidimensional tree-structured spaces for separation of concerns in software development environments**. Position paper, OOPSLA '99 Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems, http://www.cs.ubc.ca/~murphy/multid-workshop-oopsla99.

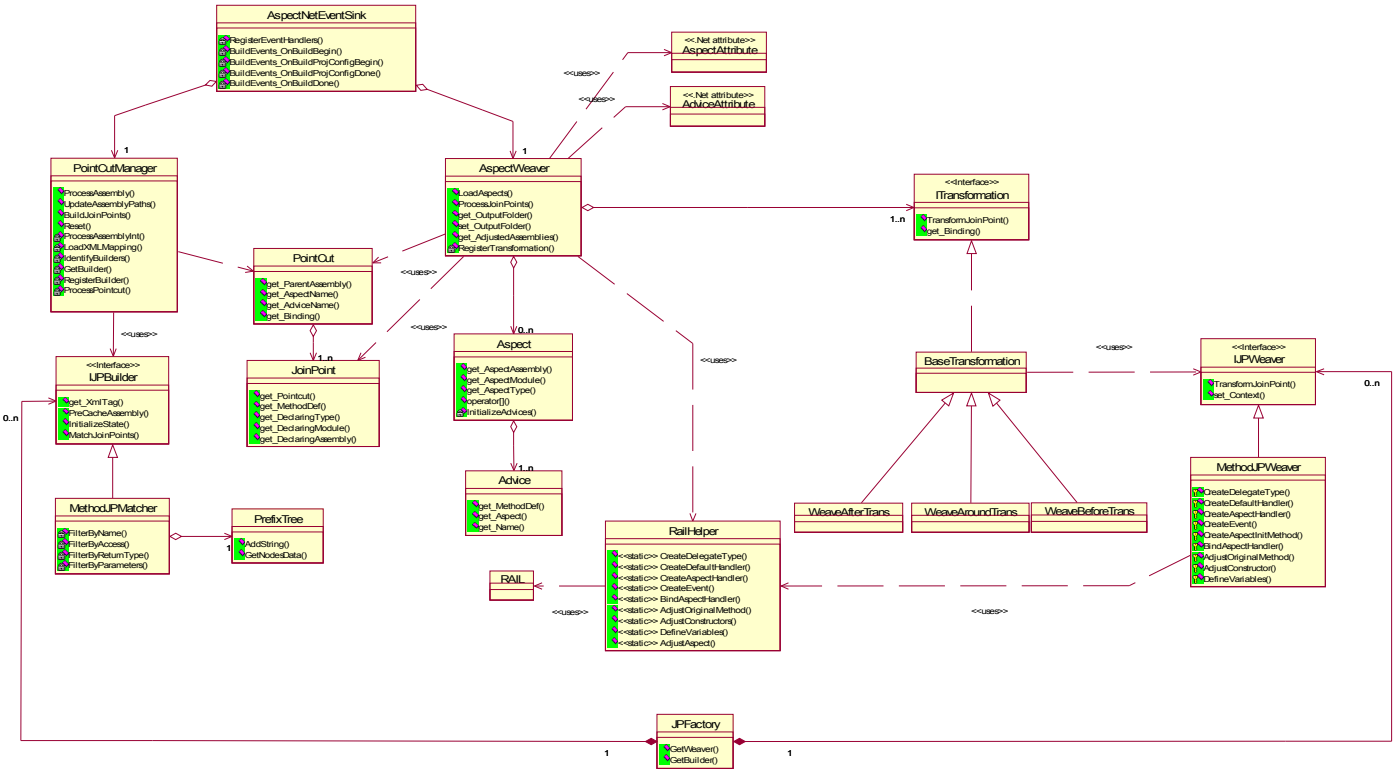
[22] **J. Rumbaugh, I. Jacobson, and G. Booch**. Unified Modeling Language Reference Manual. Addison-Wesley, 1998.

[23] **E. Gamma, R. Helm, R. Johnson, and J. Vlissides**. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

Приложение 1 - UML диаграми на решението

Клас диаграма

Фигура 28 представя клас диаграма на обектния модел на Aspect.Net. ***[22]*** Повече обяснения по отношение на различните класове, техните отговорности и взаимоотношения, може да се намери в следващите секции на приложението.



Фигура 28 - Клас диаграма на обектния модел на Aspect.Net

Основни класове, роли и отговорности

В тази секция ще предоставим по-подробно описание на илюстрираните по-горе класове. Ще разгледаме в по-големи детайли техните отговорности, роли и взаимодействие.

