

© Аврам Моис Ескенази, Нели Милчева Ескеиази, 2001
© Михаил Асенов Руев, *корица*, 2001
ISBN 954-426-311-X

СЪДЪРЖАНИЕ

УВОД.....	5
1. ОСНОВНИ ПОНЯТИЯ	6
1.1. Програми и програмни продукти	6
1.2. Характеристики на софтуера	7
1.3 Софтуерни технологии.....	10
2. ЖИЗНЕН ЦИКЪЛ НА ПРОГРАМНИЯ ПРОДУКТ	14
2.1. Моделиране на жизнения цикъл	14
2.2. Класификация на моделите на ЖЦ на ПП	15
2.3. Типове модели на ЖЦ на ПП	16
2.4. Модел на Гънтър	24
3. КОНВЕНЦИОНАЛЕН ПОДХОД ЗА РАЗРАБОТВАНЕ НА СОФТУЕР	31
3.1. Модели на жизнения цикъл и подходи за разработване	31
3.2. Определяне на софтуерната система	32
3.3 Проектиране.....	35
3.4. Програмиране	40
3.5. Интегриране и тестване	40
4. ОБЕКТНО ОРИЕНТИРАН ПОДХОД ЗА РАЗРАБОТВАНЕ НА СОФТУЕР	43
4.1. Основни понятия.....	43
4.2. Обектно ориентиран анализ и проектиране	44
4.3. Обектно ориентирано програмиране и тестване	45
4.4. Управление на обектно ориентираното разработване	46
4.5. Въвеждане на обектно ориентирания подход	49
5. ДЕЙНОСТИ, ОСИГУРЯВАЩИ РАЗРАБОТВАНЕТО НА ПРОГРАМНИ ПРОДУКТИ	51
5.1. Откриване на дефекти	51
5.2. Съпровождане	58
5.3. Документиране	61
6. КАЧЕСТВО НА СОФТУЕРА	64
6.1. Общи понятия	64
6.2. Модели на качеството на софтуера	65
7. СОФТУЕРНИ МЕТРИКИ	76
7.1. Въведение	76
7.2. Измерването в софтуерното производство	76
7.3. Класификация на софтуерните метрики	77
7.4. Примери за софтуерни метрики.....	80
7.5. Методологични проблеми на използването на софтуерните метрики	84

8. УПРАВЛЕНИЕ НА КАЧЕСТВОТО	91
8.1. Проблемът за управление на качеството	91
8.2. Компоненти на програмата за осигуряване на качеството	92
8.3. Оценяване на софтуерните процеси	98
8.4. Стандарти за качеството на софтуера	105
9. ОРГАНИЗАЦИОННО УПРАВЛЕНИЕ.....	109
9.1. Въведение	109
9.2. Основни понятия	109
9.3. Управление на софтуерни проекти.....	110
9.4. Планиране	112
9.5. Анализ и управление на риска	113
9.6. Организация на софтуерните проекти.....	114
9.7. Примерен план на софтуерен проект	117
9.8. Правила за осъществяване на софтуерни проекти	118
10. ЦЕНА НА СОФТУЕРА.....	121
10.1. Необходимост и цели	121
10.2. Критерии	121
10.3. Моделът на Боем СОСОМО	124
10.4. Метод на функционалните точки	129
10.5. Други модели	134
10.6. Оценяване при обектна ориентираност	136
11. МАРКЕТИНГ НА СОФТУЕРА	138
11.1. Общи съображения	138
11.2. Определения за маркетинг	138
11.3. Основна маркетингова концепция	139
11.4. Маркетингова стратегия.....	139
11.5. Пазарен жизнен цикъл	143
11.6. Позициониране и марка	145
11.7. Моделиране на софтуерния пазар	146
12. АВТОМАТИЗАЦИЯ НА СОФТУЕРНОТО ПРОИЗВОДСТВО	150
12.1. Автоматизация чрез индивидуални средства	150
12.2. Автоматизация чрез интегрирани среди	151
13. СЪВРЕМЕННИ ТЕХНИКИ ЗА ПРОГРАМИРАНЕ И РАЗРАБОТВАНЕ НА СОФТУЕР	159
13.1. Програмиране, осигуряващо надеждно функциониране на програмните системи	159
13.2. Разработване с прототипиране	161
13.3. Повторно използване в софтуерното производство („Re-use“)	163
13.4. Разработване на софтуер с участието на потребителя.....	166
14. ЧОВЕШКИЯТ ФАКТОР В РАЗРАБОТВАНЕТО И ИЗПОЛЗВАНЕТО НА СОФТУЕР	169
14.1. Как да се наемат най-добрите софтуерни специалисти	169
14.2. Определяне на структурата и състава на работните групи	172
14.3. Управление на взаимоотношенията в работната група	173
14.4. Организиране на комуникациите в групата	175
14.5. Организиране на сбирки и заседания на групата	176
14.6. Ергономика	176
14.7. Професионализъм и етично поведение	177
15. МАЛКАТА СОФТУЕРНА ФИРМА	180
15.1. Еволюция на малката софтуерна фирма	180
15.2. Екстремно програмиране	186
15.3. Правни проблеми	189

УВОД

Предмет на настоящата книга е това, което на английски език се нарича *Software Engineering* вече около три десетилетия. Преводът на български не може да бъде буквален, най-малкото защото инженерство и *Engineering* смислово не се покриват. Немското название — *Software Entwicklung*, или пък френското — *Genie Logiciel*, могат да ни насочат, но не и да ни дадат крайното решение. Задачата се усложнява и от разнообразието от значения, които различните автори влагат в термина. Теоретикът е склонен да остава около езиците за спецификация, верификацията и свързаните с тях формализми, практикът — да търси годни за прилагане модели на жизнения цикъл, ефективни методи за осигуряване на качество, за правилно планиране и организиране на процеса на разработка, ръководителят стига в интересите си до проблемите на софтуерния маркетинг и методите за оценяване на необходимите ресурси. Тези и много други съображения са ни довели (след 2—3 по-несолучливи опита) до българския термин *софтуерни технологии*.

Първият у нас курс по софтуерни технологии беше подгответ и четен от Аврам Ескенази и Владимир Занев в продължение на три години, започвайки от №84/85 учебна година, за студентите от IV и V курс на Факултета по математика и информатика на Софийския университет. От 1987/88 учебна година след известна преработка той стана задължителен за студентите в това учебно заведение и в четенето на лекциите и упражненията се включиха Нели Манева и Валя Петрова. С известни модификации този курс продължава и до днес да се чете там. Междувременно най-напред в Икономическия университет — Варна (с лектор Георги Зеленков), а по-късно в Пловдивския университет, International University, Нов български университет и на други места започнаха и продължават лекции по този предмет, макар и невинаги под същото име, а понякога и в рамките на повече от един курс.

Невъзможно е една книга да обхване целия предмет, още по-малко възможно е той да се прочете в рамките на един курс. Като пример ще посочим, че в някои университети в САЩ (Тексаския университет, университета в Сиатъл и др.) обучението по софтуерни технологии продължава две години в рамките на 6—8 задължителни и още толкова избираеми курса, практикум и 2—3 курсови проекта. Съответна на тази констатация е и целта на авторите — макар да е предназначена да бъде учебник за студентите по предмета „Софтуерни технологии“, настоящата книга би могла да бъде за тях и за всички практикуващи софтуерни специалисти и ръководители въведение в основополагащите направления на тази изключително динамична и с нарастващо за практиката значение дисциплина.

Главите на тази книга са разработени както следва:

Аврам Ескенази: 1, 2, 6, 8, 10, 11, 15

Нели Манева: 3, 4, 5, 7, 9, 12, 13, 14

1. ОСНОВНИ ПОНЯТИЯ

1.1. Програми и програмни продукти

1.1.1. Хронология

Естественият път за навлизане в тематиката и преди всичко в проблема програми—програмни продукти (прилики и разлики) е хронологичният. Заедно с построяването на първите компютри от средата на 40-те години се появяват и първите програми за тях. Както е известно, предназначението им е било за военни цели. Доколкото в разработката и на хардуера, и на софтуера са участвали учени, естествено е било да се направят опити за използване на компютрите за подпомагане на научните изследвания, преди всичко за изчисления.

Много скоро, още преди края на 50-те години, ръководителите на банките установяват, че банковата дейност би могла да бъде направена по-точна и побързва, ако бъдат приложени компютри. Така и става и до днес банките си остават едни от най-сериозните консуматори на компютърни ресурси. Отдавна е станало ясно (както впрочем и в много други области), че банково дело без компютри е просто немислимо.

Постепенно компютрите навлизат в търговската дейност, в медицината, в дейността на авиолиниите и в много други области. Всяка от тях налага създаването на съответните програми. Компютрите обаче си остават „големи“ (main frame) или се произвеждат с известни вариации нагоре (свръхкомпютри) или надолу (мини). Във всички случаи обаче потребителите остават или професионалисти, или добре обучени специалисти в конкретната област на приложение. Характеристиките на софтуера (интерфейс, надеждност, лекота на усвояване и др.) са следствие от тези особености на потребителите.

В края на 60-те години е извършена и една решаваща за по-нататъшното развитие на софтуерното производство стъпка. Известната фирма IBM, по това време най-големият производител на хардуер и софтуер в света, въвежда т. нар. политика на unbundling. Това означава, че софтуерът, който дотогава се е давал като безплатна (поне така е било обявявано) добавка към хардуера, започва постепенно да се продава. В началото това се е отнасяло само до част от доставения софтуер, постепенно обаче кръгът му се разширява. Какви са били стратегическите помисли на IMB с този техен ход, е предмет на обсъждане дълго време, но едно е ясно — той въщност дава тласък за развитие на софтуерната промишленост.

Когато през 80-те години се появяват персоналните компютри, а през 90-те средствата за глобална комуникация стават потенциално достъпни за всеки, това променя значително изброените характеристики на програмите за компютри, доколкото кръгът от потребители рязко нараства и става изключително разнообразен по образование, квалификация, възраст, интереси и пр.

1.1.2. Проблеми

От така скицираното развитие на програмите за компютри за последните около 50 години лесно могат да се видят няколко най-общи техни характеристики:

— всяка програма би трябвало да може да бъде поправяна, разширявана, подобрявана от своя автор или от друго квалифицирано лице; това е станало очевидно вероятно още на първия месец от експлоатацията на първата компютърна програма;

— първоначално не е било много ясно дали други хора освен автора ще ползват написаната програма; днес, разбира се, този въпрос е немислим — може по-често да се случи обратното — след написването авторът никога да не ползва своето творение;

— последното обаче веднага води до необходимостта от преносимост на програмата от един компютър на друг;

— след като отдавна програмите са станали обект на производствена и търговска дейност, ясно е, че трябва да има начин те да бъдат оценявани, преди да бъдат продадени.

От тези характеристики непосредствено следва, че програмата трябва да може да бъде записана на някакъв технически носител и да бъде придружавана от определени документи, които я описват в различни аспекти.

1.1.3. Определения

След тези подготвителни бележки сме готови да дадем *определение за програма и за програмен продукт*.

Програмата е последователност от инструкции, които, когато бъдат декодирани от компютър (или от компютър и транслираща програма), водят до решаването от страна на компютъра на дадена задача.

Програмният продукт е програма или съвкупност от взаимодействащи програми, записани върху технически носител и придвижени от съответна документация.

Веднага възниква въпросът за все по-често използваната дума *софтуер*. Ще смятаме термините *софтуерен продукт* и *програмен продукт* за еквивалентни по смисъл.

Думата *софтуер* според [1] се употребява за обозначаване на съвкупност от взаимодействащи програми и в този смисъл се доближава донякъде до понятието *програмен продукт*.

Друго определение, което по общо виждане се доближава още повече до понятието *програмен продукт*, е формулирано през 1983 г. от голямата професионална американска асоциация на електронните инженери *IEEE* и гласи, че *софтуерът* — това са компютърни програми, процедури, правила и евентуално придружаваща документация, както и данни, отнасящи" се до функционирането на компютърната система.

Днес най-голяма гражданска общественост е придобило разбирането за *софтуера като общо понятие за програми и/или програмни продукти*.

1.2. Характеристики на софтуера

За да достигнем до предмета на софтуерните технологии, ще разгледаме няколко най-общи характеристики особености на софтуера. Редът на разглеждането им не е свързан с тяхната значимост.

1.2.1. Необходими ресурси

Софтуерните продукти в сравнение с много други се отличават невероятно много по отношение на влаганите в тях ресурси, откъдето следва и цената им. Знае се, че един нов автомобил може да струва примерно от 5 000 до 200 хил. щатски долара, а ако се разгледат 95% от продаваните автомобили, този интервал ще се свие значително. За сравнение ще посочим софтуерни продукти около двата края на скалата.

— По отношение на използвания ресурс и от време, и хора (а оттам и цена) на горния край се намират софтуерните проекти, свързани с американската космическа

програма — „Аполо“, „Скайлаб“, „Совалката“; те са изисквали труда на 700 души в продължение на 7 години; подобен по мащаб е проектът за управление на въздушния трафик на САЩ, който е бил разработан в продължение на 5 години от 500 души. Без да разполагаме с точни данни за цената на тези продукти, веднага може да се оцени, че и в двета случая тя е към милиард долара.

— По отношение на мащаба и сложността може да споменем (отново в горния край на скалата) една от резервационните системи в САЩ, която преди повече от десет години при създаването си вече е била обслужвана от 13 500 свързани терминала и е обработвала средно по 10 600 000 транзакции *на ден*.

— За сравнение можем да посочим пристрастна (но вършеща работа) информационна система за персонален компютър, която би могла да се разработи от един опитен програмист за един ден и следователно да струва примерно 100 долара.

• — Що се отнася до пример по отношение на сложност, на долния край на скалата би могла да се посочи програма за пресмятане на данък или пък прост калкулатор.

1.2.2. Абстрактност на софтуера

Софтуерният продукт не може да бъде почувствуан с нито едно от петте човешки сетива. С компютъра това не е така — той може най-малкото да бъде видян или пипнат. Разбира се, написаните команди на първичния текст на една програма могат да бъдат видени, но логическата същност на програмата, особено ако е по-голяма, няма как да бъде „видяна“, още по-малко — цялостната структура на комплекс от програми. Това е смисълът на твърдението, че софтуерът е на много високо ниво на абстрактност.

1.2.3. Уникалност на софтуерното производство

Много често хората са склонни да мислят себе си или нещата, с които се занимават, за особени. Но софтуерното производство наистина има черти, които го правят уникално. Да се върнем към сравнението с автомобилното производство:

— основните усилия при производството на програмен продукт са преди появата на първото работещо копие, докато производството на всеки автомобил (без да забравяме значителните ресурси, необходими за проектирането на дадения модел и настройката на производствените линии) изиска влагането на немалко труд, енергия, материали;

— ако има *грешка* в даден програмен продукт (а безгрешен софтуер, както е известно, няма), то тя се отнася до всички негови копия; при автомобилите този случай е твърде рядък или поне не е типичен — там дефектите могат да бъде разнообразни, но, общо взето, са индивидуални — за всеки конкретен автомобил;

— когато дойде време програмният продукт да *излезе от употреба*, все повече това става относително едновременно и поради една и съща причина — обикновено идва нова версия, от една страна, с някоя и друга възможност повече и най-вече — съобразена с променения хардуер; при автомобилите и причините за „умиранието“ на даден екземпляр са индивидуални (катастрофа, повреди, възможности на собственика и пр.) и времето на живот се отличава от екземпляр към екземпляр значително.

1.2.4. Мултидисциплинарност на разработването на софтуер

Отделните програмни продукти са изключително разнообразни по предназначение. Като изключим специфичните инструментални средства (компилатори, свързващи редактори, средства за тестване на програми), операционните системи и още малък брой типове софтуер, разработван изключително от софтуеристи, във всички останали случаи не е възможно програмният продукт да бъде създаден само от програмисти. Когато се прави счетоводен софтуер, трябва задължително да участва експерт по счетоводство, ако

се разработва статистически пакет, не може да се мине без математици статистици, за медицинския софтуер са необходими лекари, а при много видове програмни продукти трябва да се впрегнат заедно усилията на много повече групи специалисти. При такова сътрудничество почти винаги възникват проблеми на общуването, породени не толкова от личните особености на участниците, колкото от различния им тип на мислене и различните професионални езици, на които те говорят. Не е случайно, че в последните години в комплексите от курсове по софтуерни технологии много често има курс, занимаваш се с проблемите на общуването в рамките на колектива, разработващ програмен продукт. При това специфични проблеми възникват както пред ръководителите на различни нива на разработващия екип, така и между редовите изпълнители.

