СОФИЙСКИ УНИВЕРСИТЕТ „СВ. КЛИМЕНТ ОХРИДСКИ”
ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА
КАТЕДРА „ИНФОРМАЦИОННИ ТЕХНОЛОГИИ”

Дипломна работа

Компонент за съхранение на криптирани данни в Pocket PC

Стоян Йорданов Йорданов
специалност „Разпределени Системи и Мобилни Технологии”
факултетен № M-21325

Научен ръководител: доц. д-р Силвия Илиева

София, февруари 2007

Съдържание

1. Увод ____ 1
1.1. Цел и задачи ____ 2
1.2. Полза от реализацията ____ 3
1.3. Структура на дипломната работа ____ 3
2. Обзор на проблемната област ____ 6
2.1. Съхранение на криптирани данни ____ 6
2.2. Pocket PC ____ 7
2.3. Механизъм за съхранение на данните ____ 9
3. Проектиране на решението на проблема ____ 12
3.1. Изисквания към компонента ____ 12
3.1.1. Данни ____ 12
3.1.1.1. Основи ____ 12
3.1.1.2. Предмети ____ 13
3.1.1.3. Допълнителни данни на приложението ____ 14
3.1.2. Операции ____ 14
3.1.2.1. Създаване, отваряне и затваряне на файл ____ 15
3.1.2.2. Списък на предметите ____ 15
3.1.2.3. Четене и запис на допълнителни данни ____ 16
3.1.2.4. Четене на мета-данните за предмет ____ 18
3.1.2.5. Четене на съдържанието на предмет ____ 18
3.1.2.6. Запис на мета-данните за предмет ____ 19
3.1.2.7. Запис на предмет ____ 19
3.1.2.8. Изтриване на предмет ____ 20
3.2. Файлов формат ____ 20
3.2.1. Основи ____ 20

3.2.2. Заглавен блок ____ 29
3.2.3. Масив с допълнителни данни на приложението ____ 30
3.2.4. Списък на предметите ____ 32
3.2.5. Външни файлове ____ 33
3.3. Архитектура на компонента ____ 35
3.3.1. Комуникация чрез потоци ____ 36
3.3.2. Дескриптори ____ 37
3.3.2.1. Общи и служебни блокове данни ____ 37
3.3.2.2. Представяне на дескрипторите като обекти ____ 38
3.3.2.3. Йерархия от дескрипторни класове ____ 39
3.3.2.4. Правилна работа с дескрипторите ____ 40
3.3.3. Представяне на файловите примитиви ____ 41
3.3.3.1. Заглавен блок ____ 41
3.3.3.2. Предмет ____ 42
3.3.3.3. Елемент на свързания списък на предметите ____ 43
3.3.3.4. Списък на предметите ____ 44
3.3.3.5. Допълнителни данни ____ 45
3.3.3.6. Елемент на масива с допълнителните данни ____ 45
3.3.3.7. Масив с допълнителните данни ____ 46
3.3.4. Управление на пространството ____ 46
3.3.4.1. Модул за управление на пространството ____ 47
3.3.4.2. Под-модул за управление на свободното пространство ____ 48
3.3.4.3. Под-модул за управление на блоковете с данни 49
3.3.5. Модул за работа с файла ____ 49
3.3.6. Публичен интерфейс ____ 50

3.3.7. Взаимодействие между модулите при операции с блокове 50
3.3.7.1. Запис на блок с данни 51
3.3.7.2. Четене на блок с данни 52
3.3.7.3. Освобождаване на блок с данни 52
3.3.7.4. Свиване на файла 53
3.3.8. Взаимодействие между модулите при сложни операции 55
3.3.8.1. Списък на предметите 55
3.3.8.2. Запис на допълнителни данни 56
3.3.8.3. Четене на допълнителни данни 57
3.3.8.4. Запис на предмет или на мета-данните му 58
3.3.8.5. Четене на предмет или на мета-данните му 59
3.3.8.6. Изтриване на предмет 60
4. Описание на реализацията 61
4.1. Общи съображения 61
4.1.1. Стратегия за заделяне на място 61
4.1.1.1. Стратегии, използващи блокове с фиксиран размер 61
4.1.1.2. Стратегии, използващи блокове с произволна дължина 62
4.1.1.3. Избор на стратегия 64
4.1.2. Данните – като масиви от байтове 65
4.1.3. Двупасов запис на данни 66
4.2. Основи 68
4.2.1. Бавни операции 69
4.2.1.1. SlowOperationController 69

4.2.2. Помощни методи ____ 70
4.2.3. Основни типове данни ____ 72
4.2.3.1. ItemID ____ 72
4.2.3.2. IV ____ 73
4.2.3.3. Content ____ 73
4.2.3.4. EncryptedContent ____ 74
4.3. Дескриптори ____ 74
4.3.1. DataBlockDescriptor ____ 76
4.3.2. GenericDataBlockDescriptor ____ 79
4.3.3. ListElementDataBlockDescriptor ____ 79
4.3.4. CustomDataItemDataBlockDescriptor ____ 79
4.3.5. CustomDataItemArrayDataBlockDescriptor ____ 80
4.4. Файлови примитиви ____ 80
4.4.1. FileHeader ____ 80
4.4.2. Item ____ 81
4.4.3. ItemList; ListElement ____ 82
4.4.4. CustomDataItem ____ 83
4.4.5. CustomDataItemArray; CustomDataItemArrayElement 84
4.5. Модул за работа с файла ____ 85
4.5.1. DataFileManager ____ 85
4.5.1.1. Публични методи и свойства ____ 86
4.5.1.2. Частни методи и свойства ____ 86
4.5.2. SpaceManager ____ 87
4.5.2.1. Свиване на файла ____ 87
4.5.3. FreeSpaceManager ____ 88
4.5.3.1. FreeSpaceBlock ____ 88
4.5.4. DataBlockManager ____ 89

4.6. Публичен интерфейс ____ 90
4.6.1. DataStorage ____ 90
5. Тестване на компонента ____ 93
5.1. Описание на тестовата инфраструктура ____ 93
5.2. Тестване на функционалността ____ 94
5.2.1. Произволни данни и операции ____ 94
5.2.2. Повторна проверка ____ 96
5.2.3. Тестване свиването на файла ____ 96
5.2.4. Повторна проверка след свиването ____ 97
5.3. Тестване на производителността ____ 97
6. Изводи и възможности за подобрение ____ 99
7. Заключение ____ 101
Използвана литература ____ 102
Приложение 1 – Списък на фигурите ____ 103
Приложение 2 – Списък на таблиците ____ 104

1. Увод

Всеки има нужда да може да носи различни лични данни със себе си. В съвременното общество все повече и повече дейности като изпращане на електронна поща, пазаруване, или дори банкиране, могат да бъдат извършвани от удобството на дома, от парка, кафенето, библиотеката. Този прогрес обаче бързо увеличава нуждата от помнене на информация, свързана с ползването на тези услуги – електронни адреси, потребителски имена и пароли, номера на банкови сметки. Помненето на такава информация е обременяващо и трудоемко, а записването ѝ на лист хартия, носен в портфейла, би било неразумно, тъй като попадането ѝ в чужди ръце би имало неприятни последствия.

Решението на този проблем е информацията да бъде носена на сигурен и защитен носител, загубата или кражбата на който не би изложила данните на потребителя. Логична стъпка е да се прибегне до помощта на всевъзможните преносими електронни устройства, ставащи все по-достъпни и разпространени с навлизането в 21-ви век. Чрез тях всичката необходима информация може да придружава собственика си в криптиран вид, недостъпна за чуждите очи, като същевременно е удобно достъпна навсякъде и по всяко време. Повечето устройства, обаче, не предлагат сигурно съхранение на данни. Отделно, използването на външни карти памет, както често се налага за разширяване на възможностите на устройството, излага данните на допълнителен риск, тъй като данните могат да бъдат извадени от картата дори и без въвеждането на правилна потребителска парола в устройството.

Ето защо възниква нуждата от специализирано приложение, което да криптира и съхранява данните на потребителя, същевременно предоставяйки удобен достъп до тях. Освен удобен потребителски интерфейс и сигурно криптиране на данните, основ-

на функционалност на едно такова приложение е съхранението на информацията сигурно и надеждно; механизмът за съхранение трябва да предлага бърз достъп както за съхранение, така и за извличане на криптираната информация, като същевременно операциите с данните са надеждни и не допускат тяхната повреда при внезапно прекъсване на операцията (причинено, например, от внезапно свършване на батерията, изпускане на устройството и т.н).

1.1. Цел и задачи

Целта на настоящата дипломна работа е разработката на компонент за съхранение на данните на приложение за Pocket PC, позволяващо сигурно съхранение на лични данни. Компонентът трябва да предоставя бърз достъп и сигурно съхранение на големи обеми криптирани данни на приложението; самото приложение е цел на дипломната работа на Милена Йорданова (Йорданова, 2007).

За постигане на така поставената цел трябва да бъдат изпълнени следните задачи:

- Анализ на необходимостта от специализиран компонент:
    - Анализ на нуждите на приложението, използващо компонента;
    - Анализ на ограниченията и особеностите при съхранението на данни в Pocket PC;
    - Сравнителен анализ на вече съществуващи решения и технологии;
- Проектиране на подходящ механизъм (файлов формат), позволяващ ефективно съхранение и достъп до данните, имайки предвид извършените анализи;
- Анализ и проектиране на алгоритми за управление на данните:

- Надеждни алгоритми за промяна и изтриване на данни (винаги да поддържат файла в консистентно състояние по време на изпълнението си, за да се избегне опасността от повреда при непредвидени ситуации);
- Алгоритми за управление на ресурсите (заделяне и освобождаване на свободното пространство при изтриване или промяна на данни, свиване на файла с цел освобождаване на системни ресурси и т.н.);

- Проектиране на гъвкава архитектура на компонента, позволяваща максимална независимост от използващото го приложение и конкретните му типове данни;
- Разработка на компонента;
- Тестване на компонента.

1.2. Полза от реализацията

Реализацията на компонента за съхранение на данните позволява надеждното и ефективно съхранение на криптираните данни на приложението за сигурно съхранение на криптирани данни, разработвано от Милена Йорданова, което позволява на потребителите си да носят своите важни данни винаги със себе си, без да се притесняват от попадането им в чужди ръце.

1.3. Структура на дипломната работа

Настоящата дипломна работа е разделена на няколко глави, всяка от която разглежда различен аспект от решението на проблема.

В „Обзор на проблемната област" обсъждаме какви са изискванията, предизвикателствата и проблемите, свързани със съхранението на криптирани данни в устройства с ограничени изчислителни ресурси, оперативна памет и място за съхранение.

В „Проектиране на решението на проблема” се опитваме да решим тези предизвикателства и проблеми и в резултат проектираме компонент, който отговаря на високите изисквания на приложението за съхранение на лични данни. За целта първо анализираме какви са изискванията към компонента, какви типове данни трябва да се съхраняват и каква функционалност за работа с тях трябва да се предлага на приложението, което го използва. На базата на този анализ проектираме файлов формат за съхранение на данните и оптимизиран за нуждите на приложението така, че операциите с данните да бъдат бързи и надеждни. Накрая проектираме архитектурата на самия компонент за съхранение на данните който да оперира с така разработения файлов формат и да предлага изискваната функционалност на клиентското приложение, съобразявайки се с ограничените изчислителни ресурси и памет на устройството.

Проектираното решение претворяваме в дела в „Описание на реализацията”. В тази глава описваме конкретната реализация на архитектурата и файловия формат, обсъждаме йерархията от класове и избраните структури за поддържане на информацията и за ускоряване на операциите по време на работа на компонента, и се спираме подробно върху комуникацията между отделните части на компонента и как всяка от тях допринася за постигане на общата цел.

Следва „Тестване на компонента”, където разглеждаме методите, използвани за тестване на компонента в процеса на разработка. Отделено е внимание както на проверката на функционалността на компонента, така и на тестването на производителността на операциите.

Дипломната работа завършва с главите „Изводи и възможности за подобрение”, следвана от „Заключение”. В първата разглеждаме нерешените проблеми и възможностите за подобрение

пред компонента, а във втората предлагаме поглед назад към извършената работа.

2. Обзор на проблемната област

2.1. Съхранение на криптирани данни

Същността на разглежданото решение е съхранение на криптирани данни с неизвестно съдържание и големина. Те могат да бъдат просто няколко думи, въведени от потребителя, или пък съдържанието на многомегабайтов файл – за компонента за съхранение на данните това няма значение. Общото между отделните единици, които трябва да бъдат съхранявани, е, че те представят някаква конкретна информация от интерес за потребителя на приложението, която е била криптирана и трябва да бъде съхранена заедно с някаква придружаваща информация, принадлежаща на самото приложение.

Придружаващата информация може да съдържа всичко, което е необходимо на приложението за правилното класифициране и обработка на потребителските данни. В нея може, например да бъде съхранен типът на съхраняваните данни (парола, данни за банкова сметка и т.н.), име, което да бъде показвано в потребителския интерфейс за тези данни, и т.н. Ще наричаме тази допълнителна информация „мета-данни” (metadata). Мета-данните, също като самите данни, също се съхраняват в криптиран вид с цел защита на неприкосновеността на информацията на потребителя. Конкретното им съдържание се определя от клиентското приложение в зависимост от информацията, която то има нужда да знае за съхранените потребителски данни, и не от интерес за разработвания от нас компонент.

Отделно, тъй като данните и мета-данните са криптирани, клиентското приложение има нужда да може да съхрани използваните инициализационни вектори на криптографските алгоритми, за да може правилно да декриптира данните, когато това се наложи (Йорданова, 2007). Следователно компонентът за съх-

ранение на данните трябва да предлага тази възможност. При съхранение на данни или мета-данни той трябва да приема инициализационните вектори, използвани за тяхното криптиране (клиентското приложение използва различни вектори за данните и за мета-данните), и да ги връща заедно с данните или мета-данните, когато те бъдат поискани обратно.

Броят на съхранените единици информация обикновено не е твърде голям. Потребителят на клиентското приложение трябва да може да съхрани няколко десетки до няколкостотин сравнително малки обекта (пароли, банкови сметки и т.н.), но и определен брой по-големи предмети като например важни файлове и снимки.

2.2. Pocket PC

Важен фактор, с който компонентът за съхранение на данни трябва да се съобразява, е ограниченото пространство, налично за съхранение на криптираната информация. Тъй като компонентът ще бъде използван на мобилно устройство, това пространство ще е далеч по-малко, отколкото сме свикнали да имаме свободно на един персонален компютър. Дори и потребителят да използва допълнителни разширителни карти с памет, все пак шансовете са, че на тези карти ще има и друга важна за потребителя информация и че приложението все пак ще трябва да оперира с ограничени ресурси. Ето защо разработваното решение трябва да оползотворява наличното пространство максимално, доколкото това е възможно без да се жертва излишно производителността на приложението; трябва да бъде постигнат компромис между бързодействие и пестене на наличното място.

Решението трябва да се съобразява, също така, и със силно ограничената оперативна памет на мобилното устройство. Тази памет, освен, че е наистина малко, обикновено се използва едновременно както за оперативна памет, така и за съхранение на

информация, като съотношението се определя динамично от операционната система в зависимост текущите нужди. На устройството, с което разполагаме по време на разработката на настоящата дипломна работа, например (HP iPAQ hx4700), основната памет е 64МБ, която се използва както за съхранение на данни, инсталиране на програми и т.н., така и за оперативна памет. За последната са отделени 20 от наличните 64МБ, като тези 20МБ се споделят както от операционната система, така и от различни стартирани клиентски приложения. Крайният резултат е, че разработваното приложение може да се възползва само от няколко, най-много 6-7МБ оперативна памет. Отделно, устройството разполага и със 128МБ вградена flash памет, както и с разширителен слот тип Secure Digital, но те не могат да бъдат използвани като оперативна памет, а само за съхранение на данни. Всичко това означава, че приложението за съхранение на криптирани данни, и в частност разработвания като цел на настоящата дипломна работа компонент, трябва да могат да оперират с данни, които са по-големи от наличната оперативна памет. Например приложението трябва да може да криптира, а компонентът за съхранение на данните – да съхрани, данни, идващи от файл с големина 30МБ, въпреки, че устройството може да не разполага с 30 свободни мегабайта оперативна памет.

На последно място, мобилното устройство притежава ограничени изчислителни ресурси, тъй като, за разлика от настолните или дори преносими компютри, Pocket PC обикновено използват сравнително слаб процесор поради малките си размери и живот на батерията. Компонентът за съхранение на данните трябва да използва подходящи структури от данни, за да минимизира максимално натоварването на процесора на устройството и същевременно да може да осъществява бърз достъп до съхранената информация.

2.3. Механизъм за съхранение на данните

Разработваният компонент трябва да използва някакъв конкретен механизъм за съхранение на данните. Тук имаме избор дали да използваме вече съществуващо решение, или пък да разработим ново такова, специално пригодено за нуждите на приложението за съхранение на данни.

От вече съществуващите решения логичният пръв избор е файловата система. Windows Mobile (операционната система, използвана при Pocket PC) използва файлова система FAT за съхранение на файловете и директориите. Тъй като операционната система вече трябва да съхранява и категоризира информация (файлове и директории), може би бихме могли да се опитаме да се възползваме от този факт и да използваме директно файловата система на устройството като механизъм за съхранение на данните. Бързо бихме видели, обаче, че тя не е подходяща за такива цели по няколко причини. Първо, файловата система има различни ограничения, например за брой файлове в директория или за минимално пространство, заето от един файл. Въпреки, че максималният брой файлове, които може да съществуват в дадена директория, обикновено е достатъчно голям за целите на приложението, той все пак може да бъде достигнат при определени условия, ако потребителят на приложението има наистина голям брой лични данни, които трябва да бъдат съхранени (този фактор може да бъде още по-важен, ако изберем да съхраняваме мета-данните и инициализационните вектори също като самостоятелни файлове, вместо по някакъв начин вътре във файла, представящ съответния запис с потребителски данни). Минималното пространство, заето от един файл, обаче, за нас е по-важното ограничение. Тъй като при FAT всеки файл заема цяло число клъстери (поне 512 байта), съхранението на множество малки файлове за различните записи с данни би прахосвало неоправдано много място. Второ, файловата система не е пригоде-

на за постоянно отваряне и записване на файлове в директория, съдържаща голям брой от тях. Тъй като FAT използва линейно търсене в списъка с файловете в дадена директория, за да намери желания файл по име, производителността на файловата система намалява линейно с увеличаване на броя на файловете в директорията (Jansen, 2006). Всичко това показва, че трябва да се спрем на някакъв по-специализиран механизъм за съхранение на данните на приложението, който да ни позволява да не прахосваме излишно много място при съхранението им, като същевременно предлага бърз достъп до тях.

Друго вече съществуващо решение, което заслужава внимание, е SQL Server 2005 Compact Edition. Както подсказва името му, SQL Server 2005 Compact Edition е мобилна версия на популярната база данни на Майкрософт. Тя е безплатна, предлага мобилен достъп до релационна база данни и е съобразена с ограничените ресурси на съвременните мобилни устройства. Въпреки това, SQL Server 2005 Compact Edition не е оптималното решение, което търсим. Тъй като базата данни е общо решение, което не е оптимизирано специално за сценариите на конкретното приложение, тя най-вероятно би използвала наличното място на устройството неоптимално – нещо, което се стремим да избегнем. Истинският проблем, обаче, е, че SQL Server Compact Edition е ориентиран към корпоративни решения и предлага твърде много неща, от които компонентът за съхранение на данните няма нужда (отдалечен достъп до настолна база данни, синхронизация и т.н). За всичко това се заплаща с използваната от SQL Server оперативна памет – цели 5МБ (Microsoft Corporation, 2006), което в нашият случай е неприемливо.

Подходът, на който се спряхме, се състои в разработването на специализиран файлов формат[1], който да позволи сигурното и надеждно съхранение на криптираната информация (т.е. да няма възможност за повреда на данните при внезапното прекъсване на операция с тях), да предлага бърз достъп до нея както за запис, така и за четене, и да използва наличното пространство на устройството, както и неговата оперативната памет, възможно най-оптимално.