Като цяло дизайнът на системата обхваща две основни групи класове, реализиращи две основни функционалности: ***[23]***

- идентифицирането на точките на свързване, указани чрез XML файла;
- реализиране на самото свързване на аспектите с базовия код.

Самото свързване на аспектите може да се представи като трансформация върху базовия код ***[18]***. Самите трансформации могат да бъдат 3 вида, като от това се определя кога се извиква съветът от съответния аспект:

- преди извикването на базовия метод;
- след извикването на базовия метод;
- вместо извикването на базовия метод.

По-долу ще изброим и предоставим кратка информация за класовете, изобразени в клас диаграмата.

IJPBuilder

Този интерфейс цели обособяването в един модул на функционалността, която се грижи за:

- индексиране на съдържанието на асемблито, което се обработва в момента;
- зареждане и интерпретиране на части от съдържанието на XML конфигурацията, грижещи се за идентифицирането на точките на свързване;
- идентифициране на точките на свързване

Индексирането на съдържанието на асемблито, което се обработва е нужна, защото един и същ IJPBuilder може да бъде използван за дефиницията на множество pointcuts. С оглед на оптимизация, на всеки IJPBuilder е предоставена възможност да индексира съдържанието на файла преди да бъде използван за идентифициране на точки на свързване.

Обикновено на всеки IJPBuilder съответства конкретен XML елемент, който от своя страна може да съдържа произволен XML. За да се абстрахира базовата функционалност от конкретните детайли по интерпретирането на тези части от XML файла, всеки IJPBuilder е отговорен за зареждането и интерпретацията на елементите от XML файла, за които той предоставя съответния таг.

Като резултат от всичко казано по-горе, за разширяването на Aspect.Net с нова функционалност, идентифицираща друг тип точки на свързване, просто трябва да бъдат реализирани и регистрирани нови инстанции на този интерфейс.

MethodJPMatcher

По подразбиране, Aspect.Net реализира само един вид точки на свързване, а именно изпълнението на метод. За целта имаме класа MethodJPMatcher, реализиращ интерфейса IJPBuilder, който идентифицира методи като точки на свързване. В момента този клас поддържа филтриране по следните критерии:

- име на клас;
- име на метод;
- параметри (име, тип);
- тип на връщаната стойност;
- модификатор на достъп – публичен, частен, предпазен (public, private, protected).

За да се ускори най-бавната част от идентифицирането, а именно търсенето по име на метод, MethodJPMatcher използва реализация на префиксно дърво, в което съхранява всички методи, на всички дефинирани типове в съответното асембли.

PrefixTree

Този клас реализира стандартно префиксно дърво.

PointCutManager

Този клас имплементира функционалността, свързана с идентифицирането и контрола на точките на свързване. За да може да изпълнява тази си роля, той има нужда от следната информация като входни данни:

- XML файл, описващ множеството от точки на свързване, както и аспектите (и съветите), които трябва да бъдат интегрирани с базовия код;
- Обекти, реализиращи интерфейса IJPBuilder, способни да намерят точките на свързване и да се самоинициализират от XML конфигурацията;
- Базовия код, който бива сканиран за съответните точки на свързване.

Веднъж подадена горната информация, PointCutManager е готов да изпълнява основната си цел – идентифициране на множеството от точки на свързване. Самото идентифициране на точките на свързване става по следния начин:

- като първа стъпка зарежда се и се верифицира спрямо Aspect.Net XSD схемата и се анализира съответният XML файл;
- за всеки един обект, реализиращ интерфейса IJPBuilder се предоставя възможност да кешира или индексира текущото асембли;
- за всяка дефиниция на pointcut в XML файла, се идентифицират всички IJPBuilder наследници, които трябва да сканират асемблито и за всеки един се:
    - извиква методът InitializeState, за да зареди параметрите, които ще се използват за сканиране;
    - извиква методът MatchJoinPoints, който намира всички отговарящи точки на свързване от текущото асембли.

PointCut

Когато PointCutManager класът идентифицира точките на свързване, те се групират в PointCut клас. Класът PointCut е колекция от JoinPoint класове и като такъв предоставя функционалност за добавяне, премахване и достъпване на елементите си. Той също притежава някои атрибути, общи за всички принадлежащи към него точки на свързване (мета информация, вид на свързване и т.н.)