1.2.5. Специфични проблеми на надеждността

Вече беше споменат известният на всички програмисти факт, че абсолютно безпогрешен софтуер няма. Разбира се, малка програма с линеен характер и еднообразни входни данни вероятно би могла да бъде проверена докрай. Типичният програмен продукт обаче никога не може да бъде абсолютно гарантиран срещу грешки. Причината за това е известна — прекалено много са възможните пътища през програмата, допустимите (и недопустими) съвкупности от данни, да не говорим за действията от страна на потребителя, които наистина не могат да бъдат изцяло предвидени. Следователно, дори да разполага с много време, пари и други ресурси, разработчикът едва ли би бил в състояние да докаже абсолютна липса на грешки в предлагания програмен продукт. Естествено, съществуват чисто теоретически методи, гарантиращи пълна безпогрешност на програмата, но те са засега приложими само към прекалено тривиални програми. Има разработени и технологии за тестване и отстраняване на грешки, които обаче поне от теоретическа гледна точка не могат да гарантират пълна липса на грешки. Да не говорим за иначе много добрата в други отношения обектно-ориентирана технология, при която за съжаление все още няма и теоретически възможности за доказване на безпогрешност на програмите.

Няма да се спирате на отдавна станалите анекdotични примери за програмата, която работила безпогрешно х години и изведнъж сгрешила (което е напълно възможно), или за онази 0 вместо 1 в програмата, управляваща космическа ракета, довела до преждевременно прекратяване на полета. Ежедневно сме свидетели на програмни продукти, разработени и продавани в стотици хиляди и милиони екземпляри от световни фирми, в които изскуча по някоя грешка.

От друга страна обаче, ако за дадена програма се докаже по един или друг начин, че не прави грешки поне при определени условия, ясно е, че нито едно копие на тази програма никога няма да направи грешка при тези условия.

1.2.6. Рискове

Производството на софтуер в много повече случаи, отколкото това става в по-стари и утвърдени производства, може да доведе до по-малка или по-голяма загуба. Не са малко случаите, когато потребители съдят производителя на поръчания от тях софтуер и успяват да получат огромни суми за *неизпълнени изисквания* от доставения програмен продукт. Често се цитира като пример Голяма хардуерно-софтуерна компания, която преди години трявало да реализира програмна система за резервация на самолетни билети. Едно от изискванията било свързано с максималното време на отговор при извършването на транзакция в реално време. Не са известни подробните, но поради това, че в много малък брой случаи специфицираното в заданието време е било просрочвано, фирмата производител е била осъдена да плати на клиента обезщетение от десетки милиони долари.

Друг обичаен рисков фактор е *срокът на доставка*. Все още не е възможно да се планира напълно сигурно крайният срок на доставка на разработван програмен продукт поради — да го кажем най-общо — огромния брой определящи фактори, за част от които няма обективни измерители, а се разчита на експертното мнение на опитни специалисти и ръководители. Неслучайно опитът тук е от решаващо значение, защото е добре изследван и описан т. нар. 90% синдром — неопитните софтуеристи (ръководители, проектанти, програмисти) планират и работят (несъзнателно) така, че обикновено 90% от програмния продукт се завършват за определен (обикновено точно определен) период от време, а за останалите 10% се оказва, че е необходимо в най-добрая случай още толкова време.

1.2.7. Софтуерът — средство, а не цел

Софтуерът е функция от трети ред. Това означава, че софтуерът е средство, което задейства (управлява) някаква система, а тя самата довежда до искания резултат. Потребителят иска да получи документ в определена форма и с определено съдържание, което става с помощта на компютъра, задействан от текстообработваща програма. В случая документът е цел от първи ред. Компютърът е средството, с което се достига до резултата, и поради това е от втори ред, а текстообработващата програма е от трети ред.

Смисълът на тази малко абстрактна конструкция е в това софтуеристите да помнят винаги, че това, което създават, е всъщност само средство за постигане на някаква цел. От доста време вече има трудове, посветени на психологията на т. нар. „софтуерни фанатици“ и техните приоритети, противоречащи на здравия разум.

1.3. Софтуерни технологии

1.3.1. Терминология

Терминът *Software Engineering* се е появил за първи път на една конференция на НАТО през 1969 година. Както вече беше казано в началото, за най-подходящ еквивалент на български смятаме *софтуерни технологии*, а отхвърляме другия кандидат — софтуерно инженерство. Поучително е да се отбележи, че споменатата конференция е била проведена с цел обсъждане на проблема със *софтуерната криза*. Няколко години преди това са се появили компютрите от т. нар. трето поколение, тяхната мощност, превъзхождаща на порядък тази на компютрите от второ поколение, е изисквала програмни продукти, непознати дотогава по своя мащаб и сложност. Оказалось се е, че в този период не е било ясно как да се произведе такъв софтуер. Така възникава естествената необходимост от специална дисциплина, която да се опита да помогне за преодоляването на софтуерната криза.

1.3.2. Определения

Известни са немалък брой определения на софтуерни технологии. Ето някои от тях:

- Наур през 1969 година определя: установяването и използването на здрави принципи за разработване с цел по икономичен начин да се произведе софтуер, който е надежден и функционира ефективно върху реална машина;
- според дефиницията на IEEE (Асоциацията на американските електронни инженери) софтуерните технологии представляват систематичен подход към разработването, експлоатирането, съпровождането и изваждането от експлоатация на софтуера;

— определението на Феърли от 1984 г. гласи: технологична и мениджърска дисциплина, занимаваща се систематично с производството и съпровождането на софтуерните продукти, които се разработват навреме и на основата на точно определени разходи.

Може да се смята, че второто от тези определения най-много се приближава до съвременното разбиране за същността на софтуерните технологии. Едно допълнение, което смятаме за необходимо, е прилагателното „качествен“ към „софтуер“. Второ, огромното разпространение на търговска основа на програмни продукти е свързано с техния маркетинг. От друга страна, изваждането от експлоатация по важност и по съдържание далече отстъпва на другите елементи на дефиницията. Следователно определението, което ще ползваме оттук нататък, гласи:

Софтуерните технологии представляват систематичен подход към разработването на качествен софтуер, както и към неговото предлагане на пазара, експлоатация и съпровождане.

1.3.3. Цели

При така поставеното условие за *качество* на разработвания софтуер необходимо е да се идентифицират основните му атрибути. Това ще бъде направено тук на най-общо ниво, а ще бъде разгледано подробно в специална глава на тази книга.

Всеки програмен продукт следва да може да бъде *лесно съпровождан*. Това означава, че в него трябва да могат относително лесно да се внасят изменения и подобрения, както и лесно да се отстраняват грешки. За да може това да се осъществява, е необходимо програмният продукт да бъде добре документиран.

Софтуерът трябва да бъде *надежден*. Това означава, че грешките в него трябва да сведени до възможния минимум, а продуктът да изпълнява очакваните от потребителя функции.

Програмният продукт трябва да бъде *ефективен*. Това не означава използване на възможностите на хардуера до границите на допустимото — най-малкото защото такъв софтуер обикновено се съпровожда много трудно. Все пак потребителят очаква оптимално в известен смисъл експлоатиране на ресурсите на компютъра преди всичко по отношение на икономичното използване на оперативната памет и достигането на голяма бързина на изпълнение.

Всеки софтуерен продукт трябва да предоставя *удобен и лесен за усвояване потребителски интерфейс*. Това става все по-актуално поради драстичното нарастване броя на потребителите и разширяването на спектъра им. Не са малко случаите, когато поради недобре проектиран или реализиран интерфейс много от възможностите на продукта остават неизползвани от повечето потребители. Това от своя страна директно води до намаляване броя на продажбите му.

Особено място сред тези атрибути на качеството заема *цената* му. Тук става въпрос за минимизиране разходите по разработването, но и особено по съпровождането, доколкото последните са неочеквано големи.

Не е възможно да бъдат постигнати едновременно най-добри стойности за всички изброени атрибути. Просто защото някои от тях направо си противоречат. Веднага се вижда например, че ако интерфейсът се направи прекалено „дружелюбен“ (естетичен, ергономичен, лесноразбираем, снабден с много помощни указания и пр.), ефективността на програмния продукт ще пострада. Нещата се усложняват и от факта, че цената на подобренията в много случаи не расте линейно, т. е. малки подобрения в определен атрибут могат да изискват значително (дори експоненциално) нарастване на вложените ресурси, следователно и на цената.

Въсъщност, погледнато най-общо, задачата на софтуерните технологии е да покаже как да се произвежда софтуер, който да притежава в оптимално съотношение упоменатите характеристики. Това е смисълът на преодоляването на софтуерната криза, станала причина за появяването на тази дисциплина. Общо е обаче мнението, че софтуерната криза все още не е преодоляна. Независимо от значителните постижения, подобряващи методите и технологиите на разработване на софтуер или довели до създаването на мощни инструментални средства, изглежда, че нуждите от софтуер нарастват по-бързо, отколкото се подобрява производителността на разработчиците на софтуер.

1.3.4. Наука и практика

Както в много други области, и в полето на софтуерните технологии практиците невинаги са склонни да следват незабавно теоретиците. Това е обяснимо, а и „здравословно“. Не всяка теория се оказва толкова полезна, колкото е изглеждала на пръв поглед, а понякога се оказва дори, че не е вярна. От друга страна, пренастройването към един нов метод, език или технология изисква от крайния му потребител (ръководител, проектант, програмист) усилия и време за усвояването му. Много поучително в това отношение е едно изследване, извършено наскоро в САЩ. Анкетираните професионалисти практици са отговорили на въпроси, свързани с приложението на нови методи, предлагани им от науката за софтуерните технологии. Формулирали са отговора си съгласно следната петстепенна скала:

- използвам активно;
- изучавам задълбочено;
- следя литературата;
- очаквам, без да следя;
- не проявявам никакъв интерес.

Тук не са толкова интересни установените от проучването най-прилагани технологии и методи (прототипиране, езици от четвърто поколение, методи и средства за изграждане на графически потребителски интерфейси), колкото тези, които не се прилагат. За повечето от тях преобладаващият отговор (в различен процент) е най-ниската степен — „не проявявам никакъв интерес“. Най-отблъскваната методика се оказва прилагането на метрики и различни техники на измерване. Обяснението е, че сред теоретиците не съществува единно становище относно това, кои метрики кога да се прилагат, както и че липсват за повечето теоретично разработени метрики съответни програмни средства, които ги реализират. Това, разбира се, не пречи големи фирми (Hewlett—Packard, IBM) да ги прилагат масово и задължително. Особено учудващо е, че практиците нямат голям интерес и към обектно ориентираните (OO) методи. Тук обяснението се търси в това, че от една страна, за проектантите и програмистите от по-възрастното поколение тази парадигма е необичайна и трудна за усвояване, че немалка част от тях се занимават със съпровождане на софтуер, създаван преди доста време с тогавашните средства и неподлежащ на преработване с OO-методи, че независимо от неоспоримите им предимства по отношение на валидирането на програмите OO-методите засега изостават от класическите структурни методи.

Свързан с този е и въпросът, докъде всъщност е науката в създаването на софтуерни продукти. Ясно е, че ако трябва например да се натоварят повечко пакети в багажника на кола, никой не решава въпроса математически (въпреки изключително елегантните методи, които математиката дава за целта) — всеки прави няколко проби и решава проблема. В много случаи софтуеристите не могат без формални методи или пък биха постигнали по-лоши резултати без тях. Във всеки случай с такива методи трябва да се процедира, когато задачата е добре формулирана и разбрана или пък е с рутинен

характер. Когато обаче е поставен сложен проблем или пък такъв, изискващ творческо решение, евристичният подход е по-добрата възможност.

Литература

1. Fox J.M., Software and its Development. Prentice-Hall, Inc., Englewood Cliffs, 1982.
2. Charette R.N., Software Engineering Environments, Concepts and technology. Intertext Publications, Inc., McGraw-Hill, Inc., New York, 1987.
3. Sommerville I., Software Engineering. Addison Wesley Publishing Company, 4. edition, 1992.
4. Glass R.L., Formal Methods vs. Heuristics: Clarifying a Controversy, The Journal of Systems and Software, 15(1991), No 2, p. 103—105.
5. Glass R.L., „Who Cares?” Technologies in Practice, The Journal of Systems and Software, 41(1998), No 1, p. 1—2.

2. ЖИЗНЕН ЦИКЪЛ НА ПРОГРАМНИЯ ПРОДУКТ

2.1. Моделиране на жизнения цикъл

2.1.1. Защо се нуждаем от модел на жизнения цикъл

Жизненият цикъл (ЖЦ) на програмния продукт (ПП) обхваща целия период на неговото създаване и използване. За начало на ЖЦ се смята моментът на възникване на идеята за създаването му. За край на ЖЦ се смята моментът, в който се преустановява използването на последното копие на ПП. В някои случаи се разграничава физическият от логическия край на ЖЦ. За логически край се говори тогава, когато използването на ПП е било толкова дълго, че от известно време нататък не са били необходими никакви усилия по съпровождането му, т. е. спряло е внасянето на промени в ПП за поправяне на грешки, за усъвършенстването му или за пренасянето му в нова операционна и хардуерна среда или ПП продължава да съществува пасивно по отношение на разработчиците си.

Модели са необходими не само в областта на софтуерните технологии. Целта на конструктивните дисциплини е да се отговори на три основни въпроса:

- какво правим — каква е същността на обектите, които изучаваме и които евентуално се опитваме да построим;
- как го правим — с какви операции си служим, какви са техните характеристики;
- в какъв ред го правим — каква е точната и по възможност еднозначна последователност от операции.

2.1.2. Какви свойства има моделът

За да се отговори на горните въпроси, се използват модели на разглежданите обекти. Всеки модел се характеризира с три особености:

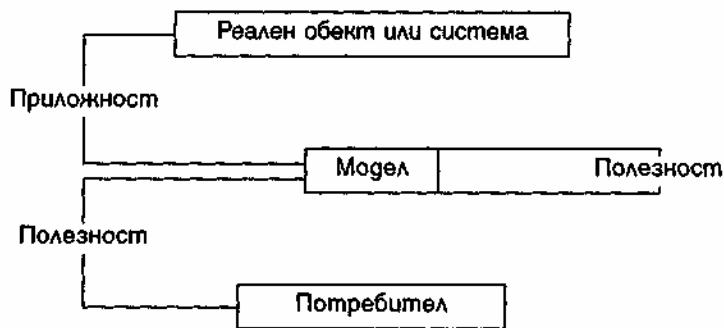
— **Изоморфизъм** — в математиката това понятие се дефинира съвсем точно. Нестрого казано, на всеки обект от реалността съответства обект от модела и на всяка операция върху един или няколко обекта от реалността съответства операция върху съответните обекти от модела.

— **Абстрактност** — на практика не е възможно да се постигне пълен изоморфизъм — причината е прекалено сложната реалност (поне за тези обек-

ти от реалността, които си заслужава да се изследват). Това впрочем е напълно в сила за програмния продукт и неговия жизнен цикъл — не е възможно да се построи модел, който да отрази всичките им особености.

— **Език за описание** — всеки модел се описва на някакъв език — чисто математическите модели ползват езика на някой дял от математиката, възможни са различни графични, диаграмни, блокови и други подобни езици. В много случаи обаче за описание на даден модел се ползва естествен език, а не са редки случаите, когато се ползва и повече от един език. Моделите на ЖЦ на ПП обикновено се отнасят към последния случай.

От казаното следва основният проблем при моделирането: представянето на реалността в модела не е нито точно, нито пълно, а това от своя страна може да се окаже решаващо и за неговата полезност. Затова при оценяването на моделите обикновено се разглежда следната схема:



Фиг. 2.1. Схема на моделиране

Изследването на адекватността между модела и реалната система определя *приложността* на модела. Изследването на вътрешните характеристики на модела и способността му да дава адекватна информация определят *валидността* на модела. Що се отнася до потребителя, той оценява модела от гледна точка на неговата *полезност*.

2.1.3. Какви могат да бъдат следствията от един успешен модел

Ако изграждането на модела е успешно, това има немалко положителни следствия:

— **Методологически** — моделът дава отговори на общи въпроси от рода на това, следва ли съпровождането да бъде разглеждано като изцяло отделна дейност или не, какъв тип специалисти следва да го извършват, възможно ли управлението на мерките по осигуряване на качеството да се разглеждат отделно от цялостния Производствен процес, следва ли за отделните извършвани функции да се правят отделни планове и кой да ги прави, кой да ги обсъжда и кой да ги одобрява и т. н.

— **Организационни** — как да се комплектова и ръководи екипът разработчик, какъв тип юерархия да се възприеме (ако въобще се приеме юерархична организационна структура), следва ли всеки проект в дадената организация да си има собствен ръководител и ако това е така — докъде да се простират управленските му пълномощия, допустимо ли е даден специалист или дадена група от специалисти да работи едновременно по повече от един проект и т. н.

— **Технологически** — какъв тип спецификации и техники на програмиране са препоръчителни евентуално с оглед типа на разработвания софтуер, какви метрики да се ползват за оценяване качеството на разработвания продукт докато той се прави и след като бъде завършен, възможно ли е например ползването единствено на обектно ориентиран подход от самото начало на ЖЦ до завършване на тестването на ПП.