[1] Всъщност, както ще видим в точка 3.2 – „Файлов формат”, разработеното решение използва хибриден модел, при който малките записи потребителски данни, въведени директно чрез потребителския интерфейс на клиентското приложение, се съхраняват в един-единствен основен файл, докато съдържанието на записи, чиято информация идва от файлове (като например криптирани снимки, файлове и т.н.), се съхраняват в отделни файлове с цел по-малко прахосване на свободно място при тяхното изтриване и по-ниска фрагментация на свободното пространство.

3. Проектиране на решението на проблема

3.1. Изисквания към компонента

Всички изисквания към разработвания компонент се състоят в това той да може да съхранява определени типове данни сигурно и надеждно и да предлага подходящ набор от операции, които да могат да бъдат извършвани бързо с тези данни.

Нека анализираме какви точно данни трябва да може да съхранява компонентът и какви точно операции трябва да предлага на използващото го приложение.

3.1.1. Данни

3.1.1.1. Основи

Компонентът за съхранение на данните трябва да може да съхранява криптирани данни, идващи от приложение, съхраняващо записи за различни лични данни на потребителя. Тъй като тези записи трябва да могат да бъдат записвани и четени поотделно, е необходим механизъм за еднозначното им идентифициране и адресиране. За целта всеки запис трябва да притежава уникален идентификатор, чрез който приложението да може да осъществява достъп до неговите данни и мета-данни. За целите на компонента и използващото го приложение избрахме да използваме глобално-уникални идентификатори (GUID – Globally-Unique Identifier), тъй като се генерират изключително лесно и гарантират уникалност в много висока степен (бидейки 128-битови псевдослучайни числа). Това позволява идентификаторите да бъдат генерирани бързо, без притеснение, че новият идентификатор съвпада с идентификатора на вече съществуващ запис (въпреки, че теоретически това е възможно, шансовете това да се случи на практика са изключително, изключително малки).

Друг аспект на който следва да се обърне внимание е, че данните, които разработваният компонент трябва да съхранява, са криптирани. Както отбелязахме в точка 2.1 – „Съхранение на криптирани данни”, това означава, че редом с криптираната информация трябва да се съхранява и инициализационните вектори, използвани за нейното криптиране. Криптираните данни и техния инициализационен вектор са неразделно свързани, като всяко е безсмислено без наличието на другото. Ето защо, когато компонентът за съхранение на данните предава или получава криптирана информация от клиентското приложение, както и когато я съхранява, тя винаги е придружена от съответния инициализационен вектор.

3.1.1.2. Предмети

Приложението, използващо компонента за съхранение на данните, нарича всеки потребителски запис, съдържащ данни за единица потребителски криптирани данни, „предмет”. Това е и терминът, който ще използваме оттук нататък в настоящата дипломна работа.

В точка 2.1 – „Съхранение на криптирани данни” – посочихме, че за всеки предмет трябва да могат да се съхраняват два различни набора криптирана информация – първо, самото съдържание на потребителските данни (парола, банкова сметка и т.н.), и второ – допълнителна информация („мета-данни”), идваща от самото клиентско приложение и описваща съхранените данни, за да може приложението да борави с тях (например информация за това какъв точно тип данни са съхранени в този предмет, име, което да бъде показвано за него в потребителския интерфейс и т.н). Какво съдържат данните и мета-данните не е от никакъв интерес за разработвания компонент. Единственото, което трябва да знаем, е, че това са два набора криптирани данни, които трябва да бъдат съхранени заедно с техните криптографски

инициализационни вектори под определен общ идентификатор и да бъдат достъпни за клиентското приложение при поискване за четене или промяна.

3.1.1.3. *Допълнителни данни на приложението*

Освен предметите, съдържащи различни потребителски данни, приложението, използващо компонента за съхранение на данните, има нужда да може да съхранява и различни записи допълнителна информация със служебно предназначение, която не е свързана с никой конкретен предмет (за разлика от метаданните, които винаги описват някой определен предмет) (Йорданова, 2007). Такава допълнителна информация може да бъде например блок с данни, съдържащ информация за използваната криптография, криптографски ключове, пароли и т.н., като конкретното ѝ съдържание не е от значение за компонентани. Тя може да бъде криптирана или не, в зависимост от нуждите на приложението (например информация, описваща използваната криптография, по обективни причини няма как да бъде криптирана, докато информация, която не бива да бъде достъпна за неоторизирани лица, би следвало да бъде съхранена в криптиран вид).

3.1.2. Операции

Компонентът за съхранение на данните трябва да предлага на използващото го приложение определен набор операции, за да може то да съхранява необходимата криптирана информация и да оперира с нея. Изборът на операциите, които да се предлагат, трябва да бъде балансиран – трябва да се предоставят достатъчно възможности, за да може приложението лесно да извършва необходимите му операции с данните, но същевременно те трябва да бъдат достатъчно общи, за да се избегне излишна обвързаност на компонента за съхранение на данните и използващото го приложение.

След като по-горе анализирахме различните типове информация, която клиентското приложение има нужда да съхранява, стигаме до минималния набор от операции, които му позволяват да оперира с тях.

3.1.2.1. Създаване, отваряне и затваряне на файл

Тъй като компонентът за съхранение на данните използва специален файл, в който съхранява криптираната информация, логично е потребителят да може да поиска да има повече от един такъв файл. Това означава, че компонентът трябва да предлага възможност на използващото го приложение да указва изрично кой файл да бъде създаден или отворен; това може да стане просто чрез подаването от страна на приложението на символен низ, указващ пътя към него.

Когато приложението приключва работа, то трябва да затвори файла с данни, за да се освободят правилно използваните системни ресурси. Въпреки, че файловете се затварят автоматично при приключване на приложението, и че винаги държим файла в консистентно състояние, така че автоматичното му затваряне в края на програмата не би го повредила, все пак е добра практика файловете да бъдат затваряни изрично в момента, когато вече не са необходими.

Ето защо разглежданият компонент трябва да предоставя възможност за създаване на нов файл по зададен символен низ, указващ пътя към него, отваряне на вече съществуващ файл по зададен символен низ, указващ пътя към него, както и затваряне на текущо отворения файл.

3.1.2.2. Списък на предметите

След като отвори даден файл с данни, приложението трябва може да получи списък на предметите, съдържащи се в него, за да може да ги представи на потребителя или въобще – да оперира с тях. За да посрещне тази нужда, компонентът за съхранение

на данните трябва да предлага операция, връщаща масив с идентификаторите на всички съхранени предмети.

Както обсъдихме в точка 2.1 – „Съхранение на криптирани данни”, броят на съхранените предмети обикновено не е твърде голям, така че връщането на масив с всички идентификатори не би следвало да представлява проблем дори и за ограничената оперативна памет на устройството (ако имаме, например, 1000 съхранени предмета, връщането на 1000 идентификатора по 16 байта всеки – размерът на един стандартен GUID – означава предаването само на около 16КБ информация, което не е никакъв проблем).

Идентификаторите на предметите в случая са достатъчна информация, т.е. операцията не е необходимо да връща нищо повече. Ако клиентското приложение има нужда от допълнителна информация за някой от предметите (като например от метаданните или съдържанието му), то би могло да ги поиска отделно; ако ли пък не, тогава няма смисъл да предаваме излишно информация, която няма да бъде полезна. По този начин оставяме на приложението да прецени има ли нужда или не от допълнителна информация и да я изиска отделно, в случай, че му е необходима.

3.1.2.3. Четене и запис на допълнителни данни

В точка 3.1.1.3 – „Допълнителни данни на приложението” – посочихме, че приложението, използващо компонента, има нужда да може да съхрани по един екземпляр от няколко различни вида служебна информация, която не е свързана с нито един конкретен предмет, но въпреки всичко трябва да бъде съхранена във файла редом с останалите данни.

Приемаме, че клиентското приложение има дефиниран определен набор от типове допълнителни данни, с които то може да борави – например блок с информация за криптографията,

блок с данни за потребителския интерфейс, и т.н. За да покрием нуждите на приложението в тази област, разработвания от нас компонент отделя по един „слот” за всеки от различните типове допълнителни данни на клиентското приложение. Тези слотове са номерирани с числата от 0 до $n-1$, където n е броят на използваните от клиентското приложение типове допълнителни данни. Това позволява на приложението, имайки някакво статично предварително зададено съответствие (чрез изброим тип?) между различните типове допълнителни данни и съответните им номера на слотове, да съхранява и извлича обратно екземплярите от даден тип, без компонентът за съхранение на данните да трябва да притежава каквото и да било знание за това каква допълнителна информация се съхранява.

Например, ако приложението има 5 различни типа допълнителни данни, компонентът за съхранение на данните би му отделил 5 слота, номерирани от 0 до 4, за съхранение на тези данни. Знаейки, че блокът с информация за използваната криптография съответства например на слот номер 3, приложението може да го записва и извлича от компонента за съхранение на данните просто поисквайки го по този номер. Всичко, което компонентът вижда, са заявки за четене или запис за слот номер 3.

Всичко това изисква компонентът да предлага операция за запис на данни (криптирани или не) в конкретен слот, зададен по номер, и четене на данните от слот, зададен по номер. Ако данните, записвани в слота, са криптирани, операцията за запис трябва да съхранява и съответният им криптографски инициализационен вектор. При прочитане на информацията, в случай, че тя е криптирана, инициализационният вектор трябва да бъде връщан заедно с нея на клиентското приложение, за да може то да декриптира информацията правилно. За момента няма нужда от операция, изтриваща данните, записани в даден слот; прило-

жението, използващо компонента, има нужда да може да записва нови данни в слота, но не и да изпразва слот, който вече съдържа данни. В случай, че се появи нужда от такава операция, тя би могла да бъде реализирана изключително лесно (може би дори би могла да се използва модифицирана версия на вече съществуващата операция за запис в слот, като просто не ѝ бъдат подадени данни, които да запише).

3.1.2.4. Четене на мета-данните за предмет

Приложението може да има нужда да поиска само мета-данните за даден предмет, без да има нужда от съдържанието му. Това може да се наложи, например, с цел показване на предмета в потребителския интерфейс, тъй като името и типът му се съдържат в мета-данните. Ето защо компонентът за съхранение на данните трябва да предлага метод, връщащ мета-данните за даден предмет по подаден идентификатор на предмета.

3.1.2.5. Четене на съдържанието на предмет

Основна операция с данните е извличане на съдържанието на даден предмет от механизма за съхранение. Какъв смисъл би имало приложението да съхранява предмети, ако няма възможност след това да прочете съдържанието им обратно?

За да позволи на приложението да прочете съдържанието на даден предмет, компонентът за съхранение на данните трябва да предлага метод, който да връща криптираното му съдържание по подаден идентификатор на предмета. Нарочно позволяваме съдържанието да бъде прочетено отделно от мета-данните за предмета, вместо да задължаваме използващото компонента приложение да чете двете наведнъж, тъй като то може вече да разполага с мета-данните, или пък те просто да не са му необходими.

3.1.2.6. Запис на мета-данните за предмет

Компонентът за съхранение на данните трябва да предлага възможност на използващото го приложение да съхранява нови мета-данни за вече съществуващ предмет. Това може да се налага, например, при преименуване на предмет, когато презаписването на съдържанието му не би имало никакъв смисъл, особено ако става дума за голям предмет. В този случай приложението може просто да запише новите мета-данни, съдържащи новото име на предмета. Компонентът трябва, следователно, да предлага операция, която да съхранява нови мета-данни за вече съществуващ предмет, идентифициран чрез своя идентификатор.

3.1.2.7. Запис на предмет

Отделно от операцията, предлагаща възможност за запис на мета-данните за предмета, компонентът трябва да предлага и възможност за съхранение (или създаване) на цял предмет – както неговите мета-данни, така и съдържанието му – по подаден идентификатор на предмета. Ако предмет с подадения идентификатор вече съществува, то той бива подменен с новата си версия; ако ли не – бива създаден.

Операцията не дава възможност за записване само на съдържанието на предмета, за да се избегне възможността за създаване на предмет, който няма мета-данни – стремим се файлът винаги да бъде в консистентно състояние. Очакваме мета-данните за предметите да бъдат малки, буквално няколко до няколко десетки байта, описващи типа на предмета, неговото име за представянето му в потребителския интерфейс, както и може би още само няколко полета важна информация за него, така че записването на нови мета-данни всеки път, когато се генерира ново съдържание, не би следвало да представлява какъвто и да е проблем.

3.1.2.8. *Изтриване на предмет*

Логична и очаквана възможност е, след като потребителят е съхранил определена информация, да може да я изтрие, когато вече няма нужда от нея. Ето защо разработваният компонент трябва да предлага операция за изтриване на предмет по подаден неговия идентификатор.

3.2. Файлов формат

Имайки предвид обсъдените по-горе типове данни, които компонентът за съхранение на данните трябва да може да съхранява, както и операциите, които трябва да могат да бъдат извършвани с тях, стигаме до разработката на подходящ файлов формат, позволяващ съхранението на тези типове данни и позволяващ операциите с тях да бъдат бързи и надеждни.

3.2.1. Основи

Разработката на нов файлов формат следва да започне с важния избор на подходящ механизъм за съхранението на файла. Двоичен файл ли да бъде, или да използваме XML за съхранение на данните? Последователна структура ли ще има файлът, или пък ще има сложен формат, позволяващ записът на информация на произволно място в него? За да изберем правилно, нека разгледаме какви са предимствата и недостатъците на всеки от тези подходи и как те биха повлияли на обсъжданото решение.

Изборът между XML и двоичен файлов формат следва да се направи въз основа на това за какво ще бъде използван съответния файл. XML (Extensible Markup Language) е технология, наложила се през последните няколко години като основен механизъм за комуникация между приложения, уеб услуги и др., както и за запис на машинно-ориентирани файлове, които все пак трябва да могат да бъдат четени от хора. Предимство на XML файловете е, че информацията в тях е съхранена в текстов вид, и въпреки това е строго структурирана според някаква схема, избрана

така, че да възможно най-подходяща за представянето на конкретните данни, които XML файлът трябва да съдържа. Комбинацията от текстов файл и строга структура позволява XML файловете да бъдат четени както от хора, така и от различни типове софтуер в най-разнообразни среди, стига да разполагат със схемата, използвана за структурирането на информацията. Бидейки текстови файлове, XML файловете са по-големи, както и по-бавни за четене и запис от двоични файлове, представящи същите данни, но пък са по-преносими и лесни за съвместна работа на няколко приложения.

Двоичните файлове, от своя страна, могат да бъдат много по-компактни от XML файловете и да бъдат четени и писани много по-бързо поради липсата на необходимост информацията да бъде представяна в текстов вид и изрично да се обозначава структурата ѝ (както това става при XML чрез използването на различни тагове). Това чудесно свойство на двоичните файлове, обаче, е за сметка на затруднената съвместна работа с други приложения, тъй като двоичните файлове обикновено са силно свързани с вътрешното представяне на данните на генериращата ги програма, което може да не отговаря на вътрешното представяне на данните, използвано от други приложения.

Изборът на подходяща стратегия за съхранение на данните също е особено важен, тъй като има директно влияние върху производителността и надеждността на операциите с данните. Тук трябва да изберем между два принципно различни механизма – последователно съхранение на данните или файл с непоследователен достъп.

При използването на последователен файлов формат информацията се записва последователно във файла, без да се оставят празни места. Новозаписана информация може да се добави в края на вече съществуващ файл, но изтриването или промя-

ната на информация във файла (което често е свързано с промяна в дължината на записаните данни) налагат генерирането на целия файл наново, за да може информацията в него отново да бъде наредена последователно. Типичен пример за използването на последователна структура са XML файловете. В тях информацията е наредена последователно, според предварително определена схема, като записът на нова информация или изтриването или промяна на вече съществуваща информация налага генерирането наново на целия файл.

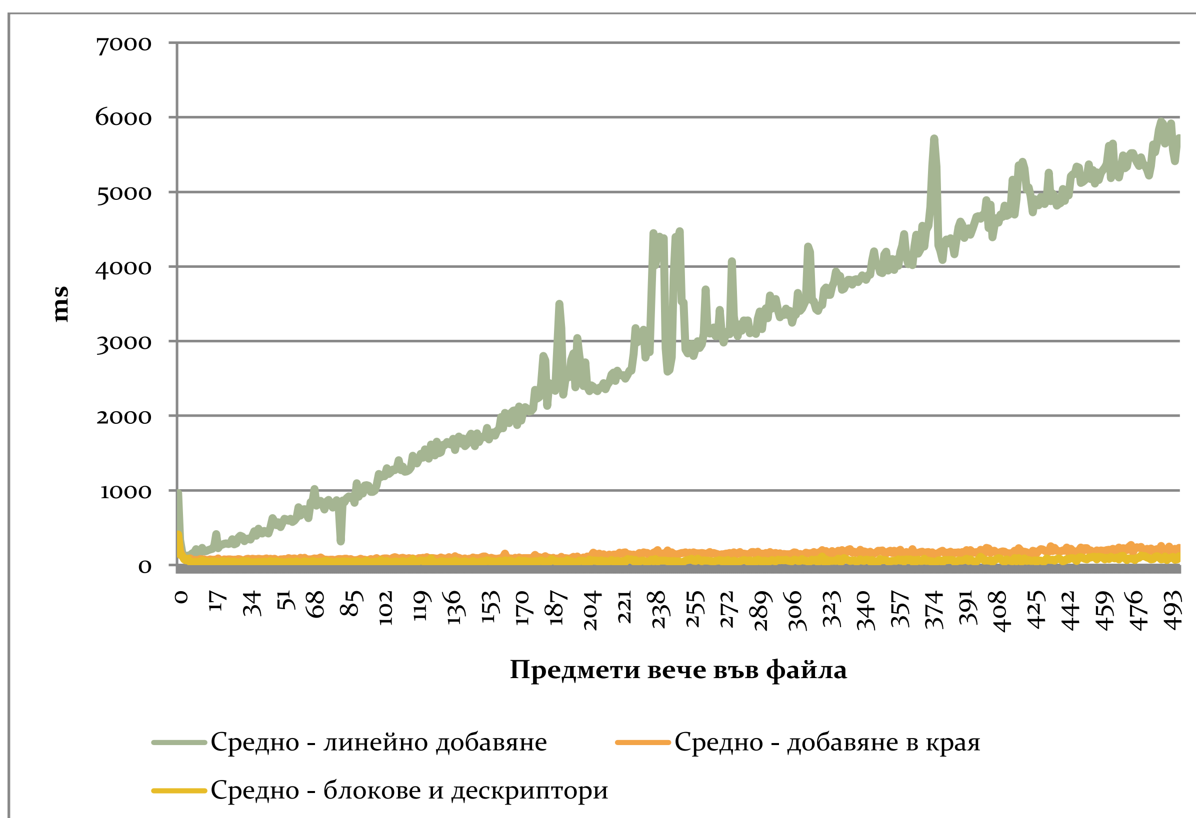
Използването на файл с последователен достъп би могло да предлага бърз достъп за четене (при използване на двоичен файлов формат), но записът или промяната на данни при този подход е изключително бавна операция, особено с нарастване на обема на файла, поради необходимостта от презапис на цялата налична информация наново[2]. Изборът на XML като технология за съхранение на файла, от своя страна, би наложил използването на последователен файл, тъй като няма начин да се променя или изтрива произволно съдържание в XML файл – целият XML файл трябва да бъде генериран наново.

Другият механизъм, който бихме могли да изберем, е използването на файл с непоследователен достъп. Файлът се държи отворен за четене и писане, като данните се записват, четат и изтриват във файла подобно на операциите във файлова система. Това решение предлага бърз достъп както за четене, така и за запис на данни. При запис е необходимо просто да се запише ново копие на данните в някое свободно пространство вътре във фай-

[2] Поради практически съображения, при изтриване или промяна на данни не е възможно просто частта от информацията, следваща променяните данни, да бъде изместена напред или назад вътре във файла, тъй като това би означавало файлът за момент да съществува в неконсистентно състояние, което би било неприемливо. Налага се файлът да бъде генериран наново – дали върху оригиналния файл, или пък в ново копие, с което след това оригиналът да бъде подменен. Отделно, дори и изместването на информация вътре във файла да беше приемливо, постоянното изместване на големи обеми данни при изтриване или промяна на информация би направило тези операции неприемливо бавни.

ла, или пък в края му, а старото копие да бъде маркирано като свободно пространство. При изтриване, по същия начин, е необходимо просто пространството, заето от изтриваните данни, да бъде маркирано като свободно, което би позволило запис на нови данни в него, когато това се наложи.