JoinPoint

JoinPoint съдържа мета-информацията, необходима за уникалното идентифициране на дадена точка на свързване в рамките на едно асембли. Част от тази мета-информация е:

- име на типа, към който принадлежи даденият метод;
- име на метода, който отговаря на точката на свързване;
- име на асемблито, което дефинира дадения тип и точка на свързване;
- тип на свързването, според дефиницията в XML файла.

Информацията, съдържаща се в инстанциите на този клас, представлява връзката между PointCutManager и AspectWeaver класа.

AspectWeaver

Този клас реализира логиката по избиране и изпълняване на трансформациите (реализиращи интерфейса ITransformation) върху базовия код,

енкапсулиран в JoinPoint. Като входни данни, той получава асемблитата с аспектния код. След зареждането им ги индексира и построява хеш таблица, като за генерирането на ключовете на стойностите се използва самият тип на аспектите (т.е. класовете, които имплементират съответният аспект). По този начин, когато в последствие трансформациите търсят аспектния код, който да вмъкнат в базовия, те бързо и лесно могат да достъпят съответните данни по самия тип на аспекта.

Тъй като свързването се реализира чрез трансформации, а се контролира от AspectWeaver, то самите класове, реализиращи трансформациите се инстанцират и агрегират от AspectWeaver класа. Тук съответно е и мястото, което трябва да бъде модифицирано, при евентуални бъдещи нови видове трансформации (извън текущите, реализиращи before, after и around свързване) – след реализиране на съответен наследник на интерфейса ITransformation, последният трябва да бъде регистриран в конструктора на AspectWeaver.

За да почне да свързва базовия с аспектния код, AspectWeaver му се подава множество от JoinPoint класове, които биват обходени и всеки от тях бива използван от подходяща трансформация, предварително регистрирана в AspectWeaver и указана в XML файла за съответната точка на свързване.

AspectWeaver още предоставя верификация на изходния (след свързването) код, така че по време на разработка на евентуални разширения, програмистите да могат лесно и удобно да разберат дали произведеното от тях разширение генерира валиден .Net код.

Aspect

Този клас представлява колекция от Advice класове. Като такъв той предоставя методи за добавяне, премахване и модифициране на елементите си. Отговорен е също и за зареждането и създаването на елементите си, т.е. обектите от тип Advice.

Advice

Този клас съдържа мета-информацията нужна за уникално идентифициране на съответния метод на даден аспектен клас, в дадено асембли:

- име на асемблито;
- име на класа на аспекта;

- сигнатура на метода, който реализира съответния съвет.

ITransformation

Този интерфейс се използва за представяне на всички видове свързване на аспектен код в системата. Всички евентуални бъдещи нововъведени типове свързване трябва да бъдат създадени като класове, реализиращи този интерфейс.

В момента Aspect.Net предоставя три вида трансформации по отношение на реда на извикване на аспектния код спрямо базовия:

- извикване на съвета преди извикване на базовия метод (реализиран от WeaveBeforeTrans);
- извикване на съвета след извикване на метод (реализиран от WeaveAfterTrans);
- извикване на съвета вместо извикване на базовия метод (реализиран от WeaveAroundTrans).

И трите класа се наследяват от класа BaseTransformation, който всъщност е отговорен и за 3те вида свързване. Все пак трите класа са нужни, ако в бъдеще трябва да се добави функционалност, различна за всяка от трансформациите.

BaseTransformation

Това е класът, който реализира интерфейса ITransformation. Реализацията на метода TransformJoinPoint е изключително проста:

- получава инстанция на клас, реализиращ IJPWeaver от JPFactory класа;
- създава контекст на свързването (инстанция на WeaveContext);
- извиква метода TransformJoinPoint на обекта от тип IJPWeaver.

IJPWeaver

Класовете реализиращи този интерфейс са способни да свържат даден съвет към дадена точка на свързване. В зависимост от точките на свързване и начина на постигане на самото свързване са възможни различни реализации на интерфейса IJPWeaver. В момента Aspect.Net предлага само една реализация – класът MethodJPWeaver

MethodJPWeaver

Това е класът, който реализира свързването на съветите към методи на базовата функционалност. Тук няма да се спираме на самия алгоритъм и идеята зад него, която доведе до текущата му реализация, тъй като това беше детайлно разгледано в глава 5, точка 5.2.

Самата работа на ниво MSIL се извършва чрез използването на класа RailHelper.