2.2. Класификация на моделите на ЖЦ на ПП

Възможни са най-различни класификации. Тази, която следва, се опитва да обхване почти всички по-известни модели на ЖЦ на ПП, като отразява определящия ги признак. Към типовете ЖЦ в скоби са дадени по един или повече примера за съответния тип. Изрично трябва да се отбележи, че посочените като примери ЖЦ са на най-различно равнище на разработеност — някои от тях са изложени в единствена статия, понякога само като идея, други са предмет на цяла книга, в която твърде подробно и

конструктивно са развити необходимите елементи за прилагане на съответния модел, трети пък, по-малко на брой, имат и реализирани поддържащи ги програмни средства.

1. Пълни

1.1. Едномерни

1.1.1. Хронологични

1.1.1.1. Стандартни (Боем, Метцгер, Фрийман)

1.1.1.2. Модифицирани (каскаден, прототипен)

1.1.1.3. Разклонени (Фокс)

1.1.2. Функционален (Хамилтън—Зелдин)

1.2. Многомерни

1.2.1. Двумерни (Гънтър)

1.2.2. Тримерни (Питърс—Трип)

1.3. Еволюционни

1.4. Спирални

2. Частични — за повторно използване (Епълтън)

2.3. Типове модели на ЖЦ на ПП

Тук ще разгледаме най-общо типовете модели на ЖЦ, като следваме дадената по-горе класификация. Редът на изложение ще бъде различен от този на класификацията, за да обхванем по-напред тези модели, които представляват повече теоретичен интерес — било поради това, че са твърде абстрактни, било поради това, че вече са твърде остарели и са интересни най-вече от гледна точка на проследяване развитието на идеите за моделиране на ЖЦ на ПП. По-подробно ще се спрем на модела на Гънтър [1] (1.2.1.), който, от една страна, е твърде

подробно разработен, а от друга — твърде всестранно представя ЖЦ на ПП. Основният му недостатък е, че някои от конкретните числови стойности в него **вече** не обхващат увеличилото се разнообразие от типове софтуер. Ние обаче **няма** да навлизаме в такива подробности.

2.3.1. Стандартни хронологични модели

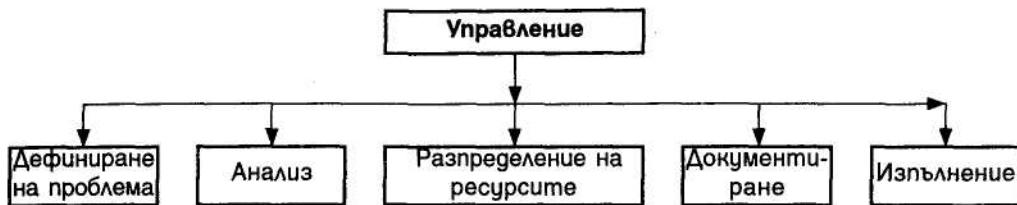
Съгласно класификацията (1.1.1.1.) тези модели са построени на хронологичен принцип. Това означава, че тяхна основа е развитието на ПП във времето. Основният обект в тези модели е **фазата**. Всяка фаза е отрязък от време, през което се извършват определени дейности върху разработвания ПП. В следващата таблица в успоредни колони са дадени трите най-известни стандартни модела.

Модел на фрийман	Модел на Метцгер	Модел на Боем
1. Анализ на необходимостта от създаване на ПП 2. Специфициране на ПП: създаване и описане условията за реализацията му. Оформяне на „Спецификация на ПП“	Определяне на софтуерната система	1. Определяне на изискванията към ПП 2. Определяне на изискванията към средата на използване и на разработване
1. Проектиране архитектурата на софтуерната система (външен проект) 2. Детайлно проектиране (вътрешен проект)	Проектиране	1. Предварително проектиране 2. Детайлно проектиране
Реализация на ПП 1. Програмиране 2. Тестване	1. Кодиране (програмиране) 2. Вътрешно тестване (по модулни и системно) 3. Приемни изпитания от независими експерти — дали софтуерната система може да се пусне на пазара	1. Програмиране 2. Тестване и експериментиране (експериментално внедряване, проверка на работоспособността на ПП)
Поддържане: осъществяване на връзки с потребителя за инсталациране на ПП, обучение, регистрация на грешки	1. Инсталациране 2. Експлоатация на ПП	Използване на ПП в реални условия

Табл. 2.1. Стандартни модели

2.3.2. функционален модел

Най-характерният пример за функционален модел е разработеният от Ха-милтън и Зелдин [2] — в нашата класификация той попада в категорията с код 1.1.2. Според тези автори най-същественият признак, по който следва да се декомпозира ЖЦ на ПП, е типът извършвани дейности. Такава група от приличащи си дейности по този модел се нарича функция. Следващата схема илюстрира разглеждания функционален модел.



Фиг. 2.2. Функционален модел

Дефинирането на проблема би следвало да дойде от взаимодействието на потребителя с разработчика.

Анализът представлява изясняване на проблема и избиране на решение в най-общия му вид.

Разпределението на ресурсите е особена функция, доколкото в другите модели на практика не се среща. Тук в нея се включва планирането и разпределението на всякакви ресурси, необходими за изпълнението на разработката — човешки, хардуерни, софтуерни, документационни и пр.

Документирането е функцията, резултатът от която са различните видове документация — вътрешна (за нуждите на разработката), потребителска (за нуждите на експлоатацията, която се извършва от потребителя) и съпровождаща (за нуждите на съпровождането, следователно отново става въпрос за вътрешна документация, ползвана от самия разработчик).

Изпълнението е реализацията на планираните и фиксираните във вече изброените функции дейности.

Всички функции са свързани с една **управленска**, която въпълнява идеята за непрекъснатата управляемост и контролируемост на процесите по разработването на програмния продукт.

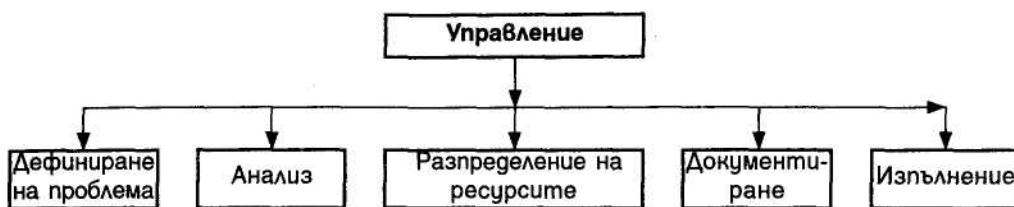
Логическата връзка между отделните функции се илюстрира от стрелките и тяхната посока.

Този модел, съгласно авторите му, се е използвал реално — на неговата основа е било разработено инструментално програмно средство (ПП) под името **VSEIT**. В литературата обаче не са известни други модели, изцяло и единствено основани на функционалността.

2.3.3. Разклонен модел на Фокс

Това е също особен модел (1.1.1.3. по класификацията). Той отразява гледната точка на един известен специалист с огромен практически опит от

разработването на много големи софтуерни системи. В този модел се отделя **особено** внимание на дейностите по съпровождането. В този смисъл **Фокс като че ли** изпреварва своето време. Доста по-късно от неговата книга [3] се **появяват**, от една страна, задълбочени изследвания, които обективно доказват **голямата** цена на съпровождането (като агрегация на усилия, квалификация, **време** и Други ресурси), а така също все повече се среща идеята за непрекъснатото усъвършенстване на веднъж разработения ПП. На фиг. 2.3. е илюстриран **моделът на Фокс**.



Фиг. 2.3. Модел на Фокс

Впрочем самият Фокс твърди, че терминът „съпровождане“ не е подходящ, и използва „поддържане“. Без да се впускаме в доводите на Фокс, ще игнорираме **този негов** възглед, за да постигнем единство в терминологията на всички **разглеждани** модели.

Като важен принос на Фокс следва да разглеждаме въведената от него **успоредност** на две от фазите. Тази идея ще видим видоизменена по-нататък в **каскадния** модел.

Разработването съгласно Фокс е най-тежката фаза от ЖЦ на ПП. **Безпроблемното използване** зависи силно от вложените усилия при разработването. **Съпровождането** според Фокс може да отнеме при големи ПП толкова **усилия**, колкото и разработването.

Известно внимание Фокс отделя на възможни модификации, които нарича **патологични** ЖЦ. При тях някоя от фазите не е застъпена или пък непрекъснатостта особено на връзката между разработването и съпровождането е разкъсана. Това се случва при някои разработки, когато тези две фази се възлагат на **различни** колективи.

Ясно е, че такова равнище на подробност не би довело до адекватен модел, **приложим** за практически нужди, поради което Фокс въвежда още едно ниво за **всяка** от фазите, което съдържа разбивка на харacterните за съответната фаза **дейности** („усилия“, както ги нарича той). За разработването това са:

- Дефиниране на изискванията
- Проектиране
- Написване на програмите
- Сглобяване на програмите
- Тестване
- Документиране

За съпровождането дейностите са следните: ■— Добавяне на нови функции

- Усъвършенстване на съществуващи функции
- Добавяне на функции, свързани с промени в хардуера
- Отстраняване на установени грешки

2.3.4. Тримерен модел на Питърс—Трип

Този модел (1.2.2. по класификацията) представлява само теоретически интерес. Той е опит да се обхванат заедно различни страни на процеса на разработване на софтуер, но като че ли третото въведено измерение, свързано с използваните формални апарати, няма практическо значение (поне равностойно на останалите две измерения).

Съгласно авторите, следва да се разглеждат три измерения (аспекта) на разработването на ПП — време, логика и формализъм. Фигуративно те могат да се представят като паралелепипед. Ние ще се задоволим с кратко описание на тези три измерения.

— *Време*. Това е вече среяната хронологическа компонента и тя отразява развитието във времето на ПП. Вероятно поради това, че чрез останалите две компоненти се допринася за по-пълното описание на ЖЦ на ПП, тук се отличават

само 4 фази — анализ на системата (ПП), проектиране, реализация и експлоатация.

— *Логика*. В това измерение се категоризират дейностите, които се извършват през отделните фази, дефинирани вече в първото измерение. Те са: определяне на проблема и оценъчно проектиране, синтез на системата (ПП), анализ на системата, оптимизация, вземане на решение, планиране на действията. В тази съвкупност от дейности, която не е подредена случайно по този начин, смисълът на синтеза е да се създадат няколко алтернативни идеини проекта, на анализа — да се идентифицират и систематизират изводите и оценките на всяка от алтернативите, а на оптимизацията — да се подредят алтернативите по един или повече критерии. След тази дейност остава само да се вземе решение за избор на алтернатива.

— *Формализъм*. В това измерение авторите включват необходимите според тях за разработването формални модели на ПП. Отличават се следните модели по реда на тяхното възникване и реализация. Започва се чисто мислен модел, очевидно отразяващ първоначалните идеи на разработчика за ПП. Този модел се преобразува в структурен. Препоръчително е той да се базира на някакъв математически формализъм от рода на графи (евентуално някакъв фиксиран тип графи — дървета, мрежи и пр.). Последният модел е лингвистичен и той придобива формата на последователност от текстови оператори с определен синтаксис или семантика. С други думи казано, това е вече било проект на ПП на определено ниво, описан формално, било ПП, реализиран във вид на определен програмен език.

2.3.5. Частични модели

Терминът „частични“ вероятно търпи известна критика. Той е използван, за да се покаже, че в този тип модели по правило липсват съществени части от естествения ЖЦ на ПП. Причината за това е, че тези модели се опитват да отразят т. нар. *reuse* техника, чийто смисъл е малко или повече използване на вече създадени компоненти от други ПП — програмни модули, проектантски решения, документация. Подходът на повторното използване се смята в момента за изключително перспективен. Полагат се усилия за оценяването на резултатите от прилагането му, както и за разработването на подходящи инструментални средства — например библиотеки от модули за повторно използване с подходящи механизми за търсене на нужните на конкретния нов ПП компоненти.

Като пример за такъв модел ще разгледаме предложения в [4]. Основното понятие в него е *asset*, чийто най-точен превод на български е като че ли не особено хубавата дума заготовка. Съгласно модела има няколко възможни равнища на ползване на заготовки. Съответната схема е дадена на фиг. 2.4.



Фиг. 2.4. Модел на заготовките

Във всички случаи прилагането на модела започва с оценка на изискванията. Оттам нататък в зависимост от резултата първоначално става насочване или към ниво 3, или към ниво 2.

Насочването към *ниво 3* означава, че разработването на новия ПП е възможно практически изцяло на основата на заготовки. След подбора им следва сглобяването им и ПП е готов за разпространение.

Следващ по сложност е вариантът с насочване към *ниво 2*. Прави се настройка на избраните заготовки. На практика оттук са възможни два изхода:

- В първия случай настроените модули са готови за използване и се повтаря процедурата от ниво 3 — пристъпва се към сглобяване.
- Във втория случай има настроени модули, които въпреки настройката не удовлетворяват изискванията. Тогава се отива на *ниво 0*.

Възможно е оценката на изискванията да покаже, че не е възможно насочване нито към ниво 3, нито към ниво 2. В този случай насочването е към *ниво 1*.

Там разработването на необходимите модули се прави на основата на идеи от достъпните заготовки. Оттук са възможни два изхода — точно както от настройката на ниво 2.

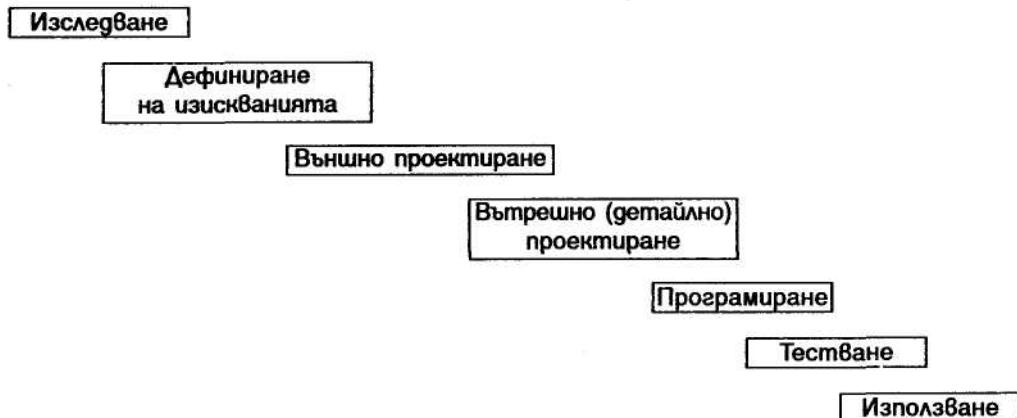
Последната възможност е *ниво 0*. Към процедурата на ниво 0 — проектиране и създаване на заготовки — не се отива непосредствено от оценката на изискванията. Насочването към нея е от ниво 1 или ниво 2, когато се е оказало, че получените програмни модули не отговарят на изискванията.

Ясно е, че в този модел не са отразени някои от дейностите, характерни за разработването на ПП. Идеята е да се подчертаят най-важните и специфични дейности по използването на готови елементи.

2.3.6. Каскаден модел

Това е модел от групата на хронологичните и модифицирани (1.1.1.2. по класификацията). Модификацията се състои в това, че отделните фази се припокриват

във времето. Това е една стъпка на подобреие на адекватността на модела на ЖЦ на ПП по отношение на стандартните модели. За илюстрация на фиг. 2.5. е един примерен вариант на каскадния модел. Трябва да отбележим, че в този пример степента на при покриване не съответства непременно на взаимното изображение на правоъгълниците.

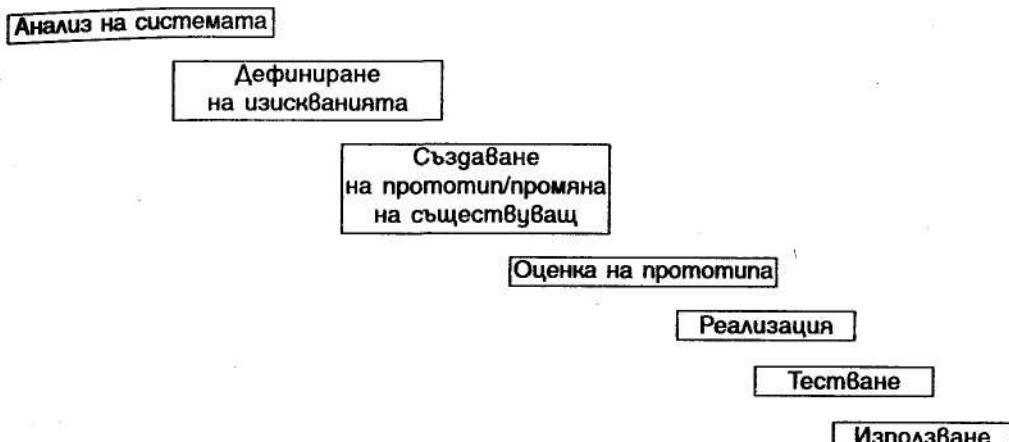


Фиг. 2.5. Каскаден модел

2.3.7. Прототипен модел

Този модифициран хронологичен модел (1.1.1.2. по класификацията) се използва твърде интензивно в практиката. Същността му се състои в това, че се реализира умалена версия (прототип) на крайния ПП, върху която се правят експерименти за установяване на съответствие с изискванията на потребителя. В зависимост от получените резултати се правят корекции с различна степен на връщане към предходните стъпки. Естествено, колкото по-близко е връщането, толкова по-бързо и евтино става окончателното завършване на прототипа. След

окончателното доказване на правилността на функциониране на прототипа и съответствието му с изискванията за разработката се пристъпва към реализацията на крайния 1111. Предимствата на такова прототипиране са очевидни — те спестяват различни ресурси на разработчика. Недостатъкът е липсата на гаранции, че прототипът ще съдържа всички съществени свойства на крайния ПП. В най-лошия случай може да се окаже, че поради неумело създаден прототип крайният ПП е разработен със значително повече ресурси, отколкото ако се е процедирало по стандартния начин, без прототип. На фиг. 2.6. е илюстриран про-тотипният модел.