Фигура 1 илюстрира разликата в производителността на прототипи на различни стратегии, които използвахме, за да проверим коя дава най-добра производителност. На фигурата ясно се вижда, че стратегията, използваща последователен достъп, е много по-бавна, отколкото стратегии, които не генерират целия файл наново при запис на предмет.



Фигура 1: Сравнение на производителността на последователен и непоследователен файл

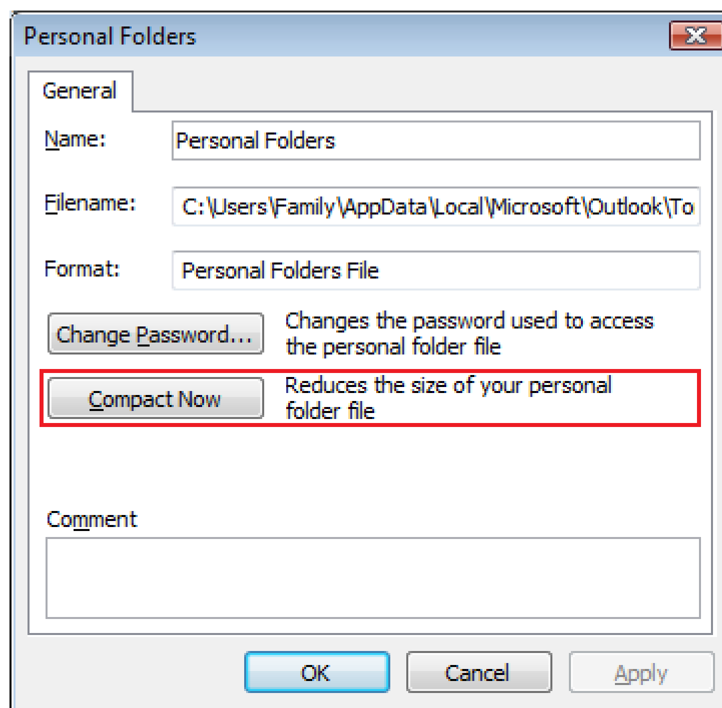
Друго предимство на използването на файл с непоследователен достъп е, че при запис на данни не се изискват толкова системни ресурси, колкото при последователния подход. Тъй като, за да избегне възможността от повреда на файла, последователният вариант, обсъден по-горе, налага генерирането наново

на второ копие на файла, което след това да бъде използвано за подмяна на оригиналното копие, то, за да бъде операцията успешна, е необходимо устройството да има налично достатъчно свободно място за създаването на новото копие на файла, а това означава, че приложение, използващо този подход, може да се възползва от най-много половината от наличното свободно място на устройството. При използване на файл с непоследователен достъп, от друга страна, данните могат просто да бъдат записвани в края на файла (в случай, че вътре във файла не съществува достатъчно голям свободен блок), което позволява много пълноценно използване на системните ресурси.

За съжаление, всички тези предимства на файловете с непоследователен достъп са за сметка на много по-сложната програмна реализация на този подход. Тъй като изтриването и промяната на информация маркират части от файла като свободно пространство, е необходимо да се реализира механизъм, който да помни къде във файла се намират свободните блокове, кое е свободен блок и кое – блок с данни. Заделянето на място за нови данни също се усложнява – ако искаме да се възползваме от вече наличното свободно пространство вътре във файла, вместо новите данни винаги да се записват в края му, трябва да реализираме по-сложни механизми за заделяне на пространството, които да намират подходящ свободен блок вътре във файла, който да бъде използван за записване на новата информация. Освен това, тъй като със записването на нови данни и изтриването на стари такива във файла се появява свободно място, което с времето все повече и повече се фрагментира[3], се налага реализацията на функционалност за „свиване" на файла – операция, при която значимата информация се придвижва напред във файла, а свободното пространство се придвижва и обединява в един го-

[3] Фрагментацията може да бъде намалена чрез използване на подходяща стратегия за заделяне на място (Jensen, 1994).

лям блок в края му, след което файлът се отрязва до реално използваната от данните дължина (тази операция би трябвало да е позната на читателя, тъй като се предлага от множество известни програми, работещи с данни, като например Microsoft® Office Outlook® - виж Фигура 2; някои от приложенията, разгледани в дипломната работа на Милена Йорданова като вече съществуващи решения за съхранение на криптирани данни, също предлагат възможност за свиване на файловете си с данни). Изборът на този вариант налага използването на двоичен файл, тъй като този подход е невъзможно да се реализира чрез текстови файлове и в частност XML.



Фигура 2: Свиване на файла в Microsoft® Office Outlook®

Тъй като планираме да използваме разработвания файлов формат на преносимо устройство с ограничени ресурси – както памет, така и процесорна мощ, бихме искали операциите с данните във файла да бъдат бързи и да пестят системните ресурси. Ето защо, въпреки, че е по-сложен за реализация, за целите на настоящата дипломна работа се спряхме на втория обсъден вариант – използването на двоичен непоследователен файл.

Както обсъдихме по-горе, използването на двоичен непоследователен файл дава възможност за запис на данни – било то съдържание на потребителски данни, мета-данни, инициализационни вектори и т.н. – навсякъде във файла. Това позволява при изтриване или промяна, старото съдържание просто да бъде маркирано като свободно пространство, а новите данни да бъдат записвани, където това е възможно (в свободен блок вътре във файла, или пък в неговия край). За целите на заделянето и освобождаването на пространството във файла няма нужда да знаем дали записваните данни са мета-данни, инициализационен вектор или нещо друго. За всички записвани данни можем да мислим просто като за блокове информация с определена дължина в байтове, които трябва да бъдат записани във файла. Механизмите за заделяне и освобождаване на място, следователно, е необходимо просто да могат да работят с тези общи блокове.

Наличието на блокове с данни на произволно място във файла, обаче, означава, че трябва да съществува някакъв механизъм, който да указва къде се намират тези данни. За целта ще използваме т.нар. „дескриптори". По своята същност дескрипторите представляват просто няколко байта, съдържащи отместването (спрямо началото на файла) на блока, към който дескрипторът сочи, както и дължината му в байтове. В случай, че дължината на сочения блок е предварително известна, както например може да се случи за служебен блок с определена структура, чиято дължина винаги е константна, тогава не е необходимо дескрипторът, сочещ към този блок, да съдържа дължината му; в този случай само отместването на блока е достатъчно (естествено, компонентът, четящ файла, следва да знае, че този дескриптор сочи именно към такъв служебен блок, но, както ще видим по-долу, това не представлява проблем). Ще използваме, освен това, и понятието за празен дескриптор, т.е. дескриптор, съдържащ стойност, указваща, че дескрипторът всъщност не сочи към ни-

какви данни. Това, подобно на NULL стойност в информатиката, позволява лесно да се обозначават краищата на свързани списъци, или пък просто ситуации, в които може да има смисъл даден дескриптор да не сочи към нищо. По този начин вътрешната структура на файла може да бъде оприличена на насочен граф, всеки връх на който е блок с данни, а всяко ребро – дескриптор, сочещ към друг блок с данни, като началото на всяко ребро се намира във върха, чийто блок съдържа дескриптора.

Концепцията за блокове с данни, сочени от дескриптори, позволява да бъдат реализирани наистина бързи операции с данните. Изтриването на данни, например, може да се реализира чрез просто пренасочване на дескриптори (подобно на изключване на елемент от свързан списък, където изключването на елемента става просто чрез пренасочване на указатели). Промяната на блок с данни може също да бъде реализирана по подобен начин – чрез записване на новото съдържание на блока в някое свободно пространство (или в края на файла), следвано от пренасочване на дескрипторите от старата позиция на блока към новозаписаните данни.

Нещо повече – концепцията за дескриптори предлага възможност операциите с файла да бъдат реализирани надеждно, така, че да минимизират възможността от повреда на файла при преждевременното им прекъсване (което би могло да бъде предизвикано по най-различни причини, особено при преносими устройства, където фактори като изпускане на устройството, свършване на заряда на батерията и други подобни не бива да бъдат пренебрегвани). За постигане на тази цел бихме могли да се възползваме от факта, че пренасочването на дескриптор е на практика атомарна операция, тъй като се състои просто от записване на няколко байта във файла. Възможно е, следователно, да проектираме операциите с файла така, че файлът да бъде в

консистентно състояние както преди, така и след пренасочването на дескрипторите. Например при промяна на данни, бихме могли най-напред да записваме новото съдържание на блока в свободно пространство във файла, което на практика не поврежда целостта му. Едва когато данните са сигурно записани, бихме могли чрез пренасочване на дескриптора, сочещ към блока, мигновено да „превключим" към използване на новозаписаното съдържание, изоставяйки старото (на практика превръщайки го в свободен блок).

Както споменахме по-горе, трябва да имаме предвид, че при изоставянето на данни във файла (при пренасочване на дескрипторите, сочещи към тях), във файла се образуват „дупки" свободно пространство. Част от това свободно пространство може да бъде използвано наново, когато във файла трябва да бъде записан нов блок с данни, който е с размер по-малък или равен на размера на някой от свободните блокове. Постепенно, обаче, би могло да се стигне до ситуация, в която във файла има свободни блокове, които са прекалено малки, за да бъде записано нещо в тях. Тези блокове биха били неизползваеми, докато не бъдат консолидирани в по-голям блок (при освобождаване на блок с данни непосредствено преди или след тях, или при изрична дефрагментация на свободното пространство). Отделно, дори и свободните блокове да са достатъчно големи за повторното им използване, те все пак биха заемали излишно място, което може да е необходимо на собственика на преносимото устройство за други цели (за съхранение на данни извън приложението, позволяващо съхранение на лични данни в криптиран вид). Следователно е необходимо да реализираме операция за „свиване" на файла като описаната по-горе, която да бъде изпълнявана автоматично (например при липса на свободно място на устройството), или пък да бъде изрично извиквана от потребителя на приложението.

3.2.2. Заглавен блок

Както обсъдихме по-горе, основната структура на разработвания от нас файлов формат представлява насочен граф, в който блоковете данни са върхове, а дескрипторите – ребра. За да може да работи с графа, обаче, компонентът за съхранение на данните трябва да разполага с отправна точка. Това може лесно да бъде постигнато чрез заглавен блок, намиращ се винаги в самото начало на файла и съдържащ важна информация, описваща файла, както и отправни точки към различните му структури.

Заглавният блок на разработения файлов формат е съвсем прост. Той съдържа единствено следните полета:

- Версия – 4-байтово число, съдържащо версията на файловия формат. За целите на настоящата дипломна работа ще обозначаваме текущия файлов формат с версия 1. Ако в понататъшна разработка се наложи модифицирането на файловия формат така, че той да стане несъвместим с предходните версии, числото, обозначаващо версията, може да бъде увеличено, за да се попречи на предна версия на приложението да отвори файла и да го повреди.
- Дескриптор към масив с допълнителните данни на приложението – съдържа отместването и дължината на блока, описващ съхранените от клиентското приложение допълнителни данни. Структурата на самия блок ще опишем в точка 3.2.3 – „Масив с допълнителни данни на приложението”.
- Дескриптор към първия елемент от списъка на предметите – съдържа отместването на първия елемент от свързания списък, описващ съхранените във файла предмети. Структурата на този списък ще разгледаме по-подробно в точка 3.2.4 – „Списък на предметите”. Дескрипторът не

съдържа дължината на блока, съдържащ сочения елемент, тъй като неговата дължина е винаги константна.

3.2.3. Масив с допълнителни данни на приложението

За да представим допълнителните данни на клиентското приложение, които обсъдихме подробно в точка 3.1.1.3, можем да използваме масив, съдържащ дескриптори към екземплярите на различните типове допълнителни данни, които приложението има нужда да съхранява. Това са именно т.нар. „слотове”, за които говорихме по-горе. Няма смисъл да се използват по-сложни структури като свързани списъци и др. подобни, тъй като приложението, използващо компонента за съхранение на данните, не би трябвало да има повече от няколко типа допълнителни данни. Бихме могли, следователно, да поддържаме масив с необходимите слотове, като просто използваме празни дескриптори за все още непопълнените слотове.

Структурата на блока, описващ въпросния масив, е съвсем проста. Той съдържа:

- Брой на слотовете – 4-байтово число, съдържащо броя на елементите на масива, които следват;
- Последователност от дескриптори, по един за всеки слот. Всеки от тези дескриптори съдържа отместването на блок, описващ съхранените допълнителни данни на приложението, или пък е празен дескриптор, показвайки, че в съответния слот няма записани данни. Дължината на сочения блок не се указва изрично, тъй като става дума за служебни блокове с предварително известна константна дължина.

Блоковете, сочени от дескрипторите на масива, от своя страна, съдържат информация, позволяваща да бъде намерено съ-

държанието на съхранените допълнителни данни. Всеки такъв блок съдържа два дескриптора:

- Дескриптор към съхранените допълнителни данни, подадени от приложението (криптирани или не, в зависимост от нуждите на приложението, както обсъдихме подробно в точка 3.1.1.3).
- Дескриптор към инициализационния вектор, в случай, че съхранените допълнителни данни са криптирани. Ако данните не са криптирани, дескрипторът към инициализационния вектор е празен.

Въпреки, че на пръв поглед може да изглежда странно, че отделяме в собствен блок двата дескриптора, описващи допълнителните данни, вместо просто да ги запишем в елементите на масива, всичко става ясно когато помислим за операцията по съхранение на данни в слот. Ако двата дескриптора се намираха в елементите на масива, не бихме могли да пренасочим и двата дескриптора атомарно – съществува вариант дескрипторът, сочещ към новите съхранени допълнителни данни, да бъде съхранен, но операцията да бъде внезапно прекъсната преди дескрипторът, сочещ към съответния инициализационен вектор, да бъде променен, като по този начин файлът бъде повреден. Чрез отделянето на двата дескриптора в собствен блок, от друга страна, постигаме така желаната атомарност – бихме могли да съхраним данните и инициализационния вектор, да съхраним и блока, съдържащ двата дескриптора, сочещи към тях, и когато се уверим, че всичко е наред, просто да пренасочим единствения дескриптор за съответния слот към новозаписания блок, по този начин едновременно превключвайки към използване на новите съхранени данни и инициализационен вектор.

3.2.4. Списък на предметите

Въпреки, че, както споменахме в точка 2.1 – „Съхранение на криптирани данни", броят на съхранените предмети във файла не би трябвало да е твърде голям и в повечето случаи няма да надвишава няколкостотин, те все пак са достатъчно много, за да бъде описанието им с масив непрактично, особено като се има предвид, че предмети могат да бъдат добавяни и изтривани сравнително често. Най-подходящо за представянето на предметите е използването на свързан списък, защото по този начин добавянето и изтриването на предмет могат да бъдат реализирани чрез просто включване или изключване от списъка чрез пренасочване на дескриптор.

Свързаният списък на предметите се състои от множество блокове, по един за всеки елемент. Всеки от тези блокове съдържа:

- Дескриптор към блока на следващия елемент. Съдържа само отместването на блока; дължината не е необходима, тъй като блоковете, съдържащи елементи на списъка, имат предварително известна константна дължина;
- Идентификатор на описвания предмет (GUID, представен като поредица от байтове);
- Четири дескриптора, сочещи към информацията на предмета:
    - Дескриптор към блока, съдържащ криптираните мета-данни за предмета;
    - Дескриптор към блока, съдържащ инициализационния вектор на криптираните мета-данни;
    - Дескриптор към блока, съдържащ криптираните данни на предмета;

- Дескриптор към блока, съдържащ инициализационния вектор на криптираните данни.

Използването на гореописаната структура позволява изтриването на предмет да бъде изключително бърза и надеждна операция – изтриването се състои в простото изключване на предмета от списъка чрез пренасочването на един-единствен дескриптор.

Трябва да отбележим, че промяната на данните или мета-данните на предмет изискват подмяната на целия елемент за съответния предмет в списъка. По този начин се избягва потенциалното разминаване на съдържанието на данните или мета-данните със съответните им инициализационни вектори, или пък на самите данни със съответните им мета-данни, в случай, че операцията бъде прекъсната преждевременно. Чрез записването на нов блок за променения елемент (част от дескрипторите на който не е проблем да сочат към вече съществуващите данни, в случай, че не биват променяни, като например ако променяме само мета-данните за предмета) се осигурява атомарността на операцията, тъй като подмяната на стария с новозаписания елемент в свързания списък става чрез пренасочването на един-единствен дескриптор.

3.2.5. Външни файлове

Макар, че за компонента за съхранение на данните не би трябвало да е от значение какви данни му се подават от клиентското приложение и от къде те идват, не можем да не отчетем факта, че данните, които трябва да бъдат съхранявани, могат да бъдат разделени на два класа: данни, въведени на ръка от потребителя в потребителския интерфейс, и данни, идващи от съдържанието на файл (например файл или картинка, които потребителят желае да съхрани криптирани). Първият клас данни обик-

новено са малки по обем, докато представителите на втория клас биха могли да бъдат много големи.

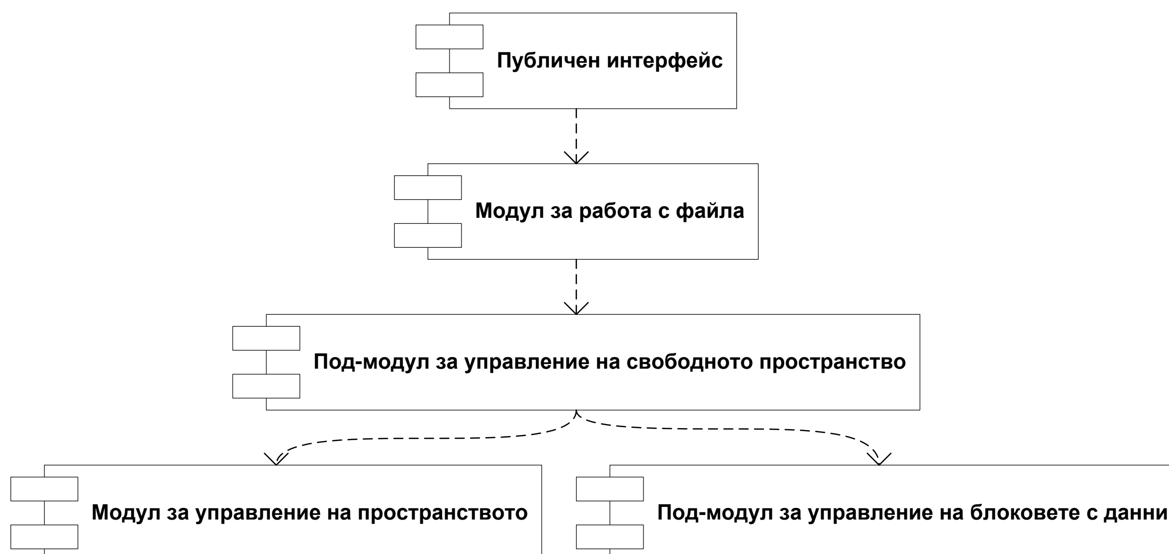
Файловият формат, който разработихме, работи добре с малки по обем данни. Изтриването на данни оставя сравнително малки „дупки” свободни пространства във файла, които могат да бъдат запълнени от други данни. Ако ли пък се наложи свиване на файла, то може да бъде осъществено чрез преместване на сравнително малки обеми от данни (тъй като свиването може да работи с преместване само на цели блокове). Големите данни, от друга страна, представляват проблем, тъй като наличието им означава, че по време на свиване на файла ще се налага придвижването на големи обеми от данни; изтриването им пък би предизвикало достатъчно голяма част от пространството на устройството да остане прахосана, докато не бъде изпълнена операция по свиване на файла.

Горните проблеми се решават чрез използването на хибриден модел за съхранение на данните. Приложението, използващо компонента за съхранение на данните, би могло да го уведоми дали данните, предназначени за съхранение, са въведени през потребителския интерфейс или идват от съдържанието на файл. Ако данните идват от потребителския интерфейс, компонентът ще ги съхрани в общия файл. Ако ли пък идват от съдържанието на файл, то тяхното съдържание бива съхранено в отделен файл извън основния файл с данни. По този начин оставяме файловата система на устройството да има грижата за съхранението, изтриването и намирането на по-големите предмети, които, в повечето случаи, са достатъчно малко, за да не се сблъскаме с намалената ѝ производителност, обсъдена в точка 2.3 – „Механизъм за съхранение на данните”.