RailHelper

Този клас играе ролята на посредник между Aspect.Net и библиотеката Rail. Ако някой ден имаме нужда да сменим версията на Rail с по-нова или въобще да сменим библиотеката Rail с друга, единствено този клас ще трябва да бъде модифициран.

JPFactory

Както вече видяхме, дизайнът на Aspect.Net предоставя необходимите абстракции, с които едно бъдещо подобряване или разширяване на функционалността би било относително лесно. Нека припомним основните места, където бъдещите модификации най-вероятно биха се случили:

- интерфейсът IJPBuilder позволява да бъде разширена текущата функционалност по идентифициране на точки на свързване;
- интерфейсът IJPWeaver позволява да бъде разширена текущата функционалност по свързване на аспектната функционалност в точките на свързване.

С други думи, ако искаме да разширим Aspect.Net, така че той да поддържа нов тип точки на свързване – прихващане на изключения, това което трябва да се направи е:

- да се реализира клас ExceptionJPMatcher, който реализира интерфейса IJPBuilder и съдържа функционалност откриваща новите точки на свързване;
- да се реализира клас ExceptionJPWeaver, който реализира интерфейса IJPWeaver и съдържа функционалност, която може да интегрира аспектната функционалност в новите точките на свързване.

За да може всичко, свързано с разширението на Aspect.Net да бъде на едно място, ние създадохме клас JPFactory, който предоставя средства, с които може да се регистрират обекти от тип IJPBuilder и IJPWeaver в него, така че да могат да бъдат ползвани навсякъде другаде, без да се налагат промени в други класове.

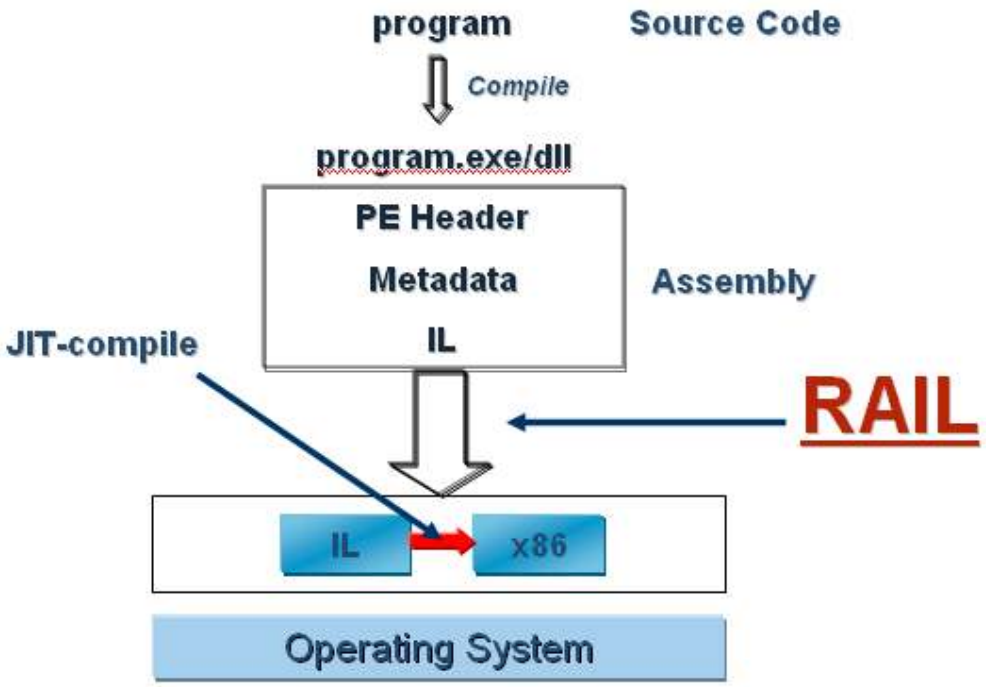
Приложение 2 – Библиотека RAIL

Една от основните характеристики на повечето модерни езици е, че те се компилират до междинен код, който се контролира и изпълнява от изпълнимата среда на дадения език. Обикновено тази изпълнима среда позволява динамичното зареждане на типовете по време на изпълнение на програмите. Два добре известни примера са Java и Microsoft .Net Framework.