Фиг. 2.6. Прототипен модел

Както се вижда, след преминаването за първи път през първите 3 фази се достига до оценка на прототипа. Обикновено се установяват несъответствия с изискванията. Тогава разработчика се връща на предходната фаза и променя прототипа (отначало чисто програмно, после на ниво проект, ако трябва), след което отново прави оценка. След известно повторение на този цикъл може да се окаже, че само промяна в прототипа не води до желаните резултати и тогава връщането трябва да стане една фаза по-назад. Следователно се налага промяна в определянето на някои изисквания. Това само по себе си е неприятно, защото всъщност изиска съгласието на потребителя (възложителя). Може обаче да се окаже, че дори и такива реални промени не водят до установяване на съответствие прототип-изисквания. Тогава следва връщане в начално положение, т. е. налага се да се направи наново изследване на проблема и възможностите за реализацията му, вероятно да се достигне до установяване на пропуснати особености и отново да се повторят всички следващи стъпки до оценката на новия прототип.

В даден момент прототипът отговаря на всички изисквания. Тогава се пристъпва към реализация на крайния ПП. Тестването, естествено, остава абсолютно необходима процедура и едва след неговото успешно приключване може да се пристъпи към използването на така създадения ПП.

2.3.8. Еволюционен модел

Във всички разглеждани дотук модели целият процес на разработка се основава на създаването и използването на малко или повече формални описания, наричани обикновено *спецификации*. Те създават яснота и определеност на разработването, но и създават трудности. Основната от тях е, че никой възложител не е в състояние от самото начало да фиксира своите изисквания така, че те повече да не се изменят в процеса на разработката. Прототипният модел прави известен опит за преодоляване на тази трудност, но това е само зачатък. *Еволюционният модел* (1.3. по класификацията) се опитва да интегрира специфициране, проектиране и реализация. Етапите в един еволюционен процес (забележете впрочем, че тук се говори за процес, а терминът „жизнен цикъл“ се изоставя) са:

— Формулиране първи вариант на изискванията към ПП. Тази „скица“ не е необходимо да бъде нито пълна, нито дори непротиворечива, но трябва да дава достатъчно указания на разработчиците, какво се очаква от ПП.

— Разработване на ПП възможно най-бързо на основата на така формулираните изисквания.

— Оценяване на ПП съвместно с потребителите или с възложителя и внасяне на изменения с оглед предявените нови изисквания от него. Това означава изменение на първоначалната функционалност на ПП и добавяне на нова, ако е необходимо.

Този подход обикновено се поддържа от т. нар. езици от четвърто поколение (4GL). Разработеният ПП може да се използва като база за строго специфициране на крайния ПП или да еволюира до ПП, който ще се достави на потребителя.

Ясно е, че този модел води до ПП, *съответстващ в най-висока степен на нуждите на потребителя*. Същевременно той поставя и *сериозни проблеми* [5]:

— Тъй като се фокусира върху крайния потребител, моделът не дава достатъчно приоритет на организационните аспекти на разработването.

— Постоянното изменяне по време на разработката влошава структурата на ПП до такава степен, че съпровождането му може да стане изключително трудно и скъпо. На практика това означава, че след относително кратък период целият ПП ще трябва да се напише отново.

— Цялостният процес няма желаната прозрачност и по този начин ръководството на проекта не е в състояние да оцени развитието му. Това е основната причина ръководителите да се въздържат от прилагането на еволюционния модел при големи проекти, в които най-тежкият проблем е управлението.

2.3.9. Спирален модел

През 1988 година Боем [6] прави опит да интегрира еволюционния модел с изискванията на управлението, предлагайки *спиралния модел* (1.4. по класификацията). В този модел разработването се движи спираловидно, основавайки се на спецификации. На всеки сегмент на спиралата се прави оценка на риска и се провежда редукция. При тази редукция се създават прототипи с оглед реду-

циране неопределенността на спецификациите, установяване на потребителски интерфейс и т. н. Това довежда или до обогатяване на спецификациите и до внасяне на по-голяма определеност в техния ПП, или пък до еволюция на прототипа до крайния ПП. При това се допуска различни компоненти на ПП да бъдат разработвани с прилагане на различни методи.

Смята се, че еволюционният и спиралният модел са най-подходящи за разработването на интерактивни системи със значителен потребителски интерфейс и за авангардни системи (например основани на изкуствен интелект), чиито изисквания много трудно могат да бъдат формулирани в началото.

2.4. Модел на Гънтър

2.4.1. Обща характеристика

По-горе бяха изтъкнати съображенията за по-подробното разглеждане на този модел. Както се вижда, по класификацията той е в групата на двумерните модели — 1.2.1. Едното измерение е хронологично, другото — функционално. Съгласно хронологичното

измерение, разработването на ПП преминава през 6 етапа, наречени фази. По време на тези фази с различна интензивност се осъществяват 7 функции.

2.4.2. фази

За всяка фаза ще определяме началния й момент, същността на фазата, какъв е резултатът в края на фазата и каква е средната й продължителност. Последната характеристика ще бъде давана така, както е определена от Гънтьр при създаването на модела с изричната забележка, че при сегашното разнообразие от ПП, нови инструментални средства и въобще значителен технологичен напредък тези времена могат само да дават известна представа за съотнасянето по продължителност на отделните фази.

Изследване

Начало. Това е началото на ЖЦ на ПП и е моментът на възникване на идеята за създаване на ПП.

Същност. През тази фаза се уточняват предназначението, основните функции и изисквания към разработвания ПП. Обикновено тази фаза включва маркетингово проучване на съществуващите в момента на софтуерния пазар аналогични ПП.

Резултат. След анализиране на съществуващи или описани в литературата аналогични ПП в края на фазата трябва да бъдат формулирани всички основни изисквания към ПП. Както и по-нататък, всеки подобен резултат се оформя в писмен документ. В случая той се нарича съглашение за изискванията (СИ).

Продължителност. 4—10 седмици.

Анализ на осъществимостта

Начало. Моментът на назначаване на ръководител на проекта.

Същност. През фазата *изследване* се установява какъв ПП трябва да бъде разработван. Въпросът е може ли това да се осъществи. Предназначението на тази фаза е да се установи може ли да се създаде ПП. Това става чрез анализ на:

а) Техническа осъществимост: анализ на достъпния хардуер както за осъществяване на разработката, така и за използване на готовия ПП. Освен вида на хардуера трябва да се анализират и необходимите за разработката други технически средства (инструментален софтуер, налични и изискуеми стандартизационни документи и пр.).

б) Икономическа осъществимост: анализират се и се оценяват в рамките на възможното цената на разработване, цена, на която би се продавал ПП, и цената за експлоатация на ПП. Анализът на цената на разработване трябва да завърши с пълна яснота, как ще бъдат осигурявани необходимите средства, т. е. икономическата осъществимост показва и откъде ще се осигурят необходимите средства.

в) Експлоатационна осъществимост: анализират се преимуществата и недостатъците на замисления ПП от гледна точка на потребителите му, например: изиска ли се специална квалификация, какви технологични операции трябва да се извършват и дали са по силите на средния потребител, ще се реализират ли разумни стойност по отношение на бързината на обработките и т. н.

г) Пазарна осъществимост: на основата на маркетингово проучване се прави опит да се прогнозира дали новият ПП ще се търси, може ли да бъде конкурентоспособен и

какви ще са силните му страни в сравнение с аналогичните ПП. Този анализ е важен за изграждането на маркетинговата стратегия на фирмата по отношение на ПП.

Анализът на всички видове осъществимост се извършва от група специалисти със съответната квалификация.

Резултат. След евентуални промени се утвърждават формулираните в СИ изисквания.

Продължителност. 1—10 седмици след края на предишната фаза.

Проектиране

Начало. Съвпада с края на фазата *изследване*.

Същност. Целта е обхващането и отразяването на потребителския възглед за ПП. Това се нарича външен проект на предвидения ПП. Той представя ПП като черна кутия, към която се подават определени данни и се получават определени резултати. В детайли се обсъжда с потребителя и се проектира потребителският интерфейс. Разглеждат се и се уточняват връзките на ПП с операционната среда и други софтуерни средства, с които е свързано използването на ПП.

Резултат. Създаденият външен проект се оформя като документа „Външна спецификация“.

Продължителност. 10 седмици.

Програмиране

Начало. Програмирането може да започне, когато външната спецификация е представена в някакъв вид, макар да не е преминала окончателно формално утвърждане. Във всички случаи обаче СИ трябва да е било утвърдено.

Същност. През тази фаза се създава детайлен (вътрешен) проект. В него фактически е описано как ще се реализира ПП. Създава се с цялостната структурна схема на ПП. Вътрешният проект представя детайлно архитектурата на ПП с пълно описание на всяка програмна част, описание на потока на данните и потока на управлението. Детайлният проект трябва да се опише на такова ниво на подробност, че по това описание да могат да се напишат програмите на ПП.

През тази фаза се извършва и така нареченото модулно тестване, т. е. всяка програмирана единица се тества самостоятелно. След завършване на модулното тестване всички създадени програмни части се интегрират в единна система, която по-нататък (но не в тази фаза) се тества като едно цяло (системно тестване)

Резултат. Получава се работоспособен ПП, който може да бъде представен за независимо тестване и последващи приемни изпитания.

Продължителност. Особено силно зависи от сложността и обема на ПП. По принцип обаче Гънтьр смята, че не трябва да надвишава 10 месеца. Ако предварителните оценки показват по-голяма продължителност, то проектът трябва още в началото да се декомпозира на няколко отделни части с продължителност на фазата програмиране под 10 месеца.

Оценка

Начало. Това е моментът, в който ПП е сглобен от готовите програмни модули.

Същност. Целта е да се извърши така нареченото независимо тестване, т. е. създаденият ПП да се тества от специалисти, които не са участвали в разработването му. След това се провеждат приемни изпитания. Тяхната основна цел е да се провери

съответствието между разработения ПП и СИ, съответствията между ПП и съставените за него спецификации; да се документира експлоатационната годност на ПП в реални потребителски условия; да се проверят качествата на документацията, включително нейната пълнота.

Резултат. Документ (протокол, сертификат), удостоверяващ експлоатационната годност на ПП, който позволява последващо предаване за разпространение и експлоатация.

Продължителност. Силно зависи от качеството на работа през предишните фази. Най-оптимистичната оценка е 1/3 от продължителността на фазата *програмиране*. Има случаи обаче, когато се достига продължителността на програмирането.

Използване

Начало. Това е моментът на издаване на документа за годност.

Същност. Тази фаза включва инсталацирането и последващата експлоатация на ПП. Извършва се обучение на потребителите с различна продължителност в зависимост от квалификацията на потребителите и сложността на ПП. Всички видове съпровождане (усъвършенстване, отстраняване на грешки, добавяне на нови функции) също се извършват по време на тази фаза.

Резултат. Експлоатацията на ПП.

Продължителност. Определя се от конкретния ПП. Физическият край на ПП настъпва, когато и последното копие на ПП се свали от употреба. Логически край на ПП — когато след достатъчно дълъг период на използване ПП е достигнал до такова състояние, че не са необходими никакви съпровождащи дейности от страна на разработчика за този ПП.

2.4.3. Функции

Под функция се разбира съвкупност от сходни дейности, извършвани от група хора със съответната квалификация. Съгласно модела на Гънтьр за всяка от седемте функции има отделна *група* от хора. Реално обаче това може да се осъществи само в големите софтуерни фирми.

Планиране

Тази функция обхваща дейностите по съставяне и проследяване на изпълнението на всички видове планове. В зависимост от обхвата си плановете се делят на:

- а) такива, които се отнасят до организацията производител (фирмата) като цяло:
 - целева програма,
 - стратегически план,
 - тактически план;
- б) свързани с всеки конкретен ПП:
 - бюджет,
 - план за работата на отделна група,
 - индивидуален план за всеки участник,
 - мрежов график,
 - други.

Групата по планиране следва да се състои от хора с икономическо образование и с някаква специализация в областта на информатиката и на софтуерните технологии. Най-интензивно групата по планиране работи по време на фазите изследване и проектиране.

Разработване

Тази функция включва всички дейности, свързани с традиционните представи за програмирането. Групата по разработване проектира, следователно съставя външната и вътрешната спецификация, извършва програмирането и тестването (разбира се, без независимото), отстранява грешки, сглобява готовите модули, консултира съставящите съпровождащата документация. Тази група включва специалисти информатики и е основната.

Обслужване

Основната идея на Гънтьр, въвеждайки тази фаза, е, че не е възможно и разумно на висококвалифицирани специалисти да се възлагат дейности от чисто технически характер.

Дейностите по обслужването се разделят на следните групи:

а) административно-правни — свързани със съставянето на някои от документите:

- договори,
- мрежови графици,
- финансови отчети,
- данъчна документация.

Частта от групата, отговаряща за този тип дейности, включва и специалист с правообразование.

б) технически — състоят се в осигуряването на:

- изчислителната техника и нейното поддържане,
- заявените инструментални софтуерни средства,
- стандарти и други нормативни документи.

в) изпитания от клас С — това са крайни изпитания, състоящи се в случаен избор сред готовите за разпращане комплекти от ПП и проверката им за качество и комплектност.

Документиране

Съответната на тази функция група отговаря за създаването и поддържането на *потребителската документация*. В нея се включват различните ръководства, инструкции, справочници, необходими на потребителя за правилното Я ефективно експлоатиране на ПП. (Трябва да се прави разлика с вътрешната документация на разработчика, която се създава от други групи.)

Групата включва специалисти с филологическо образование, евентуално специалисти, които изготвят потребителската документация на чужд език (ако ще се разпространява в чужбина). В по-големите фирми тук се включва И специалист по софтуерни технологии. Задачата му е да проектира структурата на потребителската документация, да определи подходящи примери За илюстрирането й, след завършването й да провери съответствието с действието на ПП.

Изпитания

Тази функция обхваща дейностите по установяване на наличието на дефекти и на разлики между действителните свойства на ПП и спецификациите му. Разграничават се 3 вида изпитания:

- а) Клас А — вътрешни, извършвани от самите разработчици при програмирането и сглобяването на ПП.
- б) Клас В — независими, които се изпълняват от групата по изпитанията.
- в) Клас С — случаини, описани по-горе и извършвани от групата по обслужването.

Тъй като групата по изпитания се занимава с откриване на дефекти, тя се състои от висококвалифицирани специалисти информатики.

Поддържане

Тази функция обхваща всички дейности, свързани с преките контакти на фирмата производител с потребителите:

- проучване на потребителското мнение във фазата анализ на осъществи-мостта;
- защитаване интересите на потребителя още при избора на проектантски или програмистки решения;
- обучение на потребителя;
- регламентиране и осъществяване на обратна връзка с потребителя на даден ПП: получаване и систематизиране на съобщенията за грешки и на заявките за изменения по искане от страна на потребителите;
- връзки с обществеността.

Съпровождане

Обхваща всички дейности по внасяне на промени в готовия ПП:

- поправяне на грешки;
- добавяне на нови възможности;
- адаптиране на ПП към нова операционна или хардуерна среда.

Извършва се от разработчиците. Изиска най-високата възможна квалификация, защото, както е известно, в повечето случаи внасянето на коректни изменения в съществуваща програма е по-трудно, отколкото създаването на нова програма.

Съпровождането може да се извърши на различни равнища:

- а) гаранционно съпровождане — обикновено то включва задължения за безусловно и безвъзмездно отстраняване на грешки по искане на потребителя, като по българските стандарти има срок 12 месеца;
- б) развитие на ПП и отстраняване на грешки;
- в) отстраняване на грешки;
- г) регистриране на съобщенията за грешки и на заявките за подобряване с оглед вземането им под внимание при евентуално разработване на нов подобен ПП.

Режимът на съпровождане може да зависи от конкретно подписания договор, ако има пряка връзка между разработчика и потребителя.