Външните файлове можем да съхраняваме в отделна папка в директорията, съдържаща основния файл с данни, като за имена

на файлове използваме идентификаторите на предметите (GUID във вид на символен низ). В тях е необходимо да съхраняваме само данните на съответните предмети. Мета-данните, както и инициализационните вектори на данните и на мета-данните могат да продължат да се съхраняват в основния файл с данни, тъй като са достатъчно малки. По този начин е необходимо единствено в дескриптора, който би трябвало да сочи към данните на предмета в основния файл, да използваме специална запазена стойност, показваща, че данните на предмета могат да бъдат намерени във външен файл. Всичко останало в основния файл с данни остава непроменено.

3.3. Архитектура на компонента



Фигура 3: Обща архитектура на компонента

На Фигура 3 е илюстрирана общата архитектура на разработвания компонент за съхранение на данните. В следващите подточки ще разгледаме как стигнахме до нея, ще опишем ролята на различните модули и ще покажем в детайли как те си взаимодействат за постигане на общата цел – надеждно съхранение на криптирани данни.

3.3.1. Комуникация чрез потоци

Едно от първите решения, които трябва да бъде взето при разработката на компонент за съхранение на данните, е по какъв начин ще се осъществява предаването на данни между компонента и използващото го приложение, както и вътре в самия компонент.

Тъй като компонентът може да разглежда данните, които трябва да бъдат записани, като блокове от байтове, първото решение, което идва наум, е да се комуникира чрез указатели към масиви от байтове. При този вариант приложението, подготвящо данните за запис, трябва да ги преобразува в масив от байтове, който да подаде на компонента за съхранение, за да ги запише той във файла. При четене компонентът следва да прочете данните като масив от байтове, който да върне на приложението.

При разглеждане на този вариант, обаче, бързо се вижда, че съхранението в паметта на масиви от байтове би могло да бъде проблематично на устройство със силно ограничена памет, каквото е едно Pocket PC. Докато данни, въвеждани от потребителя през потребителския интерфейс на клиентското приложение, би трябвало да са достатъчно малки, за да можем да ги представим в паметта както си поискаме, то съдържанието на голям файл или картинка, чийто обем надвишава количеството на свободната оперативна памет на устройството, не може да бъде обработвано по този начин.

Проблемът с ограничената оперативна памет може да бъде заобиколен елегантно чрез използването на потоци. Представянето чрез поток на данни, въведени от потребителя през потребителския интерфейс на приложението, заема в паметта не повече място, отколкото би заело представянето им чрез масив от байтове. Данни, идващи от файл, обаче, могат да бъдат представени без заемането на почти никаква памет, тъй като за целта

може да се използва директно потокът, опериращ с данните в самия файл.

Използването на потоци за комуникация не би затруднило приложението, използващо компонента за съхранение на данните, при криптирането или декриптирането на информация. Даже напротив – тъй като криптографската функционалност, предлагана от .NET Compact Framework, поддържа работа с потоци, като е възможно поток от данни просто да бъде „обхванат" от криптиращ или декриптиращ поток, който просто да криптира или декриптира данните в процеса на тяхното четене или запис, то представянето на данните чрез потоци е всъщност подходящо и практично. Самият компонент за съхранение на данните пък лесно може да копира съдържанието на потоците, идващи от приложението, просто чрез използването на малък временен буфер (чрез многократно запълване на буфера от входния поток и изпразването му в изходния).

3.3.2. Дескриптори

3.3.2.1. Общи и служебни блокове данни

Както показахме в точка 3.2 – „Файлов формат", файлът се състои от блокове с данни, сочени от дескриптори, съдържащи отместването и, ако е необходимо, дължината на сочения блок. Блоковете, обаче, въпреки, че си приличат, не са еквивалентни, тъй като съхраняват различни типове данни. Бихме могли, следователно, да правим разлика между различните дескриптори в зависимост от това към какъв тип блок сочат. Това би било полезно, когато трябва да определим, например, дали даден дескриптор е необходимо да записва дължината на блока, към който сочи, както и за да можем правилно да интерпретираме данните, сочени от него.

Блоковете могат да се разделят на два различни типа. Първият тип са блокове с общо предназначение, съхраняващи байтови данни без някакъв конкретен формат. Това са блоковете, съхраняващи данни, идващи от клиентското приложение – данни, мета-данни, допълнителни данни, инициализационни вектори и т.н.

Вторият тип са блокове със специално служебно съдържание, които имат строго определена структура. Такива са например блоковете, съдържащи елементи от списъка с предмети, блокът, съдържащ масива със слотове за допълнителните данни, съхранени от приложението, както и блоковете, съдържащи описанието на самите слотове.

3.3.2.2. Представяне на дескрипторите като обекти

Тъй като компонентът за съхранение на данните не може да прочете всички данни от файла в паметта при неговото отваряне, необходимо е по време на работа с файла да се помни къде в него се намират различните блокове с данни, за да могат да бъдат прочетени при нужда. Добре е това да става чрез специален тип данни, който, подобно на указател, да сочи към данните във файла; този тип данни може просто да бъде използван в по-сложните структури от данни, представящи в паметта структурата на файла.

Друг важен проблем свързан с дескрипторите, на който трябва да обърнем внимание, е тяхното пренасочване. Различните операции с данните, като например запис на предмет, промяна, изтриване и т.н., изискват пренасочването на дескриптори, които вече съществуват във файла, към нови блокове с данни. Това означава, че компонентът за съхранение на данните трябва да знае къде във файла се намират самите дескриптори, за да може тяхната нова стойност да бъде правилно съхранена на подходящото отместване.

Горните два проблема се решават елегантно чрез представянето на дескрипторите като обекти в паметта. Всеки от тези обекти може да помни къде във файла се намират сочените от дескриптора данни, така, че да могат да бъдат прочетени, когато бъдат поискани. Обектът, също така, може да помни къде във файла се намира самият дескриптор, който той представя. При пренасочване на дескриптора обектът може да запише новата му стойност на правилното отместване във файла, като по този начин постигаме както правилно опресняване на файла с данните, така и наличието на текущи стойности за всички дескриптори в паметта.

3.3.2.3. Йерархия от дескрипторни класове

Както описахме в точка 3.3.2.1 по-горе, блоковете с данни във файла са от различни типове. Представянето на дескрипторите като обекти позволява създаването на йерархия от класове, в която имаме различни типове дескриптори в зависимост от това към какъв тип блокове сочат. Така е възможно да се реализира различно поведение на различните типове дескриптори (например дескриптори, сочещи към служебни блокове данни с предварително известна дължина, да нямат нужда да записват дължината на блока във файла).

В основата на йерархията от класове стои абстрактен базов клас, предоставящ общата функционалност, която всички дескриптори трябва да притежават (или от която всички наследници на класа могат да се възползват). Тази функционалност включва помнене на местоположението на блока с данни и на самия дескриптор във файла; абстрактен метод за записване на стойността на дескриптора във файловия поток, който да бъде предефиниран от конкретните дескрипторни класове; също така методи, помагащи на наследените класове да записват и четат отместването и дължината на сочения блок данни и които в процеса на

запис/четене на тези стойности автоматично запомнят къде във файла се намира току-що прочетеният/записан дескриптор, така че наследените класове да няма нужда да се грижат за това; и т.н.

Наследените класове, от своя страна, предефинират методите за прочитане и запис на самия дескриптор във файла, като по този начин могат да вземат решението дали имат нужда да запишат дължината на сочения блок или не. Наследените класове, също така, могат да предлагат методи за прочитане или запис на конкретния тип блок, към който сочат (например дескриптор, сочещ към блок с елемент от списъка с предмети, би могъл да предлагат методи за записването на такъв елемент във файла, или пък за прочитането му).

3.3.2.4. Правилна работа с дескрипторите

За да се подсигурим срещу случайното присвояване на произволни стойности на дескрипторите, можем да се възползваме от факта, че те са представени като обекти в паметта. Тъй като модифицирането на дескриптор може да стане само чрез методите на представящия го клас, чрез правилен подбор на възможните операции за присвояване можем силно да намалим вероятността за неправилна работа с дескрипторите.

Идеалният вариант е не само всички методи за присвояване на стойност на дескриптор да бъдат концентрирани в базовия клас, но и да не съществува начин за присвояването на произволна стойност на дескриптор. За целта може да се предлагат само следните два начина дескриптор да получи стойността си:

1. Метод за заделяне на блок, към който дескрипторът да сочи, по зададена дължина. Дескрипторният клас сам вика модула за управление на свободното място (който ще опишем в точка 3.3.4 – „Управление на пространството”), за да задели място за блока, към който след това автоматично се насочва;

2. Метод, чрез който дескрипторът може да си присвои стойността на друг дескриптор, ефективно насочвайки се към същия блок. Тази операция е валидна само, когато типовете на дескрипторите съвпадат.

Така описаните операции гарантират, че всички дескриптори могат да сочат единствено към блокове, заделени от модула за управление на свободното пространство. Отделно, разбира се, дескрипторът трябва да предлага и метод, чрез който да бъде уведомен, когато следва да се пренасочи поради преместване на сочения от него блок по време на операция по свиване на файла.

3.3.3. Представяне на файловите примитиви

Подобно на дескрипторите, останалите основни структури, използвани във файловия формат, също могат да бъдат представени програмно чрез класове. Тези класове биха имали знанието как да прочетат съответната структура от файла и как да я запишат, както и да предоставят методи за лесна работа със съответната структура, скривайки детайлите на тяхното взаимодействие с файла.

Нека разгледаме как това може да стане за всяка от тях.

3.3.3.1. Заглавен блок

Както описахме в точка 3.2.2 – „Заглавен блок", заглавният блок на файла съдържа версията на файла, както и два дескриптора – дескриптор, сочещ към блок с масив, съдържащ допълнителните данни на приложението, и дескриптор, сочещ към първия елемент от свързания списък на съхранените във файла предмети.

Класът, представящ заглавния блок, не е необходимо да предоставя достъп до версията, тъй като никой друг освен него няма нужда да я знае; тя служи единствено за проверка по време на прочитане на заглавния блок, за да бъде сравнена със списък на

поддържаните версии. Достъп до двата дескриптора, обаче, може да бъде предоставен от класа на заглавния блок чрез публични свойства, даващи достъп за четене до обектите, представящи двата дескриптора в паметта.

Заслужава си да обърнем внимание върху факта, че достъпът за четене означава, че дескрипторните обекти не могат да бъдат подменени с други дескрипторни обекти, тъй като достъпът за четене ограничава възможността за писане в указателите (reference) към тези обекти. Той обаче по никакъв начин не пречи на модификацията на получените чрез него обекти, т.е. кодът, осъществяващ достъп до дескрипторите, има възможност да използва методите им, за да ги насочи към нови блокове. По този начин, чрез достъп за четене, се осигурява правилната работа с дескрипторите, тъй като никой няма възможност да подмени представянето им в паметта, без да предизвика запис на новата стойност във файла.

3.3.3.2. Предмет

Подобно на заглавния блок, класът, представящ предмет в паметта, трябва да съхранява дескрипторни обекти за всеки от четирите блока данни, представящи един предмет – криптирани мета-данни, инициализационен вектор на мета-данните, криптирани данни, както и инициализационен вектор на данните. Тук можем да направим стъпка напред, обаче, като скрием факта, че обектът на предмета съхранява дескриптори, и вместо това капсулираме процеса на четене на самите данни в него. Следователно, предметът би могъл да предоставя две свойства за четене – първото прочитащо от дескрипторите и връщащо мета-данните на предмета заедно с техния инициализационен вектор, а второто – данните на предмета, също заедно с техния инициализационен вектор.

3.3.3.3. Елемент на свързания списък на предметите

Елементите на свързания списък на предметите, описан в точка 3.2.4 – „Списък на предметите”, също както заглавния блок и предметите по-горе, могат да бъдат представени в паметта като обекти, притежаващи функционалност за своето четене и запис от файла и предоставящи лесен достъп до своите данни.

Както обсъдихме в точка 3.2.4, елементите на свързания списък се състоят от дескриптор към следващия елемент, идентификатор на описвания предмет, както и от самия предмет (във вид на дескриптори към мета-данните, данните, както и техните инициализационни вектори; това, обаче, е същата структура, чието представяне като обект описахме в точка 3.3.3.2 – „Предмет”). Класът, представящ елемент от свързания списък, следователно, би могъл да предлага свойства за четене, даващи достъп до обекта, представящ дескриптора към следващия елемент от списъка, до идентификатора на предмета, както и до обекта, представящ самия предмет[4].

С цел лесна навигация по свързания списък в паметта, класът на елемента също така може да притежава указатели към предишния и към следващия елемент в списъка. По този начин в паметта ще разполагаме с двойно-свързан списък от обекти, представящи елементите на свързания списък във файла, което ще спомогне за по-лесната манипулация на дескрипторите при промяна на списъка (например при включване, подмяна или изтриване на елемент).

[4] Обектът, представящ самия предмет, не може да бъде четен при поискване, а трябва да бъде държан в паметта по време на цялата продължителност на живота на обекта на разглеждания елемент от свързания списък. Това се налага, тъй като дескрипторите, намиращи се в обекта на предмета, трябва да бъдат налични в паметта на устройството, за да се знае, че блоковете, към които сочат във файла, са заети, а не са просто свободно пространство. Точният механизъм за отчитане на свободното и заетото пространство чрез дескрипторите ще обсъдим в точка 3.3.4 – „Управление на пространството”.

3.3.3.4. Списък на предметите

Сложността на цялата структура на списъка на предметите можем да скрием в отделен клас, който освен това може да служи като колекция на всички предмети, съхранени в компонента за съхранение на данните. Този клас може да предлага методи за включване, подмяна и изключване на предмет от списъка с предмети, както и за получаване на списък на всички съхранени предмети.

За бърз достъп до предметите, класът трябва да използва подходяща ускоряваща структура, която да му позволява да намира конкретният елемент от списъка, описващ даден предмет по зададен идентификатор на предмета. Това е необходимо, за да не се обхожда двойно-свързаният списък всеки път, когато трябва да бъде извлечен или променен даден елемент. Двойно-свързаната структура на списъка може да бъде използвана за бързо намиране на предходния и следващия елемент спрямо елемента, който бива обработван (например за пренасочване на дескриптора на предходния елемент, или за корекция на указателя към предходния елемент на следващия елемент при изключване на елемент от списъка), но за достигане до самия търсен елемент следва да се използва ускоряващата структура.

	SortedList	SortedDictionary	Hashtable
Достъпен в .NET CF	Да	Не	Да
Използвана памет	$O(n)$	$O(n)$	$\approx O(n)$
Добавяне на елемент	$O(n)$	$O(\log n)$	$\approx O(1)$
Изтриване на елемент	$O(n)$	$O(\log n)$	$\approx O(1)$
Достъп до елемент по ID	$O(\log n)$	$O(\log n)$	$\approx O(1)$

Таблица 1: Сравнение на наличните структури от данни

След сравнение на различни видове структури и техните недостатъци (виж Таблица 1) се спряхме на използването на хеш таблица за ускоряване на достъпа до елементите на списъка.

3.3.3.5. *Допълнителни данни*

Също както представихме предметите като обекти в точка 3.3.3.2, можем да представим и отделните единици съхранени допълнителни данни на приложението чрез отделен клас. В него трябва единствено да се съхраняват двата обекта, представящи дескрипторите към съдържанието и, в случай, че то е криптирано, към инициализационния му вектор. Отново, също както при предметите, бихме могли да капсулираме четенето на данните в този клас, като обектът трябва единствено да връща вече прочетеното съдържание (и инициализационен вектор, ако е криптирано).

3.3.3.6. *Елемент на масива с допълнителните данни*

Подобно на отделните елементи от свързания списък с предметите, елементите на масива с допълнителните данни (т.е. отделните слотове за различните типове допълнителни данни) също могат да бъдат представени като отделни обекти, съдържащи дескрипторен обект, сочещ към допълнителните данни, и предлагащи свойство, връщащо директно обект на класа на допълнителните данни, който описахме в предната точка.

Също както отбелязахме с бележка под линия в точка 3.3.3.3, необходимо е обектите, представящи допълнителните данни, да бъдат съхранявани перманентно от елементите на масива, вместо да бъдат четени при поискване. Това е необходимо, тъй като дескрипторите, намиращи се в тях, трябва да бъдат налични в паметта на устройството, за да се знае, че блоковете, към които сочат във файла, са заети, а не са просто свободно пространство. Механизмът за отчитане на свободното и заетото пространство чрез дескрипторите ще обсъдим подробно в точка 3.3.4 – „Управление на пространството".

3.3.3.7. Масив с допълнителните данни

Елементите на масива с допълнителните данни можем изцяло да скрием чрез представянето на целия масив като единединствен обект-колекция, предоставящ операции за извличане и съхранение на допълнителни данни по номер на слот. Тези операции работят директно с обекти, представящи допълнителните данни, като по този начин работата с поддържането на масива и неговите елементи в структурата на файла се скрива изцяло от потребителя на класа.

3.3.4. Управление на пространството

Както дискутирахме в точка 3.2 – „Файлов формат", използването на файлов формат с непоследователен достъп означава, че трябва да се полагат допълнителни грижи за управление на пространството във файла. Необходимо е да се помни, например, кои части на файла са заети с данни и кои са свободни пространства. Трябва да се използват подходящи алгоритми за заделяне на пространството, които да позволяват повторно използване на свободното място вътре във файла, вместо да го оставят неизползвано. Това обаче означава, че трябва да се поддържа списък със свободните блокове във файла, за да е възможно алгоритъмът за заделяне на място да намери подходящ свободен блок, който да използва за съхранение на данните. Нуждата от операция за свиване на файла, обсъдена в точка 3.2 – „Файлов формат", от друга страна, означава, че не е достатъчно само да се помни дали дадено пространство във файла е заето; вместо това е необходимо да се поддържа списък с всички отделни блокове с данни, за да може те да бъдат премествани индивидуално по време на свиването.

Поддържането на дескрипторите от файла като обекти в паметта предоставя интересна възможност за следене на заетите и свободни блокове във файла. Зареждането на всички файлови

примитиви в паметта, а следователно – наличието на всички дескриптори в паметта, всеки от които сочи към индивидуален блок с данни във файла, означава, че, стига да поискаме, бихме имали списък на всички блокове с данни във файла, а от там – да третираме всяко пространство, което не е сочено от дескриптор, като свободно[5]. За целта е необходимо всеки дескриптор, който бива насочван към блок във файла, да бъде регистриран, за да се следи блокът, към който сочи, за зает. При унищожаване или пренасочване на дескриптор, регистрацията му следва да бъде премахната (и евентуално подновена с новото му съдържание, в случай на пренасочване), за да бъде присъединен блокът му към наличното свободно пространство.

Регистрацията на дескрипторите предоставя, също така, възможността при преместване на блок по време на свиване на файла, дескрипторите, сочещи към блока, да бъдат пренасочени към новото му отместване, а обектите на дескрипторите, намиращи се вътре в блока – опреснени с новото им местоположение във файла.

Цялата тази логика може да бъде концентрирана в модул за управление на пространството, който може да бъде разделен на две части – под-модул за управление на свободното пространство и под-модул за управление на блоковете с данни.

3.3.4.1. Модул за управление на пространството

Модулът за управление на пространството капсулира описаната по-горе функционалност по регистрация на дескрипторите и по управление на свободното и заетото пространство. Той предлага методи за:

- Заделяне на блок по зададена дължина;

[5] Заглавният блок на файла е единственият блок, който не е сочен от някакъв дескриптор. Лесно бихме могли да осигурим непокътнатостта му, обаче, като просто го изключим от списъка със свободното пространство. Така той нито ще бъде преместен при свиване на файла, нито ще бъде презаписан от друг блок.

- Освобождаване на зает блок;
- Маркиране на блок като зает (използва се по време на начално зареждане на файла с данни);
- Регистрация на дескриптор и маркиране на пространството, сочено от него, като заето;
- Премахване на регистрацията на дескриптор (маркира пространството, сочено от него, като свободно, в случай, че той е последният регистриран дескриптор, сочещ към него);
- Свиване на файла (дефрагментира свободното пространство в края на файла, след което отрязва файла до реално използваната от него дължина).

Както описахме по-горе, за реализацията на тази функционалност, модулът за управление на пространството ще разчита на два под-модула – под-модул за управление на свободното пространство и под-модул за управление на блоковете с данни.