Интересен страничен ефект на междинния код е, че преди да бъде зареден и изпълнен от изпълнимата среда на езика, е възможно кодът да бъде инструментиран – да бъдат добавяни или премахвани инструкции, променяни дефинициите и употребата на класовете, константите и променливите. Основната идея е, че кодът може да бъде променян преди неговото изпълнение от средата. Тези трансформации могат да бъдат стартирани или след компилация на кода или непосредствено преди неговото изпълнение. Този подход може да бъде доста мощен инструмент за постигане на иначе труднопостижими цели. Например – промяна на извикването на методи, класове, сериализиране/паралелизиране на изпълнението и т.н.

RAIL (Runtime Assembly Instrumentation Library) е библиотека, реализираща програмен интерфейс, който позволява .Net асемблита да бъдат инструментирани преди тяхното зареждане и изпълнение ***[12]***. В ОИС съществуват класове (AppDomain и ResolveEventHandler), които предлагат динамичното зареждане и промяна на код от програмиста. В същото време механизмите на рефлекция (reflection) в .Net са изключително мощни – не само могат да се изследват типове по време на изпълнение на програмата, а също могат да бъдат генерирани нови типове и асемблита. Идеята на RAIL е да запълни празнината между двете концепции – предоставяне на API, позволяващо на програмиста да прихваща зареждането на класовете, да модифицира наличния код в асемблито, преди той да бъде зареден от виртуалната машина. Използвайки RAIL е възможно да се указват трансформации, които трябва да се изпълнят преди зареждането и изпълнението на дадено асемби. Трансформациите се указват чрез използването на абстрактно API, изолиращо програмиста от детайли от ниско ниво като MSIL и двоичния формат на асемблито.

Фигура 29 илюстрира процеса и методологията на библиотеката:



Фигура 29 – Процес и методология на RAIL

Конкретно по отношение на Aspect.Net, библиотеката RAIL наистина е изключително удачен инструмент, който улеснява разработката на проекта, като предоставя готово, работещо и тествано API за манипулиране и трансформиране на MSIL код. По този начин задачата по свързването на аспектите и базовия код се свежда до генериране на подходящ набор от трансформации, които да бъдат подадени на APIто на RAIL библиотеката, която от своя страна произвежда като резултат асембли, съдържащо аспектирания код.

Приложение 3 – Език AspectJ

AspectJ е лесен за употреба и практичен език, който е аспектно-ориентирано разширение на езика Java, улесняващо модуляризирането на пресичащи се отношения. ***[2]***. Той също е и основата за емпирична оценка на АОП в истинска среда.

Език

AspectJ разширява езика Java с поддръжката на два типа пресичане на отношенията. Първият тип позволява на разработчика да дефинира допълнителна реализация, която да се изпълнява в предварително дефинирани точки на свързване – това е т.нар. **динамично пресичане**. Втория тип позволява на разработчика да дефинира нови операции във вече съществуващи типове, наричано още **статично пресичане** ***[17]***.

Динамично пресичане

Динамичното пресичане в AspectJ е базирано на операции с множества. Точка на свързване е дефинирана точка от изпълнението на програмата. Pointcuts са механизъм за достъпване до множество от точки на свързване, както и някои общи за всички точки характеристики. Съвет е конструкция, подобна на метод, която позволява допълнителна реализация да бъде свързана в точките на свързване. Аспект е абстракция, позволяваща модуляризацията на пресичащо се отношение, която е изградена от pointcut, съвети и Java код. ***[20]***

Статично пресичане

Статичното пресичане в AspectJ е постигнато чрез така наречената интроспекция. Нови член променливи и методи могат да бъдат добавени статично от аспектните в базовите класове. Това е мощен инструмент, не само от гледна точка на това, че може да бъде променяно поведението на даден базов клас, а също и взаимоотношенията между базовите класове

Модел на точките на свързване

Моделът на точките на свързване е критичен елемент във всеки АОП език ***[16]***. В AspectJ този модел е дефиниран като предефинирани точки от

изпълнението на дадена програма. В този модел точките на свързване могат да бъдат дефинирани като всички места, където се изпълнява достъп до член-променливи, изпълняване на методи, обработка на изключения или статична инициализация. Следващата таблица предоставя пълен списък на динамичните точки на свързване в AspectJ ***[8]***:

Точка на свързване	**Точка на изпълнение в програмата**
method call	метод или конструктор на базовия клас е извикан
method call reception	получаване на обръщение към метод или конструктор
method execution	изпълнение на метод ***[14]***
field set or get	достъп до член променлива на обект
exception handling execution	изпълнен е код, обработващ изключение
class initialization	изпълнение на статичен инициализатор
object initialization	изпълнение на статичен инициализатор по време на създаване на инстанция на клас

.