На фиг. 2.7. са представени по хоризонталната ос фазите на ЖЦ на ПП, по вертикалната — функциите, а така също и интензивността на отделните функции по

време на отделните фази. Както обикновено се приема, колкото една функция се изпълнява по-интензивно, толкова по-наситен е цветът на представянето.

	Изследване	Осъществимост	Проектиране	Програмиране	Изпитания	Използване
Планиране						
Разработване						
Обслужване						
Документиране						
Изпитания						
Поддържане						
Съпровождане						

Фиг. 2.7. Модел на Гънтьр

Литература

- 1 **Gunther R.C.**, Management methodology for software product engineering. New York, 1978.
- 2 **Hamilton M., S. Zeldin.** The Functional Life Cycle Model and Its Automation: USE.IT. J. of System and Software, 3 (1983), p. 25—62.
- 3 **Fox J.M.**, Software and its Development. Prentice-Hall, Inc., Englewood Cliffs, 1982.
- 4 **Appleton D.**, The Asset-based Life Cycle Model, Very Large Projects, Datamation, January **15 1986**, p. 63—70.
- 5 **Somrnerville I.**, Software Process Models, ACM Computing Surveys, Vol.28, No.1, March **1996**, p. 269—271.
- 6 **Boehm B.W.**, A spiral model of software development and enhancement. IEEE Computer, 21, Sip. 61-72.

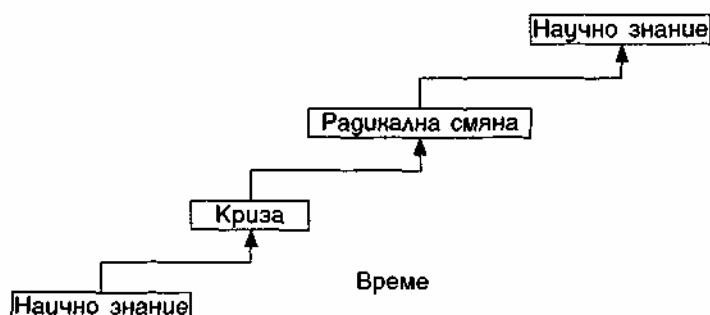
3. КОНВЕНЦИОНАЛЕН ПОДХОД ЗА РАЗРАБОТВАНЕ НА СОФТУЕР

3.1. Модели на жизнения цикъл и подходи за разработване

Проучването на специализираната литература показва, че продължава създаването на нови модели на ЖЦ. Поради стремежа да обхванат най-различни аспекти (технологични, организационни и икономически) новосъздаваните модели са все по-сложни и оттам — все по-малко приложими в практиката. Затова в последно време усилията са насочени към разглеждане на *подходи* за разработване. Всеки подход за разработване определя основните правила, практики и процедури, които трябва да се следват в процеса на създаване на ПП.

Подходите могат да бъдат с различна степен на общност. Някои от тях обединяват сродни методи. Например известни от литературата са подходът за разработване съгласно целите (чрез проследяване на изискванията или чрез про-тотипиране) или подходът с управление на риска.

Най-общите фундаментални подходи се наричат *парадигми*. Ще разграничаваме два такива подхода: *традиционнен* (конвенционален) и *обектно ориентиран*. Традиционният подход се е формирал с възникването на научното направление софтуерни технологии като възможно решение на т. нар. софтуерна криза. За съжаление десетилетия след прилагането му някои тревожни симптоми на софтуерната криза — висока цена на разработване и незадоволително качество — продължават да се наблюдават. Затова бе създаден нов поход — обектно ориентирианият. Такова развитие е в пълно съответствие с диаграмата на Kuhn [1], показана на фиг. 3.1



Фиг. 3.1.

Когато текущото състояние на научното знание не е в състояние да удовлетвори очакванията ни, се оказваме в състояние на *криза*. Тази криза се следва радикална смяна на научното знание, т. е. на начина, по който моделираме или правим нещо. Следователно сегашната ситуацията в софтуерното производство не е нова — просто се преминава от един подход към друг с неизбежните за такъв преход трудности, породени от необходимостта от усвояване и прилагане на нови идеи.

В тази глава ще представим основните принципи на традиционния подход на разработване на софтуер, следвайки хронологичен модел на ЖЦ, последователните фази на който са определяне на софтуерната система, проектиране, програмиране, интегриране и тестване, използване.

3.2. Определяне на софтуерната система

Целта на тази фаза е да се определят предназначението и обхватът на софтуерната система; да се разберат потребностите и желанията на потребителя; да се оцени осъществимостта на избираните решения; да се обсъди и избере приемлива съвкупност от изисквания, които да се специфицират, валидират и утвърдят, като се регламентира и проследяването им така, че да се осигури вграждането им в целевата система.

3.2.1. Определяне на изискванията

Основен проблем при определяне на изискванията е разминаването на специалистите в съответната приложна област (наричани най-общо потребители) И софтуерните специалисти, които отговарят за първоначалното определяне на системата (т. нар. системни аналитици или проектанти). Обикновено потребителите не могат ясно и точно да формулират потребностите и очакванията си; пропускат очевидните за тях подробности и нямат реална представа за възможностите на компютърните системи. Аналитиците пък не познават особеностите на дейностите, които ще автоматизират, и не могат да преценяват доколко са съществени обсъжданите проблеми. Затова определянето на изискванията трябва да се разглежда като итеративен процес с активното участие на подходящо избрани представители и от двете страни.

Ще опишем целите, съдържанието и прилаганите техники за всяка от последователните стъпки.

а) Формулиране на изискванията

Основна цел на този етап е съставяне на съвкупност от всички възможни Изисквания въз основа на определените вече предназначение и обхват на предлаганата софтуерна система. В [2] е предложен подходът FAST (Facilitated Application Specification Techniques). Съгласно този подход се съставят работни групи с представители на потребителите и разработчиците, със съвместните Усилия на които се идентифицира проблемът, обсъждат се алтернативни решения и се подготвя предварително множество от изисквания. Регламентирани са Формите на комуникации, подготовката и организирането на ефективни обсъждания и др. Друга подходяща техника е *анализът на различните гледни точки*, описан в [3]. Първоначално се идентифицират, например чрез „мозъчна атака“ (brainstorming), всички възможни потребители на системата, които след това се групират по сходство на гледните точки, които те представят. В резултат на интервюта, провеждани от системните аналитици с членове на групата, разговори и целенасочени обсъждания се определят изискванията на групата. Ако е необходимо, се разработват сценарии (use-cases) за реализацията на някаква основна функция и дори прототипи за нея, за да могат да се дефинират поточно съответните изисквания. Всяко изискване се описва (същност на изискването, кой и защо го предлага). Тази документация е необходима за по-нататъшното разглеждане и оценяване.

б) Анализ на изискванията

Съставената съвкупност от изисквания се изследва за пълнота и непротиворечивост. Всички изисквания се класифицират по няколко критерия — например на функционални и нефункционални, на задължителни и препоръчителни, като може да им се присъюва и коефициент на значимост. Изследват се връзките помежду им и противоречията се разрешават на основа на присвоения им приоритет. След формиране на непротиворечива система за всяко изискване се определя доколко ясно и точно е формулирано, осъществимо ли е в определената среда за разработване, с какви рискови фактори е свързано, с какви ресурси би се реализирало, проследимо ли е в процеса на реализация на ПП и как може да се провери удовлетворяването му в целевата система.

Полученото описание на изискванията е документ, който се нарича „Спецификация на изискванията“ (според други автори — „Съглашение за изискванията“).

в) Утвърждаване на изискванията

Създадената спецификация на изискванията се проверява от представители на заинтересованите страни — потребители и разработчици, като се изследват и изискванията като цяло, и всяко изискване поотделно. Обичайната техника е използване на стандартни въпросници, чрез които се постига система-тичност и изчерпателност на проверката. След приключването ѝ спецификациите на изискванията се утвърждават и стават основен документ за цялата софтуерна разработка.

г) Проследяване на изискванията

Препоръчва се планиране на дейностите по проследяване на изискванията, като в определените за проекта контролни точки се включат и проверки за постигнатата степен на удовлетворяване на избрана подсъкупност от изисквания.

3.2.2. Аналитичен модел

Създадените спецификации на изискванията към софтуера са основа за изграждане на първия формален модел на целевата система — т. нар. *аналитичен модел*. Той е резултат от проведенния структурен анализ и предназначението му е да улесни следващите дейности по проектиране. Основните съставящи на аналитичния модел са:

а) Модел на данните

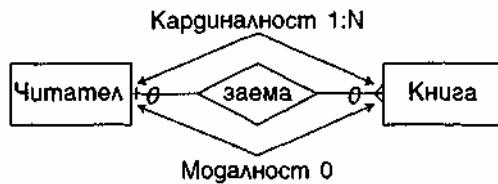
Моделът на данните представя основните *обекти*, *атрибути*, които ги описват, и *връзките* на обектите помежду им. *Обект* може да бъде всяко нещо, което създава или използва информация в софтуерната система. Описанието на обекта включва описание на същността и на атрибутите му, които представлят основните характеристики и свойства на този обект. Например в една библиотечна система обектът книга има атрибути регистрационен номер, индекс (за принадлежност към област в съответствие с общоприета класификация), заглавие, автор(и), издателство, година на издаване, цена, език, на който е написана, и др. Друг обект е **читател**, който може да се опише с атрибути име, ЕГН, адрес, образование, месторабота.

Обектите са свързани помежду си по различен начин. Например обектите читател и книга са свързани с отношения заемам, връщам, загубвам, поръчвам. Разглежданите отношения са ориентирани, т. е. имат посока, която трябва да се отчита при анализирането и изобразяването им.

За всяко отношение могат да се определят стойностите на две основни характеристики: *кардиналност* и *модалност*. *Кардиналността* е характеристика, която определя колко екземпляра от единия обект могат да са в отношение с екземпляри от другия обект. Кардиналността се описва с „един“ и „много“ и може да бъде един-към-един (1:1 читател — ЕГН), един-към-много (1:N читател — книги) и много-към-много (M:N читатели — групи по интереси).

Модалността определя дали отношението между два обекта е задължително (модалност 1) или незадължително (модалност 0). Например отношението заема между читател — книга в двете посоки има модалност 0, защото във всеки момент има читатели, които не са заети книги, и книги, които не са взети от читатели.

Използвайки приетите означения, отношенията между разглежданите обекти могат да се представят графично по следния начин:



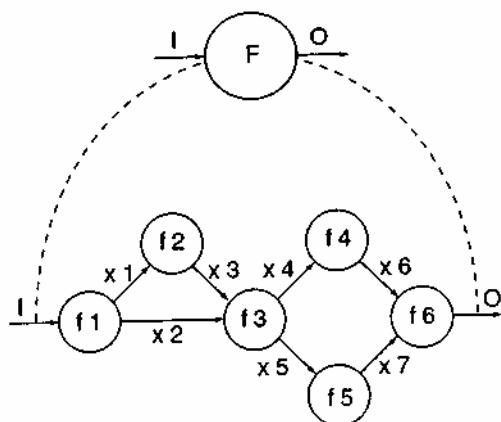
Фиг.3.2.

Диаграма, представяща обектите и отношенията между тях, се нарича *диаграма елемент-връзка* (*entity-relationship diagram ERD*). Тя е предложена за пръв път от Чен [4] за проектиране на релационни СУБД и после многократно е разширявана и по съдържание, и по форма. ERD е графично представяне, в което обектите се представят с надписани правоъгълници, а отношенията — чрез свързващите ги надписани линии. Въведени са специални означения за кардиналността и модалността на всяко отношение.

Описанието на обектите и атрибутите им заедно с диаграмата, представляща отношенията между тях, определят *модела на данните*.

б) Функционален модел

Предназначението на този модел е да представи основните функции на софтуерната система чрез проследяване на преобразуването на информацията в нея. Диаграмата на потока на данните (Data Flow Diagram) е графично представяне, описващо трансформациите на данните така, че от входните данни в системата да се получат исканите изходи. Използваните графични елементи са: правоъгълник за означаване на външни обекти, предаващи или приемащи информация от системата; кръг — за означаване на функция, променяща данните по някакъв начин и надписани стрелки, представящи съответните входни и изходни данни. Пример за диаграма на потока на данните (ДПД) е показан на фиг. 3.3



Фиг. 3.3

Широкото практическо използване на ДПД се обуславя от тяхната изключителна простота и нагледност. Те могат да се съставят с различна степен на детализираност. Например така наречената фундаментална ДПД (диаграма на ниво 0) представя софтуерната система като една обобщена функция, преобразуваща входа на системата в изхода. По-нататъшното анализиране изяснява основните функции, докато се достигне до описание, което може да бъде основа за проектирането. Графичното представяне на функциите може да се съпътства с допълнително описание на всеки елемент в ДПД. Това описание се нарича *спецификация на процеса*. Освен входа, изхода и същността на

извършваната трансформация могат да се задават и допълнителни изисквания към всяка от описаните функции.

Класическият структурен анализ е разширен с допълнителни техники за отразяване на особеностите на определени класове софтуерни системи. Например за моделиране на системи в реално време е предложено създаването на *диаграми на потока на управление* [5]. Диаграмите на потока на управление (ДПУ) и съответните им подробни описания, наречени *спецификация на управлението* (*control specification*), могат да бъдат създавани с помощта на съответни инструментални средства.

в) Поведенчески модел

За някои видове системи се предлага създаването и на модел на поведението на софтуерната система. При този подход системата се описва чрез различимите си състояния и начина на преминаване от едно състояние в друго. *Диаграмата на преобразуване на състоянията* (*state transition diagram*) представя графично наблюдаваните състояния на системата (чрез правоъгълници) и събитията, предизвикващи преминаването от едно състояние в друго (чрез стрелки). Събитията се описват чрез наредени двойки (пораждащо събитие, ответно действие).

Като задължителна част на аналитичния модел се разглежда и *речникът на данните*, който представлява систематично и точно описание на всеки елемент на данните, споменат в някой от моделите на софтуерната система. Описанието на елемент от речника обикновено съдържа:

- име на обекта;
- синоними — други имена, използвани за същия обект;
- списък на срещанията на този обект — къде и как се използва;
- описание на същността на обекта;
- допълнителна информация.

Основно преимущество на структурния анализ е простотата и нагледността. Освен с това широкото му практическо използване се обяснява и с наличието на множество инструментални средства, подпомагащи систематичното му прилагане за конструиране, описание и автоматично проверяване на пълнотата и непротиворечивостта на съответните модели.

3.3. Проектиране

3.3.1. Основни понятия

Проектирането на софтуерни системи обхваща всички дейности за превръщане на утвърдените изисквания в описание, определящо точно съдържанието и функциите на програмите, които трябва да бъдат създадени.

Основен принцип на проектирането е изследване и разбиране на проблема и последователното му декомпозиране на подпроблеми. На всяка стъпка се повтаря една и съща последователност от действия:

- идентифициране на проблема;
- съставяне на множество от възможни решения и сравняването им;

— избор на оптимално (в някакъв смисъл) решение. Обикновено се избира познато вече решение, за да може да се осъществи повторно използване на софтуерни компоненти.

Декомпозирането е съществено, защото е ефективен начин за справяне със сложността на създавания софтуер. Смята се за теоретично доказано и експериментално проверено, че ако сложен проблем се декомпозира (по подходящ начин) на два подпроблема, то усилията за решаване на цялостния проблем са по-големи от сумарните усилия за решаване на двета подпроблема.

Разграничават се два основни подхода към проектирането: функционален и обектноориентиран.

При функционалното проектиране софтуерната система се разглежда като съвкупност от компоненти, всяка от които реализира определена функция. Състоянието на системата е централизирано и се разделя между функции, опериращи върху това състояние.

При обектноориентираното проектиране системата се разглежда като съвкупност от *обекти*. Състоянието на системата е децентрализирано и всеки обект Управлява собственото си състояние. Обектите имат *множество от атрибути*, определящи състоянието им, и *операции* върху тези атрибути. Обектите Комуникират помежду си чрез *съобщения*.

По-нататъшното ни изложение е свързано с функционалното проектиране.

3.3.2. Методи и средства за проектиране

Както всяка основна дейност в разработването на софтуер и проектирането може да се реализира по два начина: като ad hoc процес или като стандартизиран процес. В първия случай въз основа на утвърдените изисквания се изготвя неформализирано описание на естествен език, което служи за ръководство при съставяне на програмите. Постистематичен е подходът с прилагане на т. нар. структурни методи, които предлагат начини за описание и процедури за създаване на софтуерни проекти.

3.3.3. Етапи на проектиране

Разграничават се два основни етапа на функционалното проектиране: *предварително* (понякога наричано *външно*) проектиране и *детайлно* (понякога наричано *вътрешно*) проектиране. Като резултат от предварителното проектиране се създава външният (архитектурен) проект. Той представя цялостната структура на софтуерната система и начините, чрез които тази структура осигурява концептуалната цялост на системата. Създаденият външен проект обхваща:

- логическата организация на данните — трансформация на създадения при анализа модел на данните в структури от данни, необходими за реализация на системата;
- структурата на системата — основните компоненти, външнопроявими-те им свойства и отношенията между тях;
- интерфейс на системата — между отделните й части, между системата и други софтуерни системи и между системата и потребителя.