3.3.4.2. Под-модул за управление на свободното пространство

Под-модулът за управление на свободното пространство има грижата да поддържа списък на свободното пространство във файла, както и да позволява при нужда да бъде намерен подходящ по големина свободен блок за съхранение на нови данни. Той трябва да предлага методи за:

- Заделяне на блок по зададена дължина чрез използване на подходящ алгоритъм за заделяне на свободното пространство;
- Връщане на блок в списъка на свободното пространство;

- Изключване на блок от списъка на свободното пространство (използва се по време на начално зареждане на файла с данни);
- Намиране на свободни блокове по различни критерии (използват се от операция по свиване на файла за намиране на подходящи блокове за запълване);
- Намиране на общото количество свободно място във файла.

3.3.4.3. Под-модул за управление на блоковете с данни

Под-модулът за управление на блоковете с данни е отговорен за поддържане на списък на дескрипторите и заетите блокове във файла. Предлага и необходима функционалност по управлението на блоковете по време на операция по свиване на файла. Предлаганите от него методи позволяват:

- Регистрация на дескриптор;
- Премахване на регистрацията на дескриптор;
- Намиране на блокове с данни по различни критерии (използват се от операция по свиване на файла за намиране на подходящи блокове за преместване);
- Преместване на блок с данни от едно отместване на друго (използва се по време на операция по свиване на файла). Тази операция копира данните от блока на новото отместване, уведомявайки всички дескриптори, намиращи се в него, за новото им отместване във файла, след което пренасочва всички дескриптори, сочещи към преместванния блок, към новото отместване.

3.3.5. Модул за работа с файла

Модулът за работа с файла предлага една-единствена отправна точка, която публичният интерфейс, както и самите при-

митивни обекти, могат да използват за работа с файла. Той има грижата за прочитането на основните структури от данни като заглавния блок, списъка на предметите и т.н., които описахме в точка 3.3.3 – „Представяне на файловите примитиви", като същевременно скрива сложността на файла с данните и предлага прост механизъм за работа с него на използващия го публичен интерфейс.

Модулът за работа с файла предлага на публичния интерфейс достъп до обектите, представящи основните файлови примитиви като заглавния блок на файла, колекцията със съхранени предмети, както и колекцията с допълнителни данни във файла. Предлага се също така и метод за свиване на файла.

3.3.6. Публичен интерфейс

Публичният интерфейс на компонента за съхранение на данните представлява просто клас, предлагащ публични методи за извършване на операциите, анализирани подробно в точка 3.1.2 – „Операции". Създаването на повече от един обект от този клас би позволило на приложението да работи с повече от един файл едновременно, в случай, че то има такава нужда.

Зад кулисите, публичният интерфейс използва модула за работа с файла и обектите, представящи различните файлови примитиви, за да извършва операциите с данните, изисквани от използващото го приложение.

3.3.7. Взаимодействие между модулите при операции с блокове

Да разгледаме как си взаимодействат модулите при изпълнението на операции, свързани с единични блокове данни. Операциите са така проектирани, че файлът с данни да бъде запазен в консистентно състояние през всички стадии на операцията с цел избягване на възможността за неговата повреда.

3.3.7.1. Запис на блок с данни

Записът на блок данни се извършва в следната последователност:

1. Създава се дескриптор от съответния тип на записвания блок;
2. Данните се предават на метода на дескриптора, позволяващ запис на съответния тип данни във файла;
3. Дескрипторът се обръща към модула за управление на пространството, за да задели място за съхранението на данните;
4. Дескрипторът си отбелязва местоположението на заделения блок, след което се регистрира в модула за управление на пространството;
5. Дескрипторът записва данните в заделения блок; ако дескрипторът е от тип, който сочи към по-сложни служебни примитиви, извиква за целта метода на представящия примитива обект, който знае как да запише примитива в заделения блок;
6. Сега кодът, записващ данните, разполага с незаписан във файла дескриптор, сочещ към съхранени данни. Има два варианта за съхранението на този дескриптор във файла:
   a. Друг дескриптор, който вече има представяне във файла, може да бъде пренасочен към новото съдържание, както описахме в точка 3.3.2.4 – „Правилна работа с дескрипторите"; оригиналният дескриптор, използван за запис на информацията, се разрегистрира и унищожава, тъй като вече не е нужен;
   b. Дескрипторът се съхранява във файла като част от по-голяма структура, съхранена на свой ред чрез описваната в момента процедура.

3.3.7.2. Четене на блок с данни

Четенето на блок с данни е изключително просто. То става по следния начин:

1. Извиква се метода на дескриптора, отговарящ за прочитане на блока му с данни;
2. Дескрипторът прочита блока с данни. Ако дескрипторът е от специализиран тип, сочещ към по-сложни служебни примитиви, за прочитането той създава обект на съответния примитив, като му предава управлението, за да може той да се прочете сам от файла;
3. Дескрипторният обект връща прочетените данни (обект на прочетения примитив) обратно на кода, поискал тяхното прочитане.

3.3.7.3. Освобождаване на блок с данни

Освобождаването на блок с данни не е процедура само по себе си, тъй като се осъществява в резултат от пренасочването на дескриптор от даден блок с данни към друг, или към запазената стойност за празен дескриптор. Тук все пак ще обърнем специално внимание как става това с цел да си представим по-ясно процеса по освобождаването на блок.

1. Дескриптор бива пренасочен или унищожен;
2. Регистрацията на дескриптора се премахва от модула за управление на пространството. В случай на пренасочване, това се извършва от метода за пренасочване на дескриптора, преди да промени стойността му и да го регистрира наново с новата му стойност. В случай на унищожаване, това се извършва от метода за неговото унищожение (Dispose);

3. Модулът за управление на пространството премахва регистрацията на дескриптора от под-модула за управление на блоковете с данни;
4. Ако под-модулът за управление на блоковете с данни показва, че това е бил последният дескриптор, сочещ към съответния блок, модулът за управление на пространството предава управлението на модула за управление на свободното пространство, за да маркира блока като свободен.

3.3.7.4. Свиване на файла

Целта на операцията по свиване на файла е да придвижи блоковете с данни напред, а свободните блокове – назад, за да може, в края на процеса, файлът да бъде отрязан до реално използваното от него място.

По какъв точно начин се избира кои блокове да бъдат преместени и къде, е детайл на реализацията. Тук ще разгледаме общата само общата процедура, за да видим как различните компоненти си взаимодействат за постигане на общата цел. Ето как става това:

1. Модулът за управление на пространството получава команда за свиване на файла;
2. Ако под-модулът за управление на свободното пространство покаже, че такова няма, операцията приключва, тъй като файлът е вече в своя окончателен вид;
3. Ако под-модулът за управление на свободното пространство покаже, че всичкото свободно пространство е вече в края на файла, се преминава на стъпката по отрязването на файла;
4. Модулът за управление на пространството използва функционалността, предложена му от под-модулите за управление на свободното пространство и за управление на

блоковете с данни, за да намери подходящ свободен блок за запълване, както и блок с данни, който да бъде преместен в този свободен блок. При нужда за дестинация може да бъде избран краят на файла, в случай, че свободните пространства вътре във файла са прекалено малки;

5. Модулът за управление на данните предава управлението на модула за управление на блоковете с данни, за да премести избрания блок данни в избраното свободно пространство;
6. За целта, модулът за управление на блоковете с данни вика някой от дескрипторите, сочещ към блока, да премести съдържанието му;
7. Дескрипторът копира съдържанието на блока, премахва регистрацията си (което маркира оригиналният блок като свободен в модула за управление на свободното пространство), пренасочва се към новото отместване на блока, след което се регистрира наново (като по този начин новият блок се появява в списъка на блоковете с данни);
8. Модулът за управление на блоковете с данни пренасочва и останалите дескриптори, сочещи към оригиналния блок, към новата му позиция, чрез присвояването им на съдържанието на оригиналния дескриптор, използван за преместването на блока[6];
9. Модулът за управление на блоковете уведомява всички дескриптори, намиращи се вътре в премествания блок, че

[6] По време на преместване на блока и пренасочване на дескрипторите файлът се запазва в консистентно състояние, тъй като структурата му е такава, че всеки блок с данни бива сочен само от един-единствен дескриптор във файла. Наличието на повече от един дескриптори, сочещи към даден блок, е възможно в паметта, тъй като при запис на данни се използват несъхранени във файла дескриптори, чието съдържание впоследствие се присвоява на истински такива, намиращи се във файла; в самия файл обаче винаги имаме най-много по един дескриптор, сочещ към даден блок.

блокът им е преместен, за да знаят те новите си отмествания във файла от тук нататък;

10. Блокът е преместен; процедурата продължава отново от стъпка 3;
11. Когато на стъпка 3 модулът за управление на свободното пространство покаже, че всичкото свободно пространство е в края на файла, файлът се отрязва до реално използваната от него дължина.

3.3.8. Взаимодействие между модулите при сложни операции

Както при примитивните операции, сложните операции с различните структури във файла са така проектирани, че да го запазват в консистентно състояние във всяка фаза на изпълнението си. Различават се единствено по мащаба на своето действие – докато примитивните операции обхващат работа с единични блокове, сложните операции с файла засягат цели структури от данни.

3.3.8.1. Списък на предметите

Тъй като обектът, представящ свързания списък на предметите, има достъп до обектите на всички предмети в паметта, генерирането на списък на предметите е наистина несложна операция:

1. Публичният интерфейс получава чрез модула за управление на файла достъп до обекта, представящ списъка на съхранените предмети;
2. Публичният интерфейс изисква от обекта, представящ списъка на съхранените предмети, списък с идентификаторите на всички съхранени предмети;
3. Обектът обхожда елементите на списъка с предмети, добавяйки идентификаторите им в масив;

4. Така създаденият масив се връща на публичния интерфейс, а от него – на клиентското приложение.

3.3.8.2. Запис на допълнителни данни

На пръв поглед записът на допълнителни данни изглежда дълга и сложна операция, но на практика е съвсем проста. Тя се състои от три основни стъпки: разширяване на масива на допълнителните данни, ако това е необходимо, съхранение на данните, и насочване на дескриптора на съответния слот към тях.

1. По процедурата, описана в точка 3.3.7.1, се записва съдържанието на допълнителните данни; резултатът от операцията е дескриптор, несъхранен във файла, сочещ към съхранените данни;
2. Ако съдържанието е криптирано, по същият начин се записва инициализационният вектор;
3. Създава се обект на класа, представящ допълнителните данни, който се инициализира с двата дескриптора;
4. Чрез модула за работа с файла се получава достъп до обекта-колекция на допълнителните данни;
5. Създаденият в стъпка 3 обект на допълнителните данни се подава за съхранение на колекцията на допълнителните данни;
6. Ако масивът на допълнителните данни вече е достатъчно голям, така, че да притежава слот за съхраняваните данни, се преминава направо на стъпка 10; в противен случай се продължава с увеличаването на размера на масива;
7. Разширява се структурата със слотове в паметта; тя се записва във файла по процедурата, описана в стъпка 3.3.7.1, като в резултат се получава несъхранен във файла дескриптор към новосъхранения масив;

8. Чрез модула за работа с файла се получава достъп до обекта, представящ заглавния блок на файла, а чрез него – до дескриптора, отговарящ за масива с допълнителните данни във файла;
9. Съдържанието на новия дескриптор се прехвърля в съдържанието на дескриптора в заглавния блок на файла, пренасочвайки го към разширената версия на масива с данни. При това старата версия се превръща в свободно пространство;
10. По процедурата от точка 3.3.7.1 се записва съдържанието на обекта на допълнителните данни и се получава несъхранен във файла дескриптор към съхранените данни;
11. Съдържанието на несъхранения дескриптор се прехвърля в съдържанието на дескриптора за съответния слот на масива, насочвайки го към новосъхранените данни.

3.3.8.3. Четене на допълнителни данни

Четенето на допълнителните данни става по следната кратка процедура:

1. Чрез модула за работа с файла се осъществява достъп до обекта, представящ масива с допълнителни данни;
2. Чрез него се получава достъп до обекта, представящ данните в конкретния слот;
3. Използва се свойството на така получения обект, връщащо съдържанието на конкретните допълнителни данни;
4. Така извиканото свойство използва методите, предложени от дескриптора на съдържанието и дескриптора на инициализационния вектор, за да ги прочете от файла.

3.3.8.4. Запис на предмет или на мета-данните му

Подобно на записа на допълнителни данни, записването на предмет се състои в запис на отделните му компоненти, запис на елемент на списъка с предмети, и включването (или подмяната) му в свързания списък на елементите. Отново, файлът се поддържа в консистентно състояние във всички фази на операцията.

1. Ако съдържанието на предмета идва от файл, съхраняваме съдържанието му във външен файл, както описахме в точка 3.2.5 – „Външни файлове", и използваме за съдържанието дескриптор, съдържащ специалната стойност, указваща, че съдържанието на предмета може да бъде намерена във външен файл. В противен случай съхраняваме съдържанието в основния файл според процедурата, описана в точка 3.3.7.1, в резултат получавайки несъхранен във файла дескриптор, сочещ към него.
2. Отново чрез процедурата от точка 3.3.7.1, се съхраняват мета-данните, инициализационният им вектор, както и инициализационният вектор на съдържанието.
3. С четирите несъхранени дескриптора, получени от предните две стъпки, се създава обект, представящ предмета.
4. Чрез модула за работа с файла се получава достъп до обекта, представящ списъка с предметите. Новосъздаденият предмет, съдържащ четирите дескриптора, му се подава за съхранение;
5. Списъкът с предметите създава нов елемент на списъка, съдържащ новия предмет, и го съхранява във файла по процедурата от точка 3.3.7.1, получавайки дескриптор към него;
6. Списъкът след това преценява къде трябва да бъде включен новият елемент;

7. Дескрипторът за следващия елемент на новосъздадения елемент се насочва към съответния следващ елемент в списъка. Забележете, че файлът все още е в консистентно състояние;
8. Новосъздаденият елемент се включва в списъка а подмененият, ако има такъв, едновременно с това се изключва, чрез пренасочването на един-единствен дескриптор, в чиято стойност се прехвърля стойността на несъхранения дескриптор, получен на стъпка 5. В случай, че се подменя вече съществуващ елемент, се пренасочва дескрипторът на съответния предходен елемент. В противен случай се пренасочва дескрипторът на заглавния блок, сочещ към първия елемент в списъка, като по този начин новият елемент се вмъква в началото му.

3.3.8.5. Четене на предмет или на мета-данните му

Както винаги, четенето е изключително проста и бърза операция:

1. Чрез модула за работа с файла се получава достъп до обекта, представящ списъка на предметите;
2. Използва се списъка на предметите, за да се получи достъп до обекта, представящ конкретния предмет, чиито мета-данни или данни трябва да бъдат прочетени;
3. Използват се свойствата на този предмет, връщащи мета-данните или съдържанието. Те, от своя страна, използват методите на дескрипторите, за да получат данните или мета-данните, както и техните инициализационни вектори.

3.3.8.6. Изтриване на предмет

Изтриването на предмет е сравнително проста операция, състояща се в изключването на елемент от списъка с предмети. Това става чрез едно-единствено пренасочване на дескриптор.

1. Чрез модула за работа с файла се осъществява достъп до обекта, представящ списъка на предметите;
2. Извиква се неговият метод за изтриване на предмет по зададен идентификатор;
3. Методът за изтриване намира елемента, отговарящ на изтривания предмет, и пренасочва съответния дескриптор в предходния му елемент (или в заглавния блок, в случай, че изтриваният елемент е пръв в списъка) така, че да изключи елемента от списъка;
4. Унищожават се обектите на свързания списък и на елемента, който описва. При това всички техни дескриптори се унищожават, регистрацията им се премахва, и мястото, сочено от тях, се маркира като свободно.

4. Описание на реализацията

4.1. Общи съображения

4.1.1. Стратегия за заделяне на място

Конкретната стратегия за заделяне на мястото във файла с данни ще има влияние върху много различни аспекти на работата на компонента – колко място ще стои неизползвано във файла, доколко фрагментирано ще е свободното място в него, както и колко време ще отнема самото заделяне (използването на сложен алгоритъм за избор на подходящ свободен блок може да бъде бавно на устройство с ограничени изчислителни ресурси като Pocket PC).

Съществуват най-различни стратегии и алгоритми за заделяне на място. Те се използват както при реализацията на различни файлови формати и файлови системи, така и при заделянето на оперативна памет – при различните видове алокатори на памет.

4.1.1.1. Стратегии, използващи блокове с фиксиран размер

Някои класове алгоритми работят с блокове с фиксиран размер (Wikipedia, 2006). При тях, всички блокове използват един и същи фиксиран размер, като заделянето се осъществява чрез отделянето на цял брой блокове и маркирането им като заети. Докато тези подходи имат своите предимства при алокаторите на оперативна памет, тъй като решават проблеми, свързани с представянето на свободната памет по някакъв подходящ и удобен начин, те нямат реално отношение към разглеждания от нас проблем за заделяне на място във файл. Работата с блокове с фиксиран размер е просто прехвърляне на проблема от заделянето на единични байтове до заделянето на блокове с някаква

фиксирана дължина; това по никакъв начин не адресира проблема за намирането на подходяща поредица от блокове при заделяне на обем данни, по-голям от използвания размер на блока, така, че да се намали фрагментацията на свободното пространство във файла.

Интересен клас алгоритми, целящи намаляване на фрагментацията на свободното пространство, използват блокове с експоненциално нарастващи фиксирани размери – например 64, 128, 256, …, 2^n байта (Newcomer, 2006; Wikipedia, 2006). Изборът на подходящ блок при заделяне е бързо, а стратегията за разцепване на големите блокове на по-малки е проста, което е много подходящо за бързи и ефективни алокатори на оперативна памет.

За целите на разработвания компонент за съхранение на криптирани данни, обаче, използването на стратегия с фиксиран (или с ограничен набор) размер на блоковете, би означавало похабяването на свободно място – нещо, което целим да избегнем, тъй като компонентът ще бъде използван на устройство със силно ограничена памет, където свободното пространство е найценният ресурс. Може да се спори, че фрагментирано свободно място, състоящо се от блокове, чакащи да бъдат използвани наново, също е прахосано, но то поне може да бъде употребено отново, или пък, при нужда, „изстискано" чрез операция за свиване на файла. Неупотребеното място в краищата на блокове с фиксирана дължина, от друга страна, е похабено завинаги. Ето защо ще се концентрираме върху стратегии, използващи блокове с променлива дължина.

4.1.1.2. Стратегии, използващи блокове с произволна дължина

При работа с блокове с произволна дължина, съществуват най-различни стратегии за избор на блок, поради възможността

заделяният блок да бъде навсякъде, където има достатъчно свободно пространство. Следователно, бихме могли да класифицираме стратегиите за заделяне на място според това по какъв начин се избира подходящия блок (Newcomer, 2006).

Най-простият алгоритъм за заделяне, т.нар. „First Fit", се състои в простото избиране на първия блок с достатъчно голям размер. Ако блокът е по-голям от необходимото, се използва само началото му, а остатъкът се връща в списъка на свободното пространство. Проблемът с този метод е, че тенденцията на алгоритъма да не подбира блоковете и постоянно да разцепва големите блокове на по-малки води до постепенно увеличение на фрагментацията (Newcomer, 2006).

Друг алгоритъм е т.нар. „Best Fit", който се опитва да намери блок с идеалната големина. Ако съществува свободен блок с големина, точно толкова, колкото се търси, то той ще бъде използван за съхранение на данните, което би било оптималният вариант. Проблемът с този алгоритъм е, че изисква списъкът на свободните блокове да бъде поддържан сортиран по големина, за да може бързо да се търси в него блок с оптимален или малко по-голям размер. Необходимостта от поддържане на сортиран списък намалява производителността на алгоритъма.

Съществува и т.нар. алгоритъм „Quick Fit", при който се поддържа кеш на свободните блокове, разпределени по размер. Quick Fit се базира на принципа, че в повечето случаи програмите заделят обекти с ограничен набор размери, така че освобождаването на блок с определена големина може скоро да бъде последвана от заделянето на блок с точно същата големина. Следователно, вместо при освобождаване на блок той да бъде сливан с останалото свободно пространство около него, алокаторът го запазва в кеша, за да може той бързо да бъде заделен отново (Newcomer, 2006).