При детайлното проектиране (*component-level*) софтуерната система се представя като йерархия от „черни“ кутии, които трябва да се реализират като обособени програмни части, наречени модули. Модулът е функционално обособен и стандартно оформлен елемент, който е основна, самостоятелно разработвана, тествана и документирана единица. При първата стъпка се създава схема на структурата, описваща основните модули, потока на данните и потока на управление между тях. При втората стъпка се специфицират модулите. Всеки модул има име и четири основни атрибута:

- вход и изход;
- основна функция;
- механизъм за реализация на функцията;
- вътрешни данни.

3.3.4. Методи за описание на проекти

Създадени са различни методи за описание на създаваните проекти, които се различават по степента си на формализираност и по нагледността и разбираемостта на получените описания. Практиката показва, че все още се предпочитат неформалните методи поради съпротивата и на потребителите, и на проектантите. Възраженията на потребителите са, че формалните описания са твърде сложни и неразбираеми за тях и те не могат да участват ефективно в обсъжда-

нето и проследяването на създаваните проекти. Възраженията на проектантите срещу формализмите са следните:

- не всички софтуерни специалисти имат необходимата теоретична подготовка (например в областта на дискретната математика и математическата логика), за да овладеят съответния апарат и да го прилагат за сложните реални системи;
- някои класове софтуерни системи са сложни за формално специфициране поради необходимостта от паралелни обработки, работа в реално време и др.;
- в софтуерните организации няма достатъчна информация за съществуващите техники за специфициране на проекти и за съответните инструментални средства, които ги подпомагат.

Използваният техники за описания са:

a) Графично описание

Този вид описание е най-старият, доколкото блок-схемите са един от най-предпочитаните начини за описание на алгоритми. В последните години се повишава приложимостта му поради появата на графични редактори и на специализирани софтуерни средства за графично проектиране.

b) Таблично представяне

Заемствано е от теорията на вземане на решения и се състои в съставяне на таблици, описващи анализираните условия и съответните реакции на системата за всяка комбинация от осъществени или неосъществени събития.

c) Текстови описания

Този вид описания са с различна степен на формализираност и най-общо се делят на две: *псевдокодове и езици за проектиране на програми* (PDL — Program Design Language).

Псевдокодът е ограничен и структуриран естествен език, обикновено английски. Речникът на езика включва глаголи, имена от речника на данните и запазени (ключови)

думи за описване на логиката на извършваните действия. Синтаксисът на езика допуска комбинации от линейна конструкция, конструкция за описание на условия и конструкция за описание на цикли. По-долу следва схематичен псевдокод за специфициране на модули. Той е създаден на основата на описани в литературата псевдокодове и е успешно приложен в реални софтуерни проекти:

СПЕЦИФИКАЦИЯ НА МОДУЛ

Module:<име>

Purpose:<предназначение>

Uses: <вход>

Returns: <изход>

Defines: <собствени данни>

Functional details:<описани чрез псевдокод>

Псевдокодът е неформален език — ограничен английски.

Речник:

= глаголи (прилагателните и обстоятелствените пояснения се пропускат поради нееднозначното им интерпретиране); = термини от речника на данните; = ключови думи за описание на логиката.

Синтаксис:

I. Последователни конструкции

=. разказвателно описание на английски език; =. оператор за присвояване.

II. Конструкции за условен преход:

IF <Boolean_expression_B> THEN <sequential_construct_S1> ELSE <
sequential_construct_S2>

III. Конструкции за цикъл

WHILE <Boolean_expression_B> DO BEGIN

END

FOR.....

NEXT

DO

UNTIL <Boolean_expression_B>

Различни начини за описание на проекти са подходящи за различните проекти или за части на един и същи проект. В [7] са описани следните критерии за сравняване на методите за описание на проекти:

- модулност;
- простота;

- леснота на редактиране;
- възможност за автоматично създаване и обработка (чрез съответни редактори);
- съпровождаемост;
- удобство на представяне на данните;
- възможност за верификация;
- сложност на прехода „проект—програма“

3.3.5. Организационни аспекти на проектирането

Успехът на проектирането зависи както от качеството на междинните продукти, създадени през предишните фази („Технико-икономическо задание“ и „Спецификация на изискванията“), така и от организацията на основните дейности. Като колективна дейност, извършвана от екип, проектирането трябва да се осъществява целенасочено и систематично, с регламентиране на основните процедури (за комуникация във и извън екипа, за управляемо променяне на изискванията, за документиране на процеса и на резултатите). Най-сложният проблем е определянето на подходящо ниво на декомпозиране и оттам — на подробност на проекта. Неясните, общи описания могат да породят проблеми при съставянето на програмите. Обратно, стремежът да се създават изключително подробни предписания още на този етап може да затрудни програмистите. Препоръката е: „Не казвай на програмис-

та, как да прави нещата. Казвай му само, какво трябва да се направи, и остави въображението му да те изненада (приятно?!).“

3.3.6. Оценяване качеството на проекти

За да се определи кои проекти са добри, трябва да се определят съответни характеристики на качеството, които могат да бъдат:

Пълнота — утвърдените изисквания да са отразени в проекта на софтуерната система.

Проектът да бъде разбираем, ясен, точно и недвусмислено описан и да не зависи от платформата, на която ще се реализира.

Проектът да бъде проверяем, т. е. да може да се проследява вграждането на изискванията.

Процесът на проектиране и създаванието на проекти да бъдат документирани.

Препоръчва се проектирането да завърши със създаването на документ, наречен „Спецификация на проекта“, който може да има следната стандартна структура:

Спецификация на проекта

I. Цел и обхват на дейностите по проектиране

II. Описание на проектите:

a) външен проект;

b) детайлрен проект.

III. Съответствие на проектите с утвърдените изисквания

IV. План за модулно и системно тестване

- V. Допълнителни условия и ограничения за проектирането
- VI. Приложения (описание на алгоритми, алтернативни процедури, таблични данни, извлечения от други документи и т. н.)

3.4. Програмиране

Структурното програмиране [8] е резултат от усилията да се създават програми, които не само могат да се изпълняват, но са и разбираеми, лесни за разуяване или променяне. Основната идея на структурното програмиране (понякога наричано програмиране без GOTO) е използването само на три класически конструкции: линейна последователност, конструкция за условен преход IF THEN ELSE и конструкция за цикъл DO WHILE. Доказано е, че всеки алгоритъм може да бъде описан с използване само на тези конструкции.

Техниката за написване на структурни програми съчетава *низходящото проектиране и постъпковото уточняване* (*stepwise refinement*). Започва се с най-обща схема на програмата, като всеки цикъл или проверка на условие се представят чрез съответните оператори, а всяка вложена част постепенно се разширява. Докато се получи окончателната програма. Милс, Ашкрофт и Мана посочват, че Всяка „правилна“ (определенi са изискванията към такава програма) неструктурна програма може да бъде трансформирана в еквивалентна на нея структурна Програма. Трансформирането е смислено, ако се налага модифицирането на съществуваща програма с усложнена или объркана управляваща структура; или ако новоразработваните програми ще са структурни и след преобразуването ще се постигне унифицирано представяне (и документиране) на всички програми.

Основно преимущество на структурното програмиране е повишаването на разбираемостта и тестируемостта на програмите и намаляване на усилията за тяхното разучаване и съпровождане. Твърди се, че усвояването и придръжането към точно определения структурен стил на проектиране и програмиране повишава производителността на проектантите и програмистите.

Основен недостатък на структурното програмиране е необходимостта от усвояване на тази специална програмистка техника и прилагането ѝ при създаване на всички програми. Консервативно настроените програмисти се съпротивляват на всяка промяна в стила им на работа и обикновено разглеждат опитите за въвеждане на стандарти като посегателство на личната им творческа свобода. Друг недостатък е, че в някои случаи структурните програми изискват по-големи изчислителни ресурси (памет и време) от съответните неструктурни програми.

3.5. Интегриране и тестване

Основните дейности по откриване и отстраняване на грешки — настройването и тестването на програми, са разгледани подробно в глава 5. на учебника. Тук ще се спрем накратко само на стратегиите за тестване на софтуерните системи. Стратегиите за тестване определят кои части на системата да се тестват, в какъв ред, с какви методи и средства, в каква среда и от кого.

Разглеждаме програмната система като йерархична структура, на най-ниското ниво на която са модулите. Обикновено се започва с модулно тестване, след което се преминава към тестване на компонентите от по-високо ниво. В зависимост от обекта и целите на тестване могат да се прилагат различни техники на тестване.

Модулно (поелементно) тестване

Основната идея на модулното тестване е да провери коректността на най-малките програмни компоненти — модулите. Те могат да бъдат разгледани, разбрани и проверени сравнително лесно и усилията за откриване и поправяне на грешките в тях не са големи. Препоръчва се тестването да започне с проверка на интерфейса (на входните и изходните данни) на модула и да продължи с проверка на логиката, обработката на данните и реализираните функции.

Още при проектирането някои от модулите могат да се идентифицират като „критични“ поради важността на реализираните функции, сложната структура или специални изисквания. Тестването на тези модули се планира и осъществява с отчитане на тези особености.

Модулното тестване изиска специална среда. Доколкото модулът не е самостоятелна програмна част, за да се изпълни, е необходимо да се създадат допълнителни програми — *драйвери* (*drivers*) за извикване на тествания модул и *опори* (*stubs*) за представяне на модулите, извиквани от тествания модул. Допълнителните разходи за разработването на драйвери и опори се компенсират от намаляването на грешките, за поправянето на които в по-късни фази ще са необходими значително повече време и средства.

Интеграционно тестване

Интеграционното тестване е систематична техника за конструиране на програмна структура от тествани вече елементи и организиране на тестване за откриване на интерфейсни грешки.

Съставянето на програмната структура за тестване може да става по два начина: *интегрално* или *инкрементално*. При първия начин след тестването на всички модули се сглобява цялата програмна система и започва системно тестване. Преимущество е, че не се разработват допълнително драйвери и опори. Основен недостатък е, че откриването и локализирането на грешки е много по-трудно поради възможността за наслагване на последиците от няколко интерфейсни грешки.

При инкременталното тестване се започва с група от два тествани модула и на всяка стъпка съществуваща до момента програмна структура се разширява само с един допълнителен модул. Така причините за новопоявяващи се грешки се установяват много по-лесно.

И двата начина могат да се реализират възходящо или низходящо. При възходящия подход се започва от най-долното ниво на йерархията (модула), образуват се подсистеми и накрая се достига до най-високото ниво — системата. При низходящото тестване се започва с тестване на системата като цяло и се продължава с тестване на елементите от по-ниските нива. Възходящото тестване изиска драйвери за симулиране на дейността на следващия компонент от по-високо ниво. Низходящото тестване изиска опори, имитиращи компонентите, подчинени на текущо разглеждания компонент. Всъщност най-често се прилага смесен подход, наречен „сандвичово тестване“, като до определено ниво в йерархията се прилага единият, а от там нататък — другият подход. Редът на прилагането им зависи от особеностите на конкретната софтуерна разработка.

Тестването може да се извършва от самите разработчици, от независима група по тестване в софтуерната организация или от външна сертифицираща фирма. Кой да извършва конкретни тестващи дейности, зависи от целите на тестването в даден момент.

Особено важни са т. нар. *приемни тестове*. Те са последна проверка на завършената софтуерна система преди разпространяването ѝ. *Алфа-тестването* се провежда с участието и на потребители, но в средата на разработване. *Бета-тества-нето* е изцяло в

реална потребителска среда, като потребителите регистрират проблемите и ги съобщават на разработчиците, които внасят съответните промени.

Литература

1. Sigfried, S., Understanding OO Software Engineering. IEEE Press, 1996.
2. Zahniser, R.A., Building Software in Groups, American Programmer, vol. 3, no7/8, July-August 1990.
3. Sommerville I. Software Engineering, Addison-Wesley Publ.Company, Fourth Edition, 1992.
4. Chen, P., The Entity-Relationship Approach to Logical Database Design, QED Information Systems, 1977.
5. Hatley, D.J., I.A.Pirbhai, Strategies for Real-Time System Specification, Dorset House, 1987.
6. Yourdon, E.N. and L.Constantine. Structured Design. Yourdoii Press, 1978.
- ? , Pressman, R. Software Engineering — A Practitioner's Approach. R.S. Pressman & Associates, Inc. 2000.
8. Вирт, Н. Алгоритми + Структури от данни = Програми. С. Техника, 1980.

4. ОБЕКТНО ОРИЕНТИРАН ПОДХОД ЗА РАЗРАБОТВАНЕ НА СОФТУЕР

Теорията и практиката на новата обектно ориентирана (OO) парадигма са се развили толкова, че могат да бъдат обект на отделен учебник, посветен на обектно ориентирани софтуерни технологии. В съответствие с целите, които сме си поставили, ще се спрем само на основните характеристики на ОО-подход и на особеностите, които го разграничават от конвенционалния подход за разработване на софтуер.

4.1. Основни понятия

Под *обект* се разбира познаваем предмет, елемент или същност (реална или абстрактна), имащ важно функционално предназначение в разглежданата приложна област. Всеки обект има състояние, поведение и индивидуалност. Структурата и поведението на сходни обекти определят общ за тях *клас*. Състоянието на обекта се характеризира с изброяване на всички възможни свойства на обекта и текущите стойности на тези свойства. Разглежданите свойства се наричат *атрибути*. Всеки атрибут има област на допустимите стойности. С всеки обект могат да се свържат *операции* (услуги или методи), които променят един или няколко атрибути на обекта. Обектът наследява всички атрибути и операции на класа, към който принадлежи.

Понятието клас е основно за ОО-подход. То обхваща (*encapsulate*) данните и алгоритмите (процедурите), чрез които може да се описе съдържанието и поведението на „нешо“ от реалния свят. Класът може да се разглежда като обобщено описание на съвкупност от подобни обекти.

Обектите в класа наследяват неговите атрибути и операции. *Суперклас* е съвкупност от класове, а *подклас* е екземпляр на клас. Съществува йерархия на класовете, като подкласовете наследяват атрибутите и операциите на супер-класа, но могат да имат и специфични за тях атрибути и операции.

Взаимодействието между обекти се осъществява чрез *съобщения* (*messages*). Съобщението предизвиква реакция и промяна в поведението на обекта получател. Той изпълнява посочената в съобщението операция и връща уп-

равлението на обекта подател. Чрез съобщенията се описва поведението на обектите и на ОО-система като цяло.

Три основни характеристики са присъщи на ОО-подход. Те са:

а) "Капсулиране" в класа на данните и операциите върху тях. Това има следните преимущества:

- подробностите на вътрешната реализация остават скрити;
- данните и операциите са обединени в едно именовано цяло — класа, което улеснява повторното му използване;
- връзките между „капсулираните“ обекти са опростени, защото осъществяването им чрез съобщения не зависи от вътрешните структури от данни.

б) *Наследяване*. Същността на наследяването е, че всеки подклас придобива автоматично (наследява) всички атрибути и операции на съответния суперклас. Така се осигурява повторно използване на проектирани и реализирани вече структури данни и алгоритми. Улеснено е внасянето на изменения, защото те се правят само в съответното

ниво в йерархията от класове и чрез наследяването се разпространяват. При необходимост от нов клас той може да се конструира изцяло или да се „вмъкне“ в съществуващата йерархия от класове, като се допишат само специфичните за него елементи. Допуска се и „множествено наследяване“ от няколко различни суперкласа. То е полезно заради по-големия брой наследявани свойства, но затруднява проследяването на връзките в йерархията.

в) *Полиморфизъм*. Това свойство дава възможност различни операции да имат едно и също име. Така се осигурява настроеваемост и гъвкавост, защото с унифицирано извикване могат да се реализират специфични обработки.

4.2. Обектно ориентиран анализ и проектиране

Целта на *обектно ориентиранния анализ* (OOA) е създаването на модел, представящ статичните и динамични характеристики на класовете и взаимоотношенията между тях. Създадени са различни методи за OOA, използващи различни съвкупности от диаграми и означения, но преминаващи през едни и същи стъпки, обобщени в [1], както следва:

1. Изясняване на потребителските изисквания за системата.
2. Идентифициране на потребителските сценарии (use-cases).
3. Избор на класовете и обектите въз основа на формулираните изисквания.
4. Идентифициране на атрибутите и операциите за всеки обект.
5. Дефиниране на структурите и йерархиите на класовете.
6. Построяване на модел, описващ обектите и връзките между тях (object-relationship model).
7. Построяване на поведенчески модел.
8. Проверка на 00 аналитични модели за съответствие с потребителските сценарии.

Проектирането на ОО-система изисква определянето на многоплаstова софтуерна архитектура, специфициране на подсистеми, които изпълняват исканите функции, описание на обектите (классовете), които са изграждащите блокове на системата, и описание на механизмите на комуникация, реализиращи потока на данните между слоевете, подсистемите и обектите.

ОО-проектиране се извършва на две нива, съответстващи на външното и детайлното проектиране при конвенционалния подход — проектиране на системата и проектиране на обектите.

При *проектирането на системата* се определя архитектурата на ОО-приложение. Създаденият аналитичен модел се декомпозира на подсистеми, като се описва предназначението на всяка подсистема и връзките между тях. Проектират се още потребителският интерфейс и логическата организация на данните (т. нар. компонента за управление на данните).

При *проектирането на обектите* за всеки обект (екземпляр на клас или подклас) се съставя *интерфейсно описание* и *описание на реализацијата* [2]. Интерфейсното (протоколно) описание определя всички съобщения, получавани от обекта, и съответните операции, които обектът извършва при получаване на съобщенията. Описанието на реализацијата (*implementation description*) специфицира свързаните с обекта структури данни и алгоритмите за всяка операция.

За стандартизирано описание на моделите в обектно ориентираното разработване на софтуер е създаден т. нар. Unified Modeling Language (UML) [3]. Основните части на UML са:

а) *Представяния.* Моделирането на сложните реални системи изисква описването на различни техни аспекти — функционални, нефункционални, организационни и др. Така системата има няколко представяния от различна гледна точка, като всяко представяне е проекция на цялостното описание на системата в съответствие с избран аспект (ракурс). Поддържаните от езика UML представяния са:

- потребителско (use-case view) — това представяне показва функционалността на системата от гледна точка на потребителите ѝ;
- логическо — това представяне показва как функционалността е проектирана за вграждане в системата — в термините на статичната структура на системата и динамичното ѝ поведение;
- компонентно — показващо организацията на програмните компоненти;
- конкурентно — описващо комуникационните и синхронизационни проблеми в конкурентните системи;
- обвързващо (deployment view) — това представяне описва разполагането на системата в конкретна компютърна среда.

б) *Диаграми.* Диаграмите са графи, описващи съдържанието на различните представяния. Езикът UML има девет различни типа диаграми (use-case diagram, class diagram, object diagram, state diagram, sequence diagram, collaboration diagram, activity diagram, component diagram and deployment diagram), чрез съчетаването на които се описват всички представяния на системата.

в) *Елементи на модела.* Използваните в диаграмите концептуални обекти се наричат елементи на модела. Те представлят основните обектно ориентирани елементи като клас, обект, съобщение и връзките между тях. Всеки елемент на модела може да се използва в различни диаграми, но смисълът и означението му не се променят.

г) *Общи механизми.* Те осигуряват допълнителна информация за семанти-ката на елементите на модела или как да се разшири моделът за специфични процеси, системи или организации.

Езикът UML може да се използва за моделиране във всички фази на разработването и за всякакви софтуерни системи. Използването му се подпомага от инструментални средства (индивидуални или интегрирани), които изчертават диаграмите, съхраняват информацията за всички модели и представянията им, осигуряват многопотребителска работа и дори могат да генерират автоматично код.

4.3. Обектно ориентирано програмиране и тестване

Програмирането представлява превърщане на ОО-проект в програмен код. Класовете, дефинирани при проектирането, трябва да се описват като класове в съответния ОО-език за програмиране като C++, Java или Smalltalk. Резултатът е програма, която може да се изпълнява. Особеностите на процеса на съставяне на програми могат да се намерят в ръководствата за съответните ОО-езици.

Целта на ОО-тестване е същата, както и при стандартния подход — да се открият колкото се може повече грешки в рамките на определените за тестването бюджет и време. Поради специфичните черти на ОО-подход обаче съдържанието на основните дейности, свързани с тестването, е различно.

Преди всичко е необходимо да се разшири обхватът на тестването така, че то да започва от най-ранните фази и първите обекти за тестване да бъдат моделите на системата, създадени след ОО-анализ и ОО-проектиране. Причината за това е, че всяка неоткрита грешка в тези модели би довела до грешки в програмите, които трудно биха се открили. Например основните семантични конструкции (класове, атрибути, операции и съобщения) се определят в модела, създаден след ОО-анализ, и неподходящият избор на всеки от тези елементи би довел до допълнителни и може би излишни усилия при проектирането и програмирането. Затова ОО-тестване включва формалните проверки за *правилност, пълнота и непротиворечивост* в контекста на синтаксиса, семантиката и използването на моделите, създадени след ОО-анализ и проектиране. Техники, определящи какво и как да се проверява, са описани в [4].

Друга съществена разлика е в *стратегията за тестване*, осигуряваща последователно разрастване на обектите, които ще се тестват. Така традиционното модулно тестване тук е *тестване на клас*, защото в ОО-контекст класът е най-малката тестируема единица, капсулирала данните (атрибутите) и операциите над тях. Вместо проверка на алгоритъма, както е в модулното тестване, при тестването на клас се проверяват операциите и промените в състоянията на класа.

Вместо традиционното интеграционно тестване на съставена по някакви Правила съвкупност от модули в ОО-системи се тества *съвкупност от класове*. В зависимост от начина на съставяне на съвкупността от класове съгласно [5] интеграционното ОО-тестване може да бъде:

- проследяващо (thread-based) — тестват се всички класове, свързани с едни и същи събития в системата;
- пластово — разглеждане на йерархията от класове и разделяне на класовете на независими (използващи само някои други класове) и последователни слоеве от зависими класове;
- кълстерно — съставяне на кълстери (групи) от взаимодействащи класове и тестване на тези кълстери.

Системното ОО-тестване съвпада с традиционното, защото целта му е да Изследва поведението на системата като цяло, без да се разглежда реализацията.

Генерирането на тестови данни зависи от метода на тестване (стресово, случайно, сценарийно) и от обекта на тестване — отделен клас или група от класове. Подробности могат да бъдат намерени в [5, 6].

4.4. Управление на обектно ориентираното разработване

Както ще видим в глава 9., стихийността в софтуерното производство се преодолява с прилагане на основните управленски техники. За разработването на ПП те са за управление на: процеса на създаване на софтуер, създавания продукт, софтуерния проект и човешкия фактор.

4.4.1. Управление на процеса на разработване

Както беше описано в глава 2, препоръчва се всяка софтуерна организация да избере модел на жизнения цикъл на ПП, който идентифицира подхода за разработване и определя последователността и съдържанието на основните фази и/или функции. За ОО-разработване Г. Буч [7] предлага т. нар. рекурсивно-паралелен модел, схематично представен на фиг. 4.1. Той е много близък до спираловидния и до еволюционния модел, споменати в глава 2., и същността му е в разработването на последователно разрастващи

и усложняващи се прототипи до окончателното изграждане на целевата система. За всеки прототип се извършват едни и същи дейности: планиране, анализ, проектиране, извлечане на компоненти от библиотеката за повторно използване, конструиране на прототипа с налични и новоразработени компоненти, тестване и оценка от потребителя, определящ изискванията към следващия прототип.

Моделът се различава от останалите подобни модели по *рекурсивното* повтаряне на дейностите (включително и на анализа и проектирането!) за всеки прототип и по възможността за *паралелното* им извършване за независимите компоненти на системата.

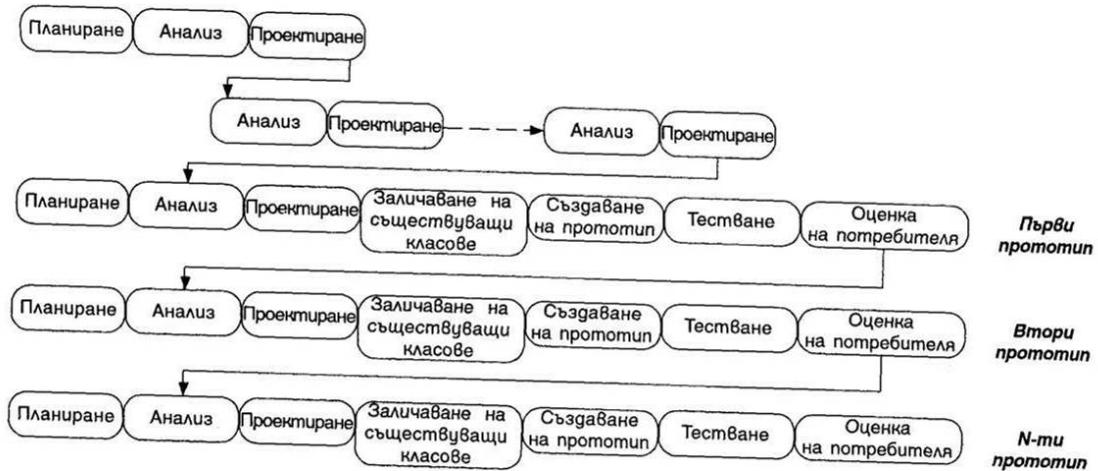
За ефективно управление на рекурсивно-паралелното разработване ръководителят на проекта трябва да осъзнае, че планирането (определянето на стандартните задачи и графици) и измерването на развитието на проекта се извършва за всяка независима компонента поотделно.

4.4.2. Управление на проекта и продукта

Стандартните техники за управление на софтуерни проекти (вж. глава 9.) са приложими и при ОО-разработване, но съдържанието на някои от извършваните дейности е различно. Ще споменем някои съществени разлики:

- а) допълване на стандартните методи за *оценяване на цената* на разработване, трудоемкостта и продължителността на софтуерен проект с разработените ОО-метрики, измерващи специфични за подхода елементи — сценарии на използване, основни и поддържащи класове и връзките между тях [8];
- б) за проследяване на развитието на проекта се препоръчва стандартната техника с използване на контролни точки, но достигането им се определя по специфичен начин В [1] са описани критерии за определяне, кога е завършил ОО-анализ (контролна точка 1), ОО-проектиране (контролна точка 2), програмирането (контролна точка 3) и тестването (контролна точка 4). Например критериите за завършване на ОО-анализ са:

- Всички класове и юерархията на класовете са дефинирани и проверени.
- Атриутите на всеки клас и свързаните с него операции са дефинирани и проверени.
- Отношенията между класовете са описани и проверени.
- Поведенческият модел на системата е създаден и проверен.
- Класовете за многократно използване са идентифицирани.

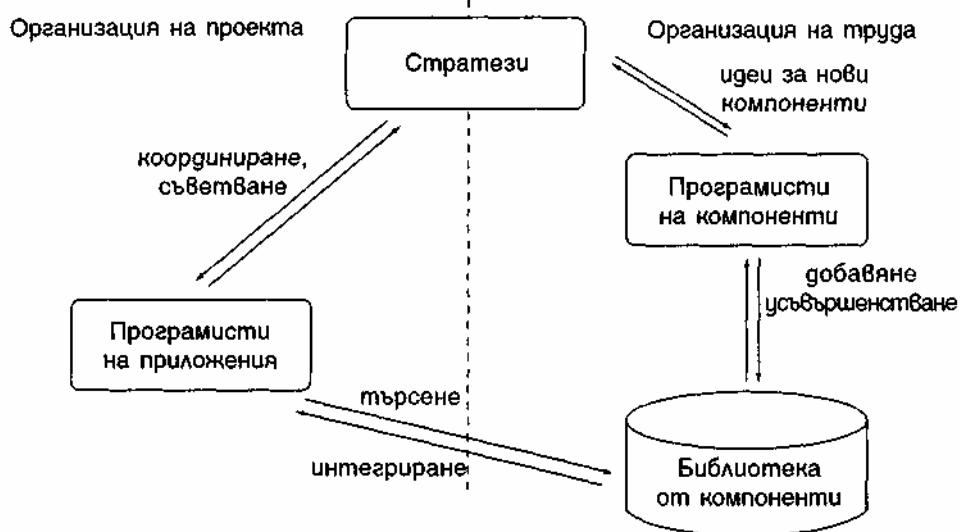


Фиг. 4.1.

Аналогични критерии са зададени и за останалите контролни точки.

в) управлението на проекта и управлението на продукта се преплитат поради специфичното покомпонентно разработване при ОО-подход.

Една от особеностите на ОО-разработване е, че всяко приложение може да се изгражда чрез компоненти. Под *компонента* се разбира клас, който е създаден и съхранен, с определено ниво на качество и степен на общност, която позволява да се използва многократно.



Фиг. 4.2.

На фиг. 4.2. е показана схема, илюстрираща управлението на проекта и продукта при ОО-разработване с използване на библиотека от компоненти. Организи-зацията на

проекта е подчинена на краткосрочни цели като завършване на проекта в определения срок и в рамките на бюджета. Организацията на продукта преследва по-дългосрочни цели — създаване на компоненти за многократно използване. Поради специалните изисквания към тези компоненти (да са с проверено и високо качество, да решават клас от сродни проблеми, да са подробно документирани) цената на разработването им е по-висока. Това се компенсира с намаляването на цената за изграждане на приложения и намаляване на усилията за съпровождане.

4.4.3. Управление на персонала

Както е показано на фиг. 4.2., разграничени са три групи изпълнители на 00 софтуерен проект: програмисти на приложения, програмисти на компоненти и стратеги.

Програмистите на приложения трябва да имат знания за приложната област и способност за абстрактно мислене. Те трябва да определят кои обекти трябва да се групират, за да се изгради приложението. Затова те се интересуват не от детайлите и реализацията на компонентите, а от същността им и за какво могат да се използват.

Програмистите на компоненти трябва да проектират и реализират всяка компонента така, че тя да е настроеваема и да се използва лесно. Независимо че отговарят за изграждане и поддържане на библиотеката от компоненти, те участват активно в началото на всеки проект, за да дадат информация, какви налични компоненти има и как могат да се използват.

Основните функции на стратегите са координиране и съветване. Те трябва да имат необходимата квалификация и опит в повторното използване (вж. глава 13) като основен подход в софтуерната организация. Разполагайки със знания за достъпните компоненти и начина им на вграждане в приложения, стратегите са посредници между двете групи, като решават и по-глобални въпроси: какво да бъде съдържанието на библиотеката от компоненти, как да се оптимизира описанието и търсенето на нужните компоненти в нея, какви автоматизирани средства да се използват и т. н.

Различни са изискванията към квалификацията, опита, знанията и уменията на трите категории софтуерен персонал и разпределението на задълженията трябва да е съобразено с тях.

4.5. Въвеждане на обектно ориентириания подход

Преминаването към ОО-разработване изиска планирани и систематични действия, успехът на които зависи от много фактори. Обобщавайки идеи от [9], ще изброим някои препоръки, следването на които би улеснило процеса на преход:

— Обучение на персонала на софтуерната фирма на всички нива. Ръководството на фирмата трябва да осъзнае предимствата на подхода и да осигури ясни цели, подкрепа и финансиране за периода от време, необходим за постигане на технологично ниво. Ръководителите на проекти трябва да бъдат специално обучени, за да разберат и да могат да прилагат ОО-технология. Най-сериозна трябва да бъде преквалификацията на разработчиците чрез осигуряване 8а подходяща литература, организиране на специални курсове и семинари за обмяна на опит с колеги от други софтуерни фирми.

— Стартiranе на пилотен проект за проверка, как разучените ОО-техники ще се прилагат в рамките на организацията. Размерът и сложността на пилотния Проект трябва да бъдат такива, че той да се реализира от не повече от петима Разработчици и за сравнително кратък период от време (от 3 до 5 месеца). За Планиране и оперативно

управление на проекта може да се привлече опитен специалист, който да ръководи проекта и да осигури документирането на възникващите проблеми и успешни решения, така че да се натрупва know-how информация. За следващите проекти се препоръчва прилагането на съвкупност от метрики за измерване на подобренията в процеса на разработване и производителността.

— Осигуряване на плавен преход от функционално ориентиран към ОО-начин на мислене. От когнитивна гледна точка възможностите на хората да възприемат нови идеи са различни и това трябва да се отчита при сформирането на екипите. Всички основни дейности трябва да се преразгледат и ако е необходимо, да се преформулират съдържанието и начинът на извършване на някои от тях. Например целите на обсъжданията между потребителите и разработчиците при анализа на изискванията при традиционния и ОО-подход са различни и това променя изцяло принципите на организирането им.

— Избиране на подходящи методи за разработване на софтуер и адаптирането им към нуждите на организацията. Критерии за избора могат да бъдат: история на създаване и използване на метода, ниво на зрелост и стабилност, степен на универсалност, адекватност с ОО-подход, сложност на усвояване и прилагане, гъвкавост, степен на автоматизираност, ползваемост и др. Разработени са техники (например въпросници), които могат да се използват при анализа и оценяването на даден метод.

Въвеждането на ОО-подход изиска предварителна подготовка и оценка на разходите и ползите за организацията, като единственият начин за проверката им е практическата реализация.