Както се вижда от описанието на Quick Fit, той е силно ориентиран към подобряване на производителността на алокатори на оперативна памет. Той обаче прави малко, за да адресира фрагментацията на свободното пространство. Производителността на заделянето на свободно място във файла не е от такова голямо значение за компонента, който разработваме, тъй като операции по заделяне ще се осъществяват сравнително рядко, когато потребителят на приложението, използващо компонента, има някакви данни за съхранение. Ето защо, за нас най-важният критерий е използването на свободното пространство.

4.1.1.3. Избор на стратегия

Имайки предвид горните съображения, избрахме алгоритъма Best Fit за реализацията на компонента. При съхранение на данни ще използваме най-малкия блок, който може да събере данните; при наличието на повече от един блок с най-малкия такъв размер, ще използваме този блок, който е по-близо до началото на файла. Предимствата на този подход са:

1. Поради избора на най-малкия блок, който може да събере данните, оставяме по-големите блокове достъпни за евентуалното бъдещо съхранение на по-големи обеми от информация. По този начин при следващо съхранение на информация има по-голяма вероятност тя да бъде съхранена вътре в свободното пространство във файла, с което се минимизира необходимостта от неговото удължаване.
2. Изборът на блок, възможно най-близо до началото на файла, когато има повече от един подходящ блок, цели да минимизира необходимостта от преместването на големи обеми от данни по време на операция за свиване на файла, тъй като свободното пространство е по-близо до края на файла.

4.1.2. Данните – като масиви от байтове

Както разгледахме подробно нееднократно, използваният от нас файлов формат изисква предварително заделяне на пространство във файла с данни за дадена информация, преди тя да бъде съхранена. Това обаче налага изискването да се знае дължината на записваните данни.

Ако горното изискване изглежда странно, нека си припомним, че, както разгледахме в точка 3.3.1 – „Комуникация чрез потоци", данните на приложението, използващо компонента, идват от него във вид на поток. За да криптира данните, приложението използва криптиращ поток измежду предлаганите в .NET Compact Framework криптографска функционалност (Йорданова, 2007). Проблемът е, че криптиращият поток криптира информацията в процеса на нейното четене, а докато не бъде криптирана информацията, няма как да се знае какъв обем заема тя в криптиран вид. Поради тази причина криптиращите потоци, предлагани от .NET Compact Framework, не поддържат непоследователно четене, и няма как да бъде установена тяхната дължина.

Резултатът за компонента за съхранение на данните е, че той няма как да знае дължината на идващите от приложението криптирани данни, преди те да са прочетени напълно. Ето защо се налага данните да бъдат предварително прочетени от компонента в масив или в поток, пишещ в паметта (memory stream), за да бъде намерена тяхната дължина. Тогава вече може да бъде заделено място във файла за тях, в което те да бъдат съхранени.

На пръв поглед горният параграф противоречи на цялата концепция за комуникация чрез потоци, обсъдена в точка 3.3.1, чиято цел беше да поддържаме съхранението на данни, чийто размер би могъл да надхвърля обема на наличната оперативна памет. На помощ обаче идва фактът, че избрахме да съхраняваме големите потребителски данни, идващи от файлове (като крип-

тирани файлове или картинки), във външни файлове, а не в основния файл с данни (виж 3.2.5 – „Външни файлове"). Това означава, че в основния файл се съхраняват данни с малък обем - информация, идваща директно от потребителя, въведена чрез потребителския интерфейс на приложението, както и инициализационни вектори и друга служебна информация на приложението. Следователно прочитането ѝ в паметта на компонента преди окончателният ѝ запис във файла с данни не представлява никакъв проблем. Големите данни, които биват съхранявани във външни файлове извън основния файл с данни, няма нужда да бъдат прочитани в паметта, тъй като компонентът няма нужда да знае тяхната големина. Тъй като записът на данни във външен файл не е свързано с предварително заделяне на място, то може да бъде осъществено на порции, без дължината на данните да е предварително известна.

За прочитането на информацията, идваща от приложението, в паметта на компонента, можем да мислим като за получаването ѝ в криптиран вид. Приложението, използващо компонента, притежава нейно копие в паметта си в некриптиран вид, който то е превърнало в поток (четящ от паметта), което после е обгърнало с криптиращ поток. Прочитането на данните от страна на компонента, следователно, предизвиква криптиране на данните. След като компонентът ги получи в криптиран вид в паметта и види колко са големи, той може да премине към тяхното съхранение.

4.1.3. Двупасов запис на данни

Също както при записът на информация, идваща от приложението, компонентът за съхранение на данните трябва да знае какъв размер имат различните му служебни структури във файла (като например елемент от списъка с предмети, масива с допълнителните данни на приложението, и т.н.), за да може да задели

правилно място за тяхното съхранение. Някои от тези структури, обаче, като масива с допълнителните данни на приложението, са с променлива дължина; за структурите с фиксирана дължина, от друга страна, не е хубаво да бъдат твърдо забити дължините им в изходния код на приложението, тъй като това затруднява неговата поддръжка. Ето защо, компонентът трябва да разполага с някакъв механизъм за определяне на дължината на служебните файлови примитиви по време на изпълнение.

За определяне на дължината на примитивите, компонентът би могъл да подходи както към данните, идващи от приложението – той би могъл да запише тяхно копие в поток, пишещ в паметта, да провери получената дължина, да задели необходимото пространство във файла, след което да прехвърли данните в него. Този подход, обаче, не е възможен, заради наличието на дескриптори в служебните блокове.

Както обсъдихме в точка 3.3.2.2 – „Представяне на дескрипторите като обекти", всеки дескрипторен обект запомня при записването си в поток, къде в него е записан, за да може, при пренасочване на дескриптора, той да запише автоматично новата стойност на дескриптора във файла. Това означава, обаче, че дескриптори, както и служебни структури, съдържащи дескриптори, трябва винаги да бъдат записвани директно във файла с данни. Ако използваме поток за запис в паметта, за да получим представяне на обекта като масив от байтове, обектът на дескриптора погрешно ще запомни позицията си в потока за запис в паметта. Последващото прехвърляне на байтовия поток в окончателния файл с данните не би опреснило обекта, което би му попречило правилно да опреснява представянето си във файла с данни при последващи пренасочвания.

Горният проблем можем да решим елегантно чрез използването на двупасов запис за всички файлови примитиви и структу-

ри. При първия пас се извършва симулирано съхранение на примитива, при което само се определя изискваното от него пространство за съхранение; това позволява правилно заделяне на необходимото за съхранение на примитива пространство. На втория пас вече примитивът се съхранява наистина в така заделеното пространство.

4.2. Основи

```
Utils
-bufferSize : int = 512
+ReadIntFromStream(in stream : Stream) : int
+WriteIntToStream(in i : int, in stream : Stream, in estimate : bool) : uint
+ReadUIntFromStream(in stream : Stream) : uint
+WriteUIntToStream(in ui : uint, in stream : Stream, in estimate : bool) : uint
+ReadLongFromStream(in stream : Stream) : long
+WriteLongToStream(in l : long, in stream : Stream, in estimate : bool) : uint
+ReadULongFromStream(in stream : Stream) : ulong
+WriteULongToStream(in ul : ulong, in stream : Stream, in estimate : bool) : uint
+ReadBoolFromStream(in stream : Stream) : bool
+WriteBoolToStream(in b : bool, in stream : Stream, in estimate : bool) : uint
+ReadStringFromStream(in stream : Stream) : string
+WriteStringToStream(in s : string, in stream : Stream, in estimate : bool) : uint
+FillByteArrayFromStream(in bytes : byte[], in stream : Stream)
+ReadByteArrayWithLengthFromStream(in stream : Stream) : byte[]
+WriteByteArrayToStream(in bytes : byte[], in stream : Stream)
+WriteByteArrayWithLengthToStream(in bytes : byte[], in stream : Stream, in estimate : bool) : uint
+ReadStream(in stream : Stream) : byte[]
+CopyFromStreamToStream(in source : Stream, in destination : Stream, in length : int, in controller : SlowOperationController)
+CopyInsideStream(in stream : Stream, in sourceOffset : ulong, in destinationOffset : ulong, in length : uint, in controller : SlowOperationController)
+CopyWholeStreamToStream(in source : Stream, in destination : Stream, in controller : SlowOperationController) : int
+ByteArraysMatch(in array1 : byte[], in array2 : byte[]) : bool

ItemID
-guidLength : int = 16
-id : Guid
+ItemID()
+ItemID(in stream : Stream)
+WriteToStream(in stream : Stream, in estimate : bool) : uint
+CompareTo(in obj : object) : int
+Equals(in obj : object) : bool
+GetHashCode() : int
+ID() : Guid

IV
-bytes : byte[]
+IV(in iv : byte[])
+Bytes() : byte[]

Content
#data : Stream
+Content(in data : Stream)
+Content(in dataBytes : byte[])
-~Content()
+Dispose()
#Dispose(in disposing : bool)
+DetachData() : Stream

EncryptedContent
-iv : IV
+EncryptedContent(in encryptedData : Stream, in iv : IV)
+IV() : IV

SlowOperationController
-cancel : bool
+CancelRequested() : bool
+ThrowIfCancelRequested()
+StartProgress(in progressMax : ulong)
+ReportProgress(in progress : ulong)
+FinishProgress()
#OnProgressStarted(in e : ProgressStartedArgs)
#OnProgressReported(in e : ProgressReportedArgs)
#OnProgressFinished(in e : EventArgs)
+ProgressStarted() : EventHandler<PrivateDataCommon.ProgressStartedArgs>
+ProgressReported() : EventHandler<PrivateDataCommon.ProgressReportedArgs>
+ProgressFinished() : EventHandler
```

Фигура 4: Класове на Private Data Common

Тъй като компонентът за съхранение на данните трябва да бъде използван от друго приложение, необходимо е да бъдат де-

финирани някакви общи типове данни, позволяващи на двете да си комуникират. Тези общи типове са дефинирани в пакета „Private Data Common” (Фигура 4). Използвахме общия пакет, за да реализираме и обща полезна функционалност (помощни методи), които биха били полезни при реализацията както на компонента, така и на използващото го приложение.

4.2.1. Бавни операции

За постигане на отзивчив потребителски интерфейс, приложението за съхранение на криптирани данни изпълнява бавните операции (повечето от които включват работа с данните) асинхронно. За поддържане на комуникация с кода, изпълняващ операцията, се използва специализиран комуникационен обект, който се предава надолу по веригата на изпълнение (Йорданова, 2007).

4.2.1.1. *SlowOperationController*

Класът SlowOperationController реализира функционалността, използван за комуникация с потребителския интерфейс. Тъй като реализира комуникация между отделни нишки, всички методи и свойства на обекта за обезопасени за работа с няколко нишки (thread-safe):

- Функционалност за прекратяване на операцията:
    - Свойство CancelRequested, чрез което може да бъде установена или проверена стойността на флаг, показващ, че потребителят иска преждевременно прекратяване на операцията;
    - Метод ThrowIfCancelRequested, който може да бъде извикан от кода на операцията в момент, в който преждевременното ѝ прекратяване е безопасно. Ако флагът, показващ, че операцията трябва да бъде прекратена, е установен, методът ThrowIfCancelRe-

quested хвърля изключение от тип OperationCancelledException, ефективно прекратявайки операцията.

- Функционалност за докладване на прогреса на операцията:
    - Метод StartProgress – използва се в началото на бавна функция или функция, изпълняваща няколко стъпки, за да съобщи на потребителския интерфейс колко стъпки има да изпълнява, по този начин помагайки му да прецени степента на прогрес на операцията;
    - Метод ReportProgress – използва се, за да бъде уведомен потребителският интерфейс, че текущата функция е изпълнила определен брой от предварително обявените стъпки;
    - Метод FinishProgress – използва се в края на функция, докладвала прогрес чрез StartProgress и ReportProgress.
    - Събития, извиквани от горните три метода. Използват се от приложението, което се абонира за тях, за да следи прогреса на операцията.

4.2.2. Помощни методи

Класа Utils реализира набор от помощни методи, използвани както от компонента за съхранение на данните, така и от използващото го приложение. Тези методи включват функционалност за:

- Съхранение и прочитане на основни типове данни в поток:
    - ReadIntFromStream / WriteIntToStream
    - ReadUIntFromStream / WriteUIntToStream
    - ReadLongFromStream / WriteLongToStream

- ReadULongFromStream / WriteULongToStream
- ReadBoolFromStream / WriteBoolToStream
- ReadStringFromStream / WriteStringToStream

- Съхранение и прочитане на масиви от байтове, с или без предхождаща ги дължина, в поток:
  - FillByteArrayFromStream – запълва масив с предварително известна дължина от поток чрез прочитане на необходимия брой байтове;
  - ReadByteArrayWithLengthFromStream – прочита поредица от байтове, предшествана от дължината си, от потока в масив от байтове;
  - WriteByteArrayToStream – записва масив от байтове в поток;
  - WriteByteArrayWithLengthToStream – записва масив от байтове в поток, като преди него съхранява и дължината му;
  - ReadStream – прочита цял поток, от текущата позиция до самия му край, връщайки резултата като масив от байтове. Не е нужно потокът да поддържа непоследователно четене или да се знае дължината му;
- Прехвърляне на данни между потоци:
  - CopyFromStreamToStream – копира зададен брой байтове от текущата позиция на даден поток в друг, започвайки от текущата му позиция. Операцията поема SlowOperationController и може да бъде прекратена преждевременно;
  - CopyInsideStream – копира зададен брой байтове от дадена позиция на даден поток в друга позиция на същия поток. Тази операция е полезна в процеса на

свиване на файла, когато блокове данни трябва да бъдат копирани във файла с данни. Операцията поема SlowOperationController и може да бъде прекратена преждевременно;

    - CopyWholeStreamToStream – копира всички данни от текущата позиция на даден поток до самия му край в друг поток. За целта се използва ограничено количество памет (буфер), който се запълва многократно от входния поток и се изпразва в изходния. Операцията поема SlowOperationController и може да бъде прекратена преждевременно;

- Сравнение на съдържанието на масиви от байтове:
    - ByteArraysMatch – сравнява два масива от байтове, връщайки „истина”, ако дължините и съдържанията им съвпадат, или „лъжа” в противен случай.

4.2.3. Основни типове данни

Основните типове данни, споделяни от компонента и използващото го приложение, включват идентификатор на предмет, инициализационен вектор, съдържание, и криптирано съдържание.

4.2.3.1. ItemID

Класът ItemID капсулира идентификаторът на предмет (който, както обсъдихме в точка 3.1.1 – „Данни”, всъщност представлява глобално-уникален идентификатор – GUID).

Предлаганата функционалност включва:

- Конструиране на идентификатор:
    - Конструиране на нов идентификатор;
    - Прочитане на идентификатор от поток;
- Метод WriteToStream, реализиращ двупасов запис в поток;

- Методи за сравнение и извличане на хеш код от идентификатора, позволяващи използването му като ключ на сортирани колекции и хеш таблици;
- Свойство ID, връщащо представянето на идентификатора като глобално-уникален идентификатор (GUID).

4.2.3.2. IV

Класът IV капсулира инициализационен вектор, използван за криптиране или декриптиране на информация от приложението, използващо компонента. По своето същество инициализационните вектори са просто масиви от байтове, връщани или подавани на криптиращите алгоритми. Капсулирането им в отделен клас позволява по-добър стил на кода поради по-силната им типизираност.

4.2.3.3. Content

Класът Content представя всякакви данни, които приложението съхранява или извлича от компонента за съхранение на данните. Тъй като приложението и компонентът комуникират чрез потоци, в сърцето на Content всъщност се съхранява отворен поток, готов да бъде прочетен от потребителя на обекта.

Content предлага:

- Функционалност за създаване на обект от тип Content:
    - Конструиране чрез поток. Съхранява отворения поток в обекта;
    - Конструиране чрез масив от байтове. Създава върху тях поток за работа с паметта (memory stream), който съхранява в обекта;
- Унищожаване на обект:
    - Dispose – затваря потока с данни, освобождавайки заеманите от него ресурси;

- Извличане на данните от Content:
    - Метод DetachData – „откачва” и връща потока, държан в Content. Обектът губи собствеността над потока. Кодът, извикал метода, получава отговорността по затварянето и унищожаването на потока, когато приключи работа с него.

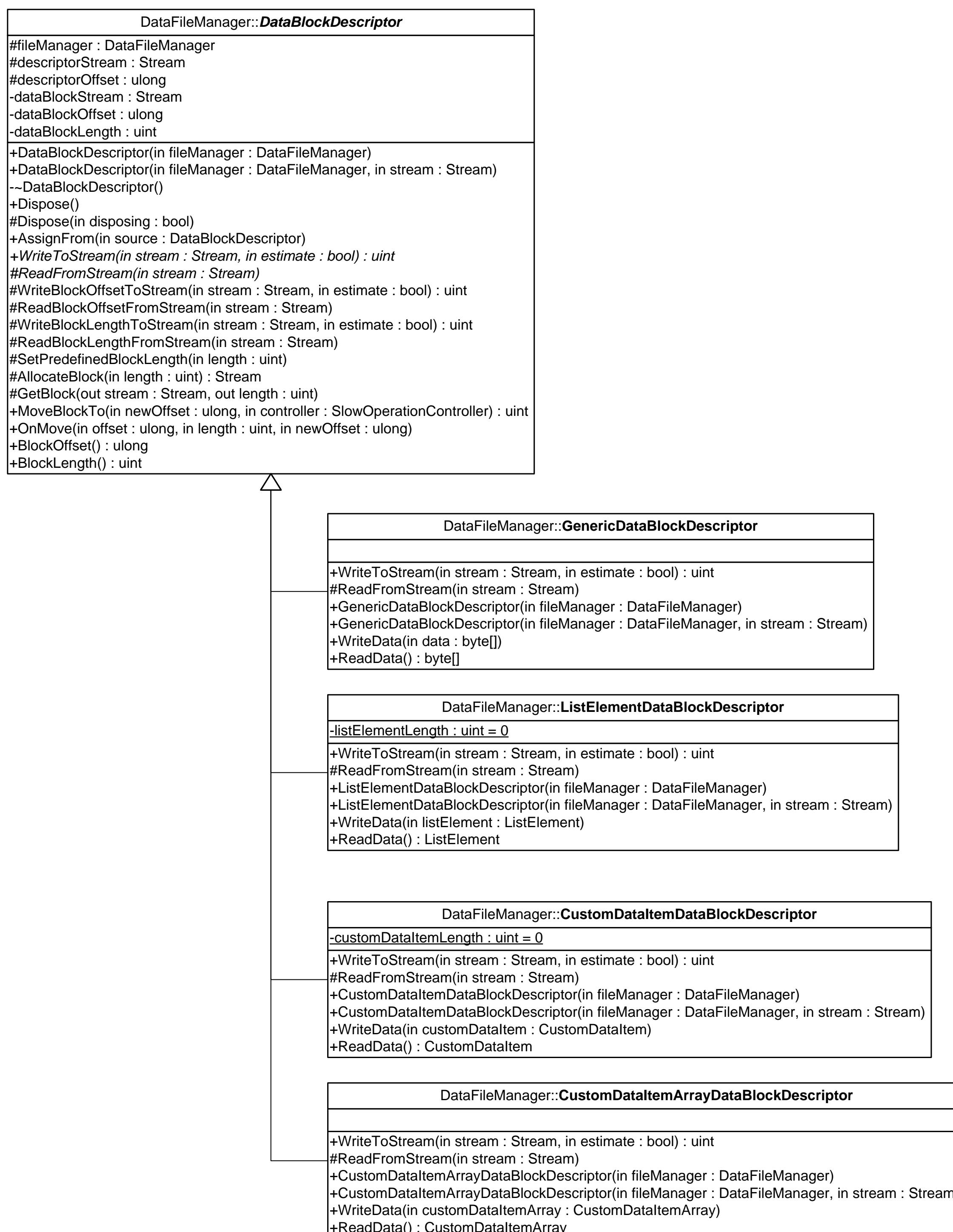
4.2.3.4. EncryptedContent

Класът EncryptedContent наследява Content, разширявайки го, за да представи криптирани данни, обменяни между приложението и компонента за съхранение на данните.

Както споменахме в точка 3.1.1 – „Данни”, криптираните данни и техният инициализационен вектор вървят винаги заедно, като никое от тях няма смисъл без наличието на другото. EncryptedContent е методът за постигане на тази цялост. Освен функционалността на Content, EncryptedContent предлага свойство за получаване на обект от класа IV, представящ инициализационния вектор на данните.

4.3. Дескриптори

В точките 3.3.2.2 – „Представяне на дескрипторите като обекти” и 3.3.2.3 – „Йерархия от дескрипторни класове” обсъдихме представянето на дескрипторите като йерархия от класове, чиито обекти „помнят” местоположението на сочените от дескрипторите блокове във файла, както и местоположението на самите дескриптори, записвайки автоматично във файла новите им стойности при пренасочване. Тази йерархия е реализирана в компонента чрез описаните в настоящата точка класове (Фигура 5).



Фигура 5: Йерархия на дескрипторите

Класовете за дескрипторите са реализирани в рамките на класа DataFileManager, за да имат достъп до неговите частни член-променливи и методи, предназначени за работа с файла. DataFileManager ще разгледаме по-подробно в точка 4.5.1.

4.3.1. DataBlockDescriptor

Абстрактният клас DataBlockDescriptor е базовият клас на йерархията от дескриптори, предлагайки основната функционалност, която всички дескриптори притежават:

- Инициализация на дескриптор:
    - Конструиране на празен дескриптор;
    - Прочитане на дескриптор от поток (извиква предефинирания за целта от наследниците метод ReadFromStream), запомняйки местоположението на дескриптора в потока;
    - Метод SetPredefinedBlockLength, който наследените класове могат да извикат, за да установят изрично дължината на сочения блок с данни, в случай, че тя е предварително известна (т.е. в случаите, когато наследниците представят дескриптори, сочещи към служебни блокове с данни);
- Унищожаване на дескриптор:
    - Dispose – унищожава дескриптор, премахвайки регистрацията му в модула за работа с файла;
- Пренасочване на дескриптор:
    - AssignFrom – присвоява на дескриптора стойността на друг дескриптор, опреснявайки дескриптора във файла да сочи към новата позиция на данните (виж точка 3.3.2.4 – „Правилна работа с дескрипторите”). Изрично се проверява, че двата дескриптора са от един и същи тип;
- Запис на дескриптор в поток:
    - WriteBlockOffsetToStream – метод, който имплементацията на WriteToStream на наследените класове

може да извика, за да съхрани отместването на блока, към който дескрипторът сочи, в поток;

- WriteBlockLengthToStream – метод, който имплементацията на WriteToStream на наследените класове може да извика, за да съхрани дължината на блока, към който дескрипторът сочи, в поток;
- Абстрактен метод WriteToStream, който наследниците предефинират. Наследниците имат избор да извикат WriteBlockOffsetToStream и WriteBlockLengthToStream. Може и да не извикат WriteBlockLengthToStream, в случай, че представят дескриптор към служебен блок, чиято дължина е предварително известна.

- Четене на дескриптор от поток:
  - ReadBlockOffsetFromStream – метод, който имплементацията на ReadFromStream на наследените класове може да извика, за да прочете отместването на блока, към който дескрипторът сочи, от поток;
  - ReadBlockLengthFromStream – метод, който имплементацията на ReadFromStream на наследените класове може да извика, за да прочете дължината на блока, към който дескрипторът сочи, от поток;
  - Абстрактен метод ReadFromStream, който наследниците предефинират. Наследниците имат избор да извикат ReadBlockOffsetFromStream и ReadBlockLengthFromStream, в зависимост от това по какъв начин е бил записан дескрипторът във файла.
- Методи, позволяващи на наследените класове да работят с данните, сочени от дескриптора:

- AllocateBlock – използва модула за работа с файла, за да задели блок с поисканата дължина във файла. След като блокът е заделен, регистрира дескриптора в модула за работа с файла, и позволява на наследниците да запишат данните, които имат да запишат, в така заделения блок;
- GetBlock – подготвя се за четене на данните, сочени от дескриптора. Връща на наследения клас поток, готов за четене на данните, както и дължината на сочените данни;

- Методи, използвани по време на операция за свиване на файла:
  - MoveBlockTo – извиква се от операцията за свиване на файла, за да бъдат преместени данните, сочени от дескриптора, на друго отместване. Подробно описание на операцията може да бъде намерено в точка 3.3.7.4 – „Свиване на файла”;
  - OnMove – метод, използван от операцията за свиване на файла, за да уведоми дескриптор, че блокът, в който се намира, е преместен, за да може обектът на дескриптора да си отбележи новото си местоположение във файла с данни;
  - Свойство BlockOffset – връща отместването на блока, сочен от дескриптора. Следва да се използва само от операцията по свиване на файла. Всички останали потребители трябва да използват другите методи за четене и писане на данни на дескриптора и не би трябвало да се интересуват от отместването на данните.
- Намиране на дължината на блока, сочен от дескриптора:

- Свойство BlockLength

4.3.2. GenericDataBlockDescriptor

Дескриптор, сочещ общ блок с данни без специална структура (т.е. масив от байтове). Винаги съхранява дължината на сочените данни, когато записва дескриптора във файла. Предлага методи за четене и запис на блока, към който сочи:

- Метод WriteData, записващ масив от байтове през дескриптора (заделяйки необходимото място и записвайки данните, както описахме в точка 3.3.7.1 – „Запис на блок с данни".
- Метод ReadData, прочитащ блока, сочен от дескриптора, като масив от байтове.

4.3.3. ListElementDataBlockDescriptor

Дескриптор, сочещ блок, съдържащ елемент от списъка с предмети. Не съхранява дължината на сочените данни, когато записва дескриптора във файла, тъй като дължината на блока е винаги предварително известна и константна. Предлага методи за четене и запис на блока, към който сочи:

- Метод WriteData, записващ обект от тип ListElement през дескриптора (заделяйки необходимото място и записвайки данните, както описахме в точка 3.3.7.1 – „Запис на блок с данни".
- Метод ReadData, прочитащ блока, сочен от дескриптора, като обект от тип ListElement.

4.3.4. CustomDataItemDataBlockDescriptor

Дескриптор, сочещ блок, описващ допълнителни данни на приложението, използващо компонента (виж точка 3.3.3.5 – „Допълнителни данни". Не съхранява дължината на сочените данни, когато записва дескриптора във файла, тъй като дължи-

ната на блока е винаги предварително известна и константна (блокът съдържа само два дескриптора). Предлага методи за четене и запис на блока, към който сочи:

- Метод WriteData, записващ обект от тип CustomDataItem през дескриптора (заделяйки необходимото място и записвайки данните, както описахме в точка 3.3.7.1 – „Запис на блок с данни".
- Метод ReadData, прочитащ блока, сочен от дескриптора, като обект от тип CustomDataItem.

4.3.5. CustomDataItemArrayDataBlockDescriptor

Дескриптор, сочещ блок, съдържащ масива с допълнителните предмети на приложението, използващо компонента (виж точка 3.3.3.7 – „Масив с допълнителните данни". Винаги съхранява дължината на сочените данни, когато записва дескриптора във файла. Предлага методи за четене и запис на блока, към който сочи:

- Метод WriteData, записващ обект от тип CustomDataItemArray през дескриптора (заделяйки необходимото място и записвайки данните, както описахме в точка 3.3.7.1 – „Запис на блок с данни".
- Метод ReadData, прочитащ блока, сочен от дескриптора, като обект от тип CustomDataItemArray.

4.4. Файлови примитиви

Класовете, описани в настоящата точка, реализират функционалността на различните файлови примитиви, описани в точка 3.3.3 – „Представяне на файловите примитиви".

4.4.1. FileHeader

Класът FileHeader реализира функционалността на заглавния блок на файла с данни (виж точки 3.2.2 и 3.3.3.1). Предлагат

се конструктори за създаване на нов заглавен блок, както и за прочитането на съществуващ заглавен блок от файл с данни. Предлага се и метод WriteToStream, реализиращ двупасов запис на заглавния блок във файла. Свойствата CustomDataItemArrayDescriptor и FirstItemListElementDescriptor, от своя страна, дават достъп до двата дескриптора, съхранени в заглавния блок.

4.4.2. Item

Класът Item реализира функционалността на предмет, съхранен във файла с данни (виж точка 3.3.3.2). Предлагат се конструктори за създаване на нов предмет по подадени четири дескриптора (дескриптор към мета-данните, инициализационния вектор на мета-данните, данните, и инициализационния вектор на данните), както и за прочитането на дескрипторите на предмет от файла с данни. Предлага се и метод WriteToStream, реализиращ двупасов запис на дескрипторите на предмет във файла.

Както описахме в точка 3.3.3.2, въпреки, че предметът реално съхранява четири дескриптора, бихме могли да капсулираме функционалността по прочитане на блоковете, към които сочат, в самия предмет. Ето защо, вместо достъп до дескрипторите, се предоставят свойствата Metadata и Content, връщащи, съответно, мета-данните и данните на предмета във вид на обекти от тип EncryptedContent[7].

За определяне дали съдържанието на предмета е съхранено във външен файл се предлага булевото свойство External. Ако предметът наистина е съхранен във външен файл, свойството му Content връща null. В този случай за достъп до инициализацион-

[7] Всъщност, налични са свойства ContentDescriptor и ContentIVDescriptor, даващи достъп до дескрипторите за данните на предмета и инициализационния им вектор. Те се използват от операцията за съхранение само на мета-данните на съществуващ предмет. Тъй като съхранението на предмет изисква четири дескриптора, операцията за съхранение на мета-данните трябва да използва вече съществуващите дескриптори за данните и инициализационния им вектор.

ния вектор на данните може да бъде използвано предлаганото свойство ContentIV.

4.4.3. ItemList; ListElement

Класът ListElement, вътрешен за ItemList, реализира функционалността на елемент от списъка с предмети (виж точки 3.2.4 и 3.3.3.3). Предлагат се конструктори за създаване на нов предмет по подадени дескриптор за следващ предмет, идентификатор на предмет, и обект, представящ предмета, както и за прочитането на елемент на списъка с предмети от файла с данни. Предлага се и метод WriteToStream, реализиращ двупасов запис на елемента във файла.

Свойствата на ListElement позволяват достъп дескриптора към следващия елемент (свойството NextElementDescriptor), идентификатора на предмета (свойството ID, връщащо обект от тип ItemID), и обекта, представящ предмета (свойството Item, връщащо обект от тип Item). Предлагат се и средства за навигация на двойно-свързания списък, описан в точка 3.3.3.3, в паметта (свойствата Previous и Next).

ListElement се използва вътрешно от класа ItemList, реализиращ функционалността на колекцията – списък с предмети (виж точки 3.2.4 и 3.3.3.4). Предлага се конструктор за създаване на нов списък върху основния файл с данни; конструкторът взима дескриптора към първия елемент на списъка от заглавния блок на файла и пресъздава списъка в паметта, създавайки двойно-свързания списък от обекти ListElement, включвайки ги същевременно в хеш таблицата, позволяваща бързото им намиране по идентификатор на предмета (виж точка 3.3.3.4).

За работа с предметите, ItemList предлага метод Store, който приема обекти от тип ItemID и Item и позволява добавянето на предмет, вече съхранен във файла, в свързания списък на пред-

метите. Ако ли пък предмет с посочения идентификатор вече съществува в списъка, той бива подменен с новия вариант. Store позволява на потребителите на класа да работят със свързания списък във файла, скривайки от тях детайлите по неговото поддържане. Методът Delete позволява изтриването на предмет по зададен идентификатор, отново скривайки сложността по поддържането на свързания списък от потребителите на класа.

Предлагат се и индексатор, позволяващ достъп до предмет по зададен идентификатор (използва се ускоряващата хеш таблица за бърз достъп до предмета), както и енумератор, позволяващ обхождането на списъка с идентификатори на съхранените предмети. Наличен е и метод ToArray, връщащ масив с идентификаторите на всички съхранени предмети. Свойството Count пък позволява да се провери броят им.

Предлага се и метод Contains, който може да се използва, за да се провери дали съществува предмет със зададен идентификатор.

4.4.4. CustomDataItem

Класът CustomDataItem реализира функционалността на елемент допълнителни данни, съхранени от приложението (виж точка 3.3.3.5). Предлагат се конструктори за създаване на нов предмет по подадени два дескриптора (дескриптор към данните и дескриптор към техния инициализационен вектор, в случай, че са криптирани), както и за прочитането на дескрипторите от файла с данни. Предлага се и метод WriteToStream, реализиращ двупасов запис на дескрипторите, сочещи допълнителните данни, във файла.

Както описахме в точка 3.3.3.5, също както при предметите, въпреки, че обектът реално съхранява дескриптори, може да се капсулира функционалността по прочитане на блоковете, към които сочат, в самия обект. Така, вместо достъп до дескриптори-

те, се предоставя свойството Content, връщащо представяните допълнителни данни. В случай, че те са криптирани (т.е. ако е съхранен инициализационен вектор), връщаният обект от тип Content всъщност е от наследения тип EncryptedContent, носещ инициализационния вектор в допълнение към криптираните данни.

4.4.5. CustomDataItemArray; CustomDataItemArrayElement

Класът CustomDataItemArrayElement, вътрешен за CustomDataItemArray, реализира функционалността на елемент от масива с допълнителни данни на приложението (виж точки 3.2.3 и 3.3.3.6). Предлагат се конструктори за създаване на нов елемент, както и за прочитането на елемент на масива от файла с данни. При последния се прочита, също така, и самия обект, представящ допълнителните данни (както обсъдихме в точка 3.3.3.6), заради необходимостта дескрипторите му да бъдат налични в паметта. Предлагат се свойство Item, връщащо самия прочетен обект (от тип CustomDataItem), както и свойство Descriptor, връщащо дескриптора към него (обект от тип CustomDataItemDataBlockDescriptor).

Класът CustomDataItemArray реализира функционалността на колекцията – масив с допълнителни данни на приложението (виж точки 3.2.3 и 3.3.3.7). Предлагат се конструктори за създаване на нов масив, както и за прочитането на масив от файла с данни (прочита броя слотове, след което прочита всеки един от тях чрез създаването на обект от тип CustomDataItemArrayElement). Предлага се и метод WriteToStream, реализиращ двупасов запис на масива във файла.

За работа с предметите, CustomDataItemArray предлага метод Store, който позволява съхранението на допълнителни данни (обект от тип CustomDataItem) в слот на масива, скривайки от

потребителите на класа детайлите по неговата работа с файла. Предлага се и индексатор, позволяващ достъп до допълнителни данни по зададен номер на слот.

4.5. Модул за работа с файла

Както казахме в точка 3.3.5, модулът за работа с файла е единствената отправна точка, която публичният интерфейс, както и обектите, представящи файловите примитиви, използват за работа с файла. Той има грижата за прочитането на основните структури от данни като заглавния блок, списъка на предметите и масива с допълнителните данни. Под-модулите му се грижат за следене на блоковете с данни във файла, както и за управление на свободното пространство.

4.5.1. DataFileManager

Класът DataFileManager реализира функционалността на модула за работа с файла, описан в точка 3.3.5. В него се съхранява файловият поток на основния файл, който стои отворен за четене и запис през продължителността на живот на обекта. Поддържат се, също така, указатели (reference) към обектите, представящи заглавния блок, списъка на предметите, както и масива с допълнителните данни.

Дескрипторите, както и модула за управление на пространството, са реализирани като вътрешни за класа DataFileManager, за да могат да използват неговите частни методи и да имат достъп до потока на файла. Публичните методи на DataFileManager са фасадата, която публичният интерфейс вижда и с която работи, без да вижда скритата зад нея сложност на файла с данни. Дескрипторите и модула за управление на пространството (заедно със своите два под-модула), от своя страна, имат пълен достъп до сложната функционалност, реализираща работата с файла.

4.5.1.1. Публични методи и свойства

Публичните методи и свойства на DataFileManager включват:

- Конструиране на класа върху отворен файл с данни;
- Унищожаване на класа – метод Dispose (затваря файла);
- Свойство FileHeader, даващо достъп до обекта, представящ заглавния блок на файла;
- Свойство Items, даващо достъп до обекта, представящ колекцията – списък на предметите;
- Свойство CustomData, даващо достъп до обекта, представящ колекцията – масив с допълнителните данни;
- Метод Defragment, изпълняващ операция по свиване на файла.

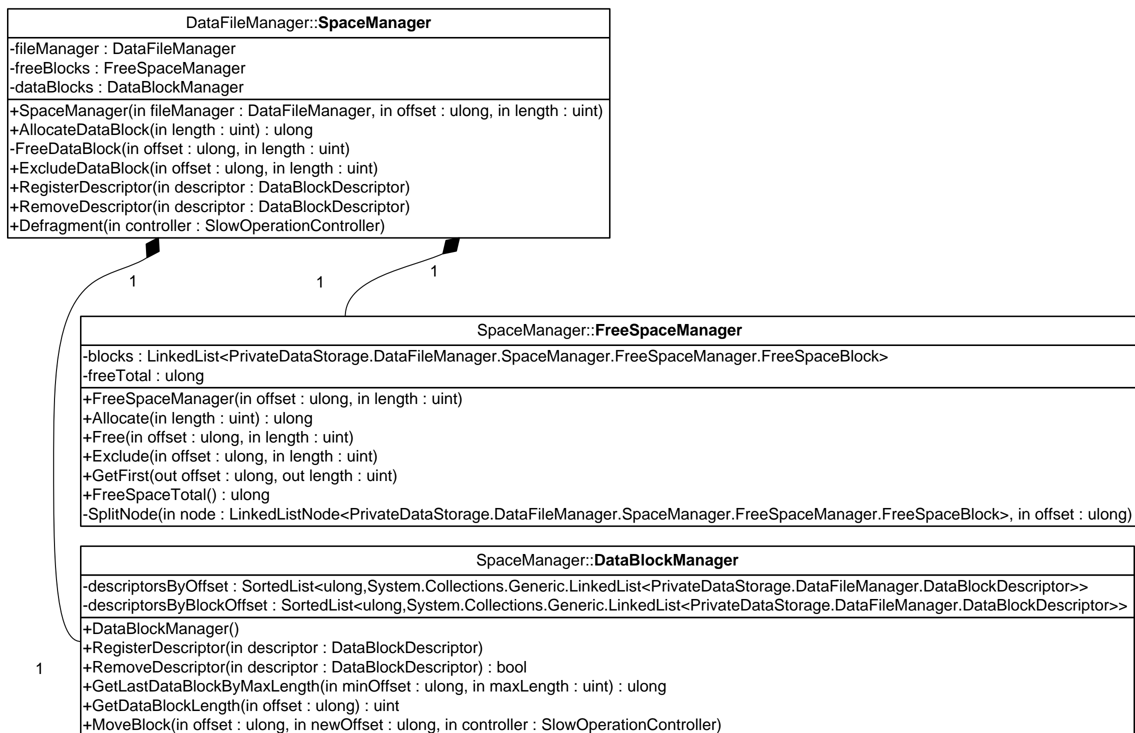
4.5.1.2. Частни методи и свойства

Частните методи и свойства на DataFileManager, както казахме по-горе, дават достъп на дескрипторите и модула за управление на пространството достъп до вътрешна функционалност за работа с файла. Те включват:

- AllocateDataBlock, позволяващ заделяне на място за съхранение във файла на блок с определена дължина. Методът делегира задачата по намирането на подходящ блок във файла на модула за управление на пространството. В случай, че той не може да намери подходящ блок, AllocateDataBlock връща отместване в края на файла, за да бъде той удължен при записа на данните;
- ExcludeDataBlock – изключва блок с данни от списъка със свободно пространство във файла (използва се по време на начално зареждане на файла). Делегира изпълнението на модула за управление на пространството.

- RegisterDescriptor – регистрира дескриптор. Делегира изпълнението на модула за управление на пространството.
- RemoveDescriptor – премахва регистрацията на дескриптор. Делегира изпълнението на модула за управление на пространството.

4.5.2. SpaceManager



Фигура 6: SpaceManager и неговите под-модули

Класът SpaceManager реализира функционалността на модула за управление на пространството, описан в точка 3.3.4.1. Основната му функционалност е реализирана в двата му подмодула – за управление на свободното пространство и за управление на блоковете с данни (Фигура 6).