Литература

1. Pressman, R. Software Engineering — A Practitioner's Approach. R.S. Pressman & Associates, Inc. 2000.
2. Goldberg, A., D.Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.
3. Eriksson, H., UML Toolkit, Wiley Computer Publishing, 1998.
4. McGregor, J., T.Korson, Integrated Object-Oriented Testing and Development Processes, CACM, vol. 37, no. 9, September 1994, pp. 59—77.
5. Binder, R., Testing Object-Oriented Systems: A Status Report, American Programmer, vol. 7, no. 4, April 1994, pp. 23—28.
6. Marick, B., The Craft of Software Testing, Prentice Hall, 1994.
7. Буч, Г. Объектно-ориентированное проектирование с примерами применения. Изд. Конкорд, Москва, 1992.
8. Lorenz, M., J.Kidd, Object-Oriented Software Metrics, Prentice-Hall, 1994.
9. Sigfried St., Understanding Object-Oriented Software Engineering, IEEE Press, 1996.

5. ДЕЙНОСТИ, ОСИГУРЯВАЩИ РАЗРАБОТВАНЕТО НА ПРОГРАМНИ ПРОДУКТИ

Процесът на създаване на софтуер може да се разглежда като съвкупност от основни дейности (напр. дефиниране на изискванията, проектиране, програмиране) и допълнителни дейности. Характерно за допълнителните дейности е, че те са „фонови“ и се осъществяват в продължение на няколко или дори всички фази на жизнения цикъл. Типични допълнителни дейности са:

- откриване на дефекти;
- управление на проекта;
- осигуряване на качеството;
- измерване;
- документиране;
- управление на софтуерните конфигурации.

Ще разгледаме някои допълнителни дейности, които са съществени за създаването на софтуера и на които не са посветени отделни части на учебника.

5.1. Откриване на дефекти

Откриването на дефекти е част от цялостното осигуряване на качеството на софтуера (вж. глава 8.). Поради изключителната му важност ще го разгледаме като отделна дейност.

5.1.1. Основни понятия

Качеството на програмните продукти е абстрактно понятие. Твърди се, че то трудно се дефинира, но е лесно да се разпознае при реалното използване на съответния програмен продукт. Една от трудностите при дефинирането е относителният характер на качеството. Могат да се избират различни дефиниции в зависимост от интересуващия ни аспект на качеството, от това кой го оценява и Кои са обектите, чието качество се изследва. В по-нататъшното изложение ще се придържаме към следната дефиниция:

Качествен е този програмен продукт, който удовлетворява формулираните към него изисквания.

Всяко отклонение от изискванията ще наричаме *дефект*.

Дефектите обикновено се дължат на една или няколко грешки.

Под *грешка* ще разбираме неправилност, отклонение или неволно преиначаване на обект или процес.

В зависимост от това, в какъв софтуерен продукт се откриват, грешките могат да бъдат грешки в проекта, в програмата, в документацията, в тестовите данни и т. н.

По-нататък ще разгледаме само грешките в програмите, дейностите за откриването и отстраняването им и средствата, подпомагащи тези дейности.

Според Майерс в програмата има грешки, ако тя не изпълнява това, което потребителят разумно очаква от нея.

Грешките могат да бъдат:

— **първични** — неправилности в текста на програмите, подлежащи на непосредствено коригиране;

— **вторични** — изкривявания на получените резултати (например зацикляне, дължащо се на първична грешка непроменяне на стойността на управляващата променлива за цикъла в неговото тяло).

Съгласно друга класификация грешките могат да бъдат:

а) **технологични** грешки — при въвеждане на програмите или при подготовка на входните данни върху технически носители;

б) **алгоритмични** грешки;

в) **програмни** грешки — неправилно използване на конструкциите от съответните езици за програмиране;

г) **системни** — свързани с функциониране в определена операционна система.

В зависимост от вида на грешките се прилагат различни техники за търсенето им.

5.1.2. Основни дейности за откриване и отстраняване на грешки

Двете основни дейности, свързани с откриването и отстраняването на грешки в програмите, са настройване и тестване.

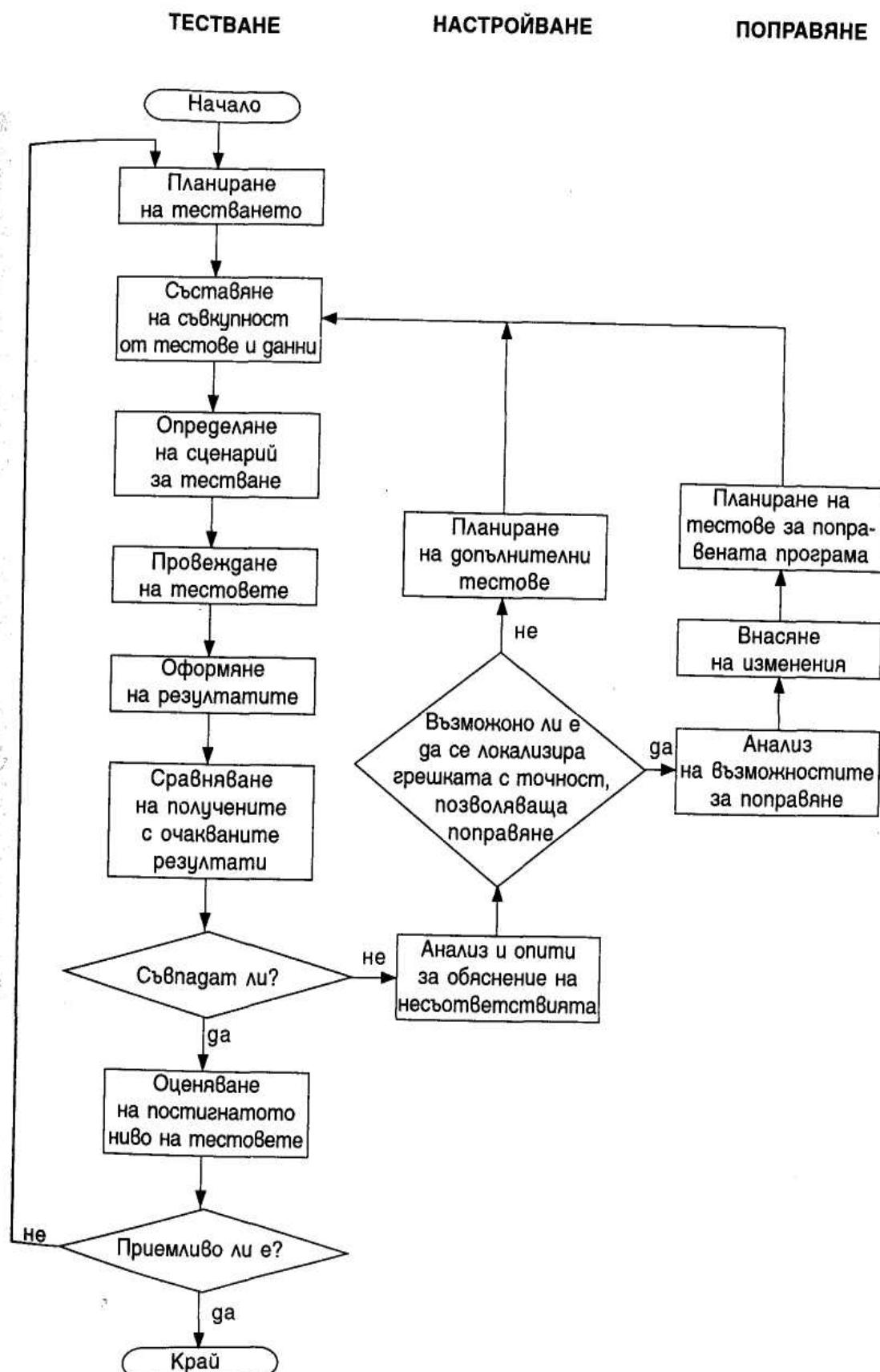
Настройване (debugging) ще наричаме локализиране и отстраняване на установени грешки.

Тестване (testing) ще наричаме изследване на програмите за установяване на съответствието им с различни по степен на формализираност характеристики, правила и изисквания.

Дейностите настройване и тестване се различават по основното си предназначение, по използваните методи и по нивото на сложност на откриваните грешки.

Когато настройването завърши, е ясно, че програмата решава някакъв проблем. Предназначенето на тестването е да докаже, че точно това е проблемът, който се иска да бъде решен.

Връзката между тестването, настройването и поправянето на грешки може да се илюстрира чрез схемата на фиг. 5.1.



Фиг. 5.1.

Тестването се осъществява в три основни стъпки: планиране, реализация и отчитане.

При *планирането* на тестването се определя целта на тестването, какво да се тества, кога, с какви данни, как и кой да го извършва.

Реализацията на тестването се описва чрез сценарии за тестване.

Отчитането се извършва чрез анализ на документираните резултати и прилагане на критериите за изчерпателността и обхвата на тестване.

Има два основни подхода за тестване: *структурен* и *функционален*.

Целта на *структурното тестване* е да се изберат такива тестови данни, че да се осигури преминаване през всички програмни части на системата.

При *функционалното тестване* се проверява правилността на реализираните основни функции.

В зависимост от избраните тестови данни и очаквани резултати тестването се дели на:

- *детерминирано тестване* — при зададени входни данни напълно е определено какви трябва да бъдат получените резултати;
- *стохастично тестване* — тестваните данни са случаини величини с определено разпределение и се знае разпределението на получаваните резултати.

В зависимост от *начина на осъществяване* тестването може да е *низходящо* (top down), *възходящо* (bottom up) и *смесено* — започва от някакво междинно ниво и се провежда в двете посоки.

Тестването би трявало да започне в най-различни етапи на създаване на програмите, като още тогава да се формулират критериите за установяване на желаното ниво на тестване.

В зависимост от целта тестването може да бъде:

- *за доказване на експлоатационната годност* — проверява се дали програмният продукт е работоспособен;
- *за атестация* — т. нар. пускови изпитания, след успешното завършване на които програмният продукт може да се разпространява;
- *за пълна функционална проверка* — дали реализираните функции съответстват на формулираните изисквания;
- за проверка на *специални програмни свойства*, например надежност или преносимост;
- за проверка на *нови свойства или функции* — реализира се след внасяне на изменения в създадения програмен продукт и се нарича регресионно тестване;
- *за проверка на работоспособността на системата в реални потребителски условия* — изследва се времето за отговор на системата, продължителността на входно-изходните операции, изчислителните ресурси; качеството на потребителския интерфейс и други.

В зависимост от това, кой извършва тестването, то може да бъде:

- *вътрешно тестване* — от самите разработчици. Всеки програмист проверява правилността на създадените от него програми. Преимущество на този подход е, че не е необходимо разучаване на програмите. Недостатък е, че програмистите имат склонност към надценяване на възможностите си и увереността, че не допускат грешки, а това

пречи на системното тестване. Форма на вътрешно тестване е т. нар. *peer review* — проверка, при която програмистите са разделени на двойки и всеки проверява програмите на партньора си.

— *независимо тестване* — от експерти, които не са участвали в разработването на програмите. Обикновено това са членове на групата по тестване, действащи в софтуерната фирма.

— *сертифициране* — проверка и издаване на сертификат, удостоверяващ наличието на определени свойства на ПП. Сертифицирането може да се извърши само от фирми, притежаващи съответния лиценз.

Един от най-сложните проблеми е да се определи докога да се тества. Обикновено тестването завършва при изтичане на предвидения срок или изчерпване на ресурсите. Специалистите по тестване твърдят, че „недостатъчното тестване е престъпление, а прекомерното — грях“.

5.1.3. Автоматизиране на дейностите настройване и тестване

Създадени са множество инструментални средства, които подпомагат дейностите по откриване и отстраняване на грешки [1]. Чрез автоматизация на настройването и тестването се постига:

- систематичност;
- подобряване организацията на тестване;
- повишаване надежността на програмната система;
- документиране на извършваните дейности;
- автоматично измерване обхвата на тестването.

Инструменталните средства, подпомагащи настройването, се наричат програми — *дебъгери*.

Основните възможности на дебъгерите са следните:

а) *разглеждане на текста на програмата* на различни нива на декомпозиране (показване само имената на функциите или процедурите, показване съдържанието на отделни програмни фрагменти). Съвременните дебъгери реализират *многопрозоречно визуализиране* на интересуващите ни програмни части.

б) *трациране на изпълнението* — показване на операторите в реда на изпълнението им при конкретните тестови данни;

в) възможност за дефиниране на *контролни точки* и определяне на действията при достигане на съответната контролна точка. Тези действия могат да бъдат:

- преустановяване изпълнението на програмата;
- разпечатване съдържанието на определени области от паметта или на указанi променливи;
- продължаване изпълнението на програмата от указан оператор.

В зависимост от предназначението си програмните средства за тестване могат да бъдат анализатори на програми, генератори на тестови данни, помощни средства и среди за тестване.

а) *Анализатори на програми* — предназначението им е да реализират избран метод на тестване чрез опериране върху тестващата програма по съответстващ на метода начин.

Статичният анализ е изследване на текста на програмите и извличане ИА определена информация от него. Статичният анализ може да бъде *елементарен* или *потоков* (статичен анализ на потока на данните и потока на управлението). При елементарния статичен анализ текстът на програмата се проследява, като се създава таблица на използваните имена и срещанията им (cross-reference). Създава се и тъй нареченият *статичен профил* на програмата, показващ кои оператори са използвани и колко пъти.

При *потоковия анализ* се построява *управляващ граф* на програмата и *граф на обръщенията*, отразяващ връзките между отделните програмни части. Потоковият анализ дава възможност да се открият няколко вида *грешки*:

- структурни — недостижими програмни части, недопустими пресичания в телата на цикли, рекурсивни обръщания към подпрограми, когато това не е разрешено;
- в дефинирането и използването на променливи — използване на променлива, преди да е получила начална стойност, дефиниране на променливи, които по-нататък не се използват, установяване на случаи при цикли, управлявани от логически израз, в които никоя от променливите, участващи в израза, не променя стойността си в тялото на цикъла;
- несъответствия и грешки при предаване на параметри между различни програмни части и използването на тези програмни части и параметри.

Обикновено потоковият анализ се извършва на две нива:

- *локално* — за всяка програмна единица;
- *глобално* — за програмната система като цяло.

Съобщенията за грешки и документиращата информация съответстват на тези две нива.

Потоковият анализ има два *недостатъка*:

- дълбочината на анализа се постига с цената на *големи изчислителни ресурси*, които са пропорционални на размера на изследваните програми;
- поради наличието на условни или изчислителни преходи откриваните аномалии се делят на *установени* и на *предполагаеми*, т. е. на такива, които непременно ще се случат, и такива, които биха могли да се случат. Пример, конструиран може би изкуствено, показва, че след статичен анализ на програма от 760 оператори са обявени 20 установени и 1700 предполагаеми аномалии.

Основни *тенденции* при статичните анализатори са разработването им за програми, написани на различни езици за програмиране, и включването на статичните анализатори в интегрирани системи за тестване.

При *динамичния анализ* програмата се изпълнява по управляем и систематичен начин и се изследва нейното поведение, за да се потвърди функционалната ѝ коректност или некоректност.

При планирането на динамичния анализ се избира стратегия, подготвят се съответстващи на стратегията тестови данни и се определя начинът на тълкуване на получаваните резултати.

Исторически първият реализиран метод за динамичен анализ е бил тъй нареченият *подход на черната кутия*, при който изследваната програмна част се разглежда като елемент с неизвестно съдържание. Към нея се подават входни данни и след изпълнението се получават определени резултати. За съжаление при този подход има възможност за установяване на несъответствията, без да се изясняват причините за тях. Така се достига

до идеята за реализиране на „*бяла кутия*“ — програмата се инструментира по такъв начин, че да се натрупва информация за поведението ѝ по време на изпълнение.

При *метода на пробите* за всеки изпълним оператор се дава броят на изпълненията му при конкретни входни данни. Тази справка показва не само

каква част от програмата е тествана, но и кои са най-често използваните програмни части. Те могат да бъдат обект на по-нататъшна оптимизация. Развитие на този метод е програмата да се инструментира така, че след изпълнението ѝ да се създаде *анотиран листинг*, в който за всеки изпълним оператор освен броя на изпълненията му се дава и допълнителна информация (например за оператора на присвояване — каква е минималната и максималната присвоявана стойност, колко пъти се е променила стойността и т. н.).

Друг подход е *вграждане в програмата на твърдения*, които първоначално са като коментари, но могат да се активизират за целите на динамичните анализатори. Тези твърдения описват някакви условия и какви да бъдат действията при наруширането им.

Формално-функционалните анализатори реализират „символично изпълнение на програмата“. Програмата се разглежда като *изчислима функция*, която може да се декомпозира на частични функции, изчислявани върху логическите пътища в програмата с подмножество на входните данни.

Сложността на тези анализатори е близка до сложността на системите за доказване на правилността на програми и затова практическото им използване засега е ограничено.

Мутационните анализатори генерираят мутанти на изследваните програми чрез *внасяне* на различни по вид и сложност *грешки*. След това се изследва приложимостта на избрана съвкупност от тестови данни и чувствителността на избран метод към типа на внесените грешки.

б) Генератори на тестови данни