4.5.2.1. *Свиване на файла*

Единственият метод на SpaceManager, притежаващ някаква по-значима функционалност, е Defragment, който реализира операцията по свиване на файла. За целта се използва прост ал-

горитъм, при който се върви от началото към края на файла, като първият намерен свободен блок се запълва от последния блок с данни, събиращ се в него. Ако никой блок с данни, следващ свободния блок, не се събира в него, блокът се разширява чрез преместване на следващия го блок с данни (в друг свободен блок, ако това е възможно, или ако не – след края на файла), и повтаряне на операцията. Чрез описаната процедура свободното пространство постепенно се премества назад, а блоковете с данни – напред във файла. В края на операцията файлът се отрязва до реално използваната от него дължина.

4.5.3. FreeSpaceManager

Класът FreeSpaceManager реализира функционалността на под-модула за управление на свободното пространство, описан в точка 3.3.4.2. Класът поддържа списък на свободните блокове, като предлага операции за заделяне на блок (Allocate), освобождаване на блок (Free), както и изключване на блок от списъка на свободните блокове (Exclude; използва се по време на начално зареждане на файла).

Предлагат се и методи и свойства, използвани по време на операция за свиване на файла – метод GetFirst, връщащ първия свободен блок, ако има такъв, и свойството FreeSpaceTotal, връщащо сумарната дължина на всички свободни блокове.

Частният методът SplitNode се използва при операциите за заделяне на пространство и за изключване на блок от списъка на свободното пространство. Операцията разцепва свободен блок на две, като в резултат блокът се скъсява, а след него в списъка се появява нов блок, съдържащ остатъка от мястото му.

4.5.3.1. FreeSpaceBlock

Класът FreeSpaceBlock, вътрешен за FreeSpaceManager, представя описанието на един свободен блок. Предлага се метод Split,

позволяващ разцепването на блока на зададено отместване, в резултат на което се получават два нови блока.

4.5.4. DataBlockManager

Класът DataBlockManager реализира функционалността на под-модула за управление на блоковете с данни, описан в точка3.3.4.3. Класът поддържа списъци на дескрипторите, сортирани отместване, както и по отместване на блока, към който сочат.

Предлаганите от модула операции включват:

- Следене на дескрипторите:
    - RegisterDescriptor – регистрира дескриптор (както бе описано в точка 3.3.4 – „Управление на пространството”);
    - RemoveDescriptor – премахва регистрацията на дескриптор. Връща булева стойност, показваща дали премахнатият дескриптор е бил последният, сочещ към съответния блок с данни (за да може модулът за управление на пространството да маркира блока като свободен);
- Методи, подпомагащи операцията за свиване на файла:
    - GetLastDataBlockByMaxLength – връща последния блок с данни, по-малък от зададен размер, и намиращ се след определено посочено отместване във файла;
    - GetDataBlockLength – връща дължината на блока с данни, намиращ се на зададено отместване;
    - MoveBlock – премества блок с данни на ново отместване във файла. За целта се намират дескрипторите, сочещи към блока, използва се методът MoveBlockTo на някой от тях, за да се премести блока, след което

дескрипторите, сочещи към него, се пренасочват към новото му местоположение. Обектите на дескрипторите, намиращи се вътре в блока, пък, се уведомяват, за да запомнят новото си местоположение във файла.

4.6. Публичен интерфейс

Публичният интерфейс на компонента, както описахме в точка 3.3.6, представлява клас, предлагащ публични методи, които приложението, използващо компонента, може да използва за работа с него. Приложението може, стига да има такава нужда, да създаде повече от един обекта на класа, което би му позволило да работи с повече от един файла с данни едновременно.

4.6.1. DataStorage

```
DataStorage
-dataPath : string
-externalItemFolder : string
-dataFileExtension : string = ".pds"
-externalSubfolderSuffix : string = " Files"
-externalExtension : string = ".pde"
-fileManager : DataFileManager
+CreateStorage(in dataPath : string) : DataStorage
+DataStorage(in dataPath : string)
-~DataStorage()
+Close()
+GetItemIDs() : ItemID[]
+GetCustomData(in type : int) : Content
+StoreCustomData(in type : int, in content : Content)
+GetMetadata(in id : ItemID) : EncryptedContent
+GetContent(in id : ItemID) : EncryptedContent
+StoreMetadata(in id : ItemID, in encryptedMetadata : EncryptedContent, in controller : SlowOperationController)
+StoreItem(in id : ItemID, in encryptedMetadata : EncryptedContent, in encryptedContent : EncryptedContent, in isFile : bool, in controller : SlowOperationController)
+DeleteItem(in id : ItemID)
+Defragment(in controller : SlowOperationController)
-ExternalFolderFromDataPath(in dataPath : string) : string
-ExternalFilePathFromID(in id : ItemID) : string
```

Фигура 7: Клас DataStorage

Публичният интерфейс на компонента за съхранение на данните, който разработваме, се реализира чрез класа DataStorage (Фигура 7). Предлаганите от него методи включват:

- Методи за работа с файла:
  - CreateStorage – статичен метод, създаващ и отварящ нов файл с данни. Връща обект на класа DataStorage;
  - Конструктор, приемащ път към файл – отваря вече съществуващ файл с данни;

  - Close – затваря файла;
  - Defragment – изпълнява операция по свиване на файла;
- Методи за работа с данните:
  - GetItemIDs – връща масив с идентификаторите (обекти от типа ItemID) на всички предмети, съхранени във файла;
  - GetMetadata – връща мета-данните (обект от тип EncryptedContent) за предмет със зададен идентификатор;
  - GetContent – връща данните (обект от тип EncryptedContent) за предмет със зададен идентификатор;
  - StoreMetadata – съхранява нови мета-данни за съществуващ предмет по зададен идентификатор;
  - StoreItem – съхранява нови мета-данни и данни за съществуващ или нов предмет по зададен идентификатор. Приема се и булев параметър, указващ дали съдържанието на предмета идва от външен файл (картинка, файл, и т.н.), или е въведен от потребителя през потребителския интерфейс на приложението. Този флаг се използва, за да се прецени дали съдържанието на предмета да бъде съхранено извън основния файл с данни;
  - DeleteItem – изтрива предмет със зададен идентификатор;
- Методи за работа с допълнителни данни на приложението:
  - GetCustomData – връща допълнителните данни (обект от тип Content), съхранени в даден слот. Ако данните са криптирани, връщаният обект е всъщност

от наследения тип EncryptedContent, носещ в себе си и инициализационния вектор за декриптиране на данните;

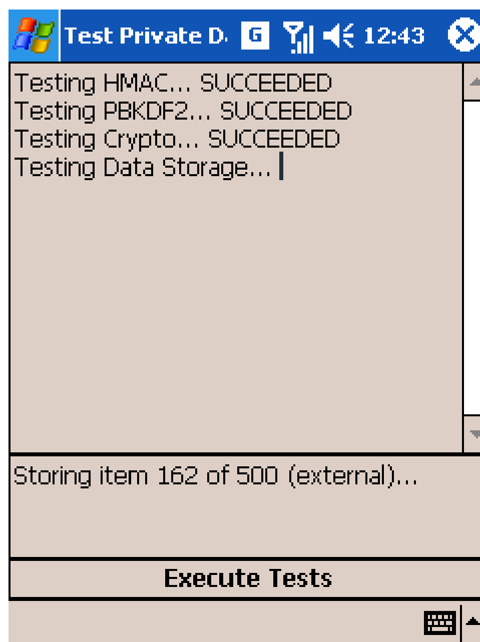
- StoreCustomData – съхранява допълнителни данни в даден слот. Допълнителните данни могат да бъдат от тип Content, в случай, че не са криптирани, или от наследения му тип EncryptedContent за криптирани данни.

5. Тестване на компонента

Тъй като компонентът за съхранение на данните, който разработваме като цел на настоящата дипломна работа, е предназначен да бъде използван от други приложения и не разполага с потребителски интерфейс, за тестването му използвахме специализирано тестово приложение (отделно, разбира се, от реалното тестване на компонента чрез използване на приложението, разработено от Милена Йорданова, за чиято цел компонентът е създаден).

5.1. Описание на тестовата инфраструктура

Тестовото приложение, което разработихме съвместно с Милена Йорданова, бе използвано както за тестване на основната криптографска функционалност на приложението, разработвано като цел на нейната дипломна работа (Йорданова, 2007), така и за тестване на функционалността на компонента за съхранение на данните. Приложението позволява изпълнението на серия автоматизирани тестове на компонентите, визуализирайки информация за прогреса на тестването (виж Фигура 8).



Фигура 8: Автоматизирано тестване на компонента

Отделните тестове на тестовото приложение са разработени като отделни класове, предлагащи свойства, връщащи името на теста, както и метод ExecuteTest за изпълнение на съответния тест. На метода ExecuteTest се подава делегат към метод, който тестовият метод може да използва, за да визуализира различна информация за прогреса на теста (долу на Фигура 8).

Тестовете на компонента за съхранение на данните представляват просто един такъв тестов клас за тестовото приложение.

5.2. Тестване на функционалността

5.2.1. Произволни данни и операции

Тестът на функционалността на компонента започва със създаването на файл с данни чрез метода CreateStorage на компонента. Файлът се създава в текущата папка на устройството (което по принцип е „My Documents”), с име, съдържащо в себе си датата и часа на началото на теста (за да се избегне презаписването или повторното използване на вече съществуващ тестов файл).

Тестовият клас след това генерира предварително определен брой (използвахме числото 10) тестови допълнителни данни, с произволно съдържание, дължина, и слот, в който да бъдат записани във файла. В слотовете може да има повторения, за да се тества функционалността на компонента за замяна на допълнителните данни, намиращи се в даден слот, с други. Така генерираните произволни допълнителни данни се съхраняват последователно в компонента за съхранение на данни, като тестовият компонент си отбелязва какво би следвало да съдържат различните слотове, за да може след това да провери дали данните, съхранени от компонента, отговарят на очакваното.

Тестовият клас след това генерира предварително определен брой (използвахме числото 500) тестови предмета, състоящи се от произволни данни, мета-данни и инициализационни вектори, всяко от тях – с произволна, но реалистична, дължина. Някаква произволна част от тестовите предмети се маркират като „идващи от външен файл”, за да ги съхрани компонентът извън основния файл с данни[8]. Всички тези произволно генерирани предмети се съхраняват в компонента за съхранение на данните.

След като тестовите предмети са съхранени, тестът продължава с произволни операции по съхранение на нови предмети, модификации на съществуващи предмети, както и изтриване на предмети. Всяка от тези три операции се генерира произволно, с еднакъв шанс. Броят на изпълнените операции е произволен. По време на изпълнението им, тестовият код си отбелязва какви модификации, изтривания и добавяния са правени, за да може след това да провери дали те са изпълнени правилно.

[8] Използвахме максимална дължина от 20 байта за инициализационните вектори, 150 байта за мета-данните, 500 байта за съдържанието на предмети, маркирани като въведени от потребителя, и 10 килобайта за съдържанието на предмети, маркирани като идващи от външен файл. Използвахме съотношение от 2:1 на вътрешните към външните предмети. Използвахме и $\frac{1}{20}$ шанс за „погрешно” маркиране на предмет, т.е. за съхранението на голям предмет вътре в основния файл и обратно – за съхранение на малък предмет извън него.

Когато произволните операции приключат, започва сравнение на данните. Проверява се дали слотовете за допълнителни данни съдържат очакваните данни. Всички тестови предмети се прочитат и сравняват с очакваните стойности. Прави се опит за прочитане и на изтритите предмети, за да се провери, че те са наистина изтрити.

5.2.2. Повторна проверка

Правилното прочитане на данните след произволните операции не означава, че файлът е записан правилно. Съществува възможност структурите, поддържани в паметта на компонента, да сочат правилно към данните, но стойностите на дескрипторите да не са съхранени правилно във файла. Ето защо, след като горната проверка премине успешно, файлът се затваря, след което се отваря наново. По този начин компонентът се принуждава да прочете обратно данните, които е записал, като при този процес всякакви грешки по съхранението на файла излизат наяве.

Ако компонентът успее успешно да отвори файла, се извършва повторна проверка на данните. Отново се проверява дали слотовете за допълнителни данни съдържат очакваните стойности. Всички тестови предмети се прочитат и сравняват с очакваните данни. Проверява се дали изтритите предмети все още липсват.

5.2.3. Тестване свиването на файла

Ако повторната проверка на данните премине успешно, се преминава към тестване на операцията за свиване на файла. За целта просто се изпълнява операция за свиване. Изтриването и промяната на предмети при произволните операции, изпълнени на по-ранният стадий на теста, гарантират, че файлът ще съдържа някакво свободно място, което трябва да бъде премахнато.

За да се уверим, че операцията по свиване на файла не презаписва грешно блокове с данни, поддържа дескрипторите правилно насочени, и т.н., в случай, че тя завърши успешно, се изпълнява нова проверка на целостта на данните. Отново се сравняват всички данни и допълнителни данни, проверяват се и всички изтрити предмети.

5.2.4. Повторна проверка след свиването

Както отбелязахме по-горе, успешното прочитане на данните не означава, че дескрипторите във файла съдържат коректни стойности; то просто показва, че структурите в паметта са наред. Затова, ако данните изглеждат както трябва след свиването на файла, файлът се затваря и отваря пак, за да се принуди отново компонентът за съхранение на данните да прочете наново всички структури. След това всички данни, допълнителни данни и изтрити предмети се сравняват за пореден път. Ако и тази проверка успее, можем да сме доста сигурни, че компонентът за съхранение на данните работи правилно. С това тестът приключва успешно.

5.3. Тестване на производителността

За тестване на производителността на компонента, когато от това има нужда, автоматизираният тест, описан по-горе, съдържа специален код, измерващ времето за изпълнение на операциите (използвахме кода, за да измерваме специално времето за съхранение на предмет в компонента, както и дали и по какъв начин това време деградира с увеличение на броя на съхранените във файла предмети).

За правилно измерване на производителността на дадено приложение, предложените от системата функции GetTickCount или обектите, връщащи текущата дата и време, са недостатъчно прецизни. Въпреки, че тези някои от тези функции измерват времето в милисекунди, всъщност точността им е много по-

малка. За правилно измерване е необходимо да се използват специалните броячи за производителността на системата (performance counters).

За целта използвахме класа Timer, предложен в статията „Developing Well Performing .NET Compact Framework Applications" (Fox & Box, 2003), извикващ чрез технологията Platform Invoke системните функции QueryPerformanceCounter и QueryPerformanceFrequency и използващ ги за правилно измерване на интервали от време в милисекунди.

В теста на компонента, ако се измерва времето за изпълнение, се създава текстов файл, който се използва за запис на данните, получени от измерванията. Този файл се създава на първата карта за разширение на паметта, налична на устройството[9]. По този начин, при изпълнение на тестовете на емулатор на устройството, можехме да се възползваме от функционалността на емулатора да представя определена папка на компютъра, на който емулаторът е стартиран, като разширителна карта памет на устройството. По този начин файлът с измерените стойности се записва директно на компютъра, използван за стартиране на емулатора, което прави удобно тяхното извличане.

[9] За намиране на пътя към картата с памет използвахме техниката, описана в статията „Working with files on Smartphone devices with the .NET Compact Framework", която може да бъде намерена в MSDN (Foot, 2004).

6. Изводи и възможности за подобрение

Пред приложение, което трябва да съхрани данните си, стои изборът дали да използва вече съществуващо решение за тяхното съхранение, като например файлова система или релационна база данни, или пък да разработи свой собствен нов файлов формат и механизъм за съхранение, който да се задоволява конкретните му нужди.

Както видяхме в настоящата дипломна работа, разработката на собствен файлов формат предлага много предизвикателства. Трябва да бъдат избрани подходящи примитиви и структури за представянето на данните във файла, които да позволяват бързи и надеждни операции с данните. Трябва да бъде избрана подходяща стратегия за заделяне на свободното пространство. При нужда, трябва да бъде реализирана операция за свиване на файла с данните, което не е никак тривиално. Представянето на файловите структури в паметта за бърза работа с тях и същевременното им поддържане в синхрон с файла също заслужава специално внимание. Предизвикателствата са много, но успешното им решаване се възнаграждава от бърз и надежден механизъм за съхранение на данните, който отговаря на конкретните изисквания на използващото го приложение.

Компонентът, който разработихме, вероятно отстъпва по характеристики на съвременна релационна база данни. На стратегиите за заделяне на памет би могло да се отдели още много внимание, тъй като то е област, в която нови има място за много нови разработки и подобрения. Свиването на файла също представлява област за подходящо бъдещо развитие.

Въпреки посочените предизвикателства, можем да кажем, че резултатът от настоящата дипломна работа задоволява поставените цели. Разработихме файлов формат, подходящ за съхране-

нието на криптирани данни и проектирахме компонент, позволяващ бързото им и надеждно съхранение в този файлов формат. Файловият формат и компонентът са съобразени с изискванията на използващото ги приложение. Компонентът се държи добре при автоматизирано тестване, както и при реална практическа употреба.

7. Заключение

В съвременното общество всеки има нужда да може да носи различни лични данни със себе си, като същевременно тяхното евентуално попадане в чужди ръце би имало неприятни последствия. Хората все повече и повече използват мобилни телефони, джобни компютри и други преносими устройства, за да имат достъп до личната си информация навсякъде и по всяко време. Необходима е разработката на приложения, съхраняващи данните на потребителя сигурно в криптиран вид. Тези приложения, обаче, имат нужда от подходящ механизъм за съхранение на криптираната информация.

В настоящата дипломна работа внимателно анализирахме нуждите за съхранение на информация на приложение за съхранение на криптирани данни в Pocket PC. На базата на извършените анализи, и след като разгледахме различни стратегии за съхранение на данните, проектирахме подходящ файлов формат, който да отговаря на нуждите на приложението за бързо и надеждно съхранение на криптирана информация. Разработихме компонент, предлагащ на приложението функционалността на файловия формат, като в процеса разгледахме различни стратегии за заделяне на свободното пространство и преодоляхме различни архитектурни предизвикателства.

Резултатът от разработката е компонент, който, макар и да притежава области за възможно бъдещо развитие, задоволява нуждите на използващото го приложение като предлага бърз, надежден, и съобразен с ограничените изчислителни ресурси механизъм за съхранение на криптирани данни.

Използвана литература

Foot, P. (март 2004 г.). *Working with files on Smartphone devices with the .NET Compact Framework.* Изтеглено на 16 декември 2006 г. от MSDN: http://msdn2.microsoft.com/en-us/library/aa446567.aspx

Fox, D., & Box, J. (декември 2003 г.). *Developing Well Performing .NET Compact Framework Applications.* Изтеглено на 16 декември 2006 г. от MSDN: http://msdn2.microsoft.com/en-us/library/aa446542.aspx

Jansen, A. (15 март 2006 г.). *How to Programmatically Improve File System Throughput*. Изтеглено на 16 декември 2006 г. от Windows CE Base Team Blog: http://blogs.msdn.com/ce_base/archive/2006/03/15/IncreaseFSThroughput.aspx

Jensen, C. (1994). *Fragmentation: the Condition, the Cause, the CURE.* Executive Software International.

Microsoft Corporation. (2006). *SQL Server 2005 Compact Edition Features Datasheet.* Изтеглено на 5 февруари 2007 г. от Microsoft Corporation web site: http://download.microsoft.com/download/7/f/c/7fc20778-4e2e-4944-b432-ed74b404e542/sqlservercompactdatasheet_final.doc

Newcomer, J. M. (21 август 2006 г.). *About Memory Allocators*. Изтеглено на 16 декември 2006 г. от FlounderCraft Ltd. web site: http://www.flounder.com/memory_allocation.htm

Wikipedia. (2006). *Dynamic memory allocation.* Изтеглено на 16 декември 2006 г. от Wikipedia: http://en.wikipedia.org/wiki/Dynamic_memory_allocation

Йорданова, М. Р. (2007). *Приложение за Pocket PC, позволяващо сигурно съхранение на лични данни.* София: Софийски Университет "Св. Климент Охридски".

Приложение 1 – Списък на фигурите

Фигура 1: Сравнение на производителността на последователен и непоследователен файл ____ 23
Фигура 2: Свиване на файла в Microsoft® Office Outlook® ____ 25
Фигура 3: Обща архитектура на компонента ____ 35
Фигура 4: Класове на Private Data Common ____ 68
Фигура 5: Йерархия на дескрипторите ____ 75
Фигура 6: SpaceManager и неговите под-модули ____ 87
Фигура 7: Клас DataStorage ____ 90
Фигура 8: Автоматизирано тестване на компонента ____ 94

Приложение 2 – Списък на таблиците

Таблица 1: Сравнение на наличните структури от данни ______ 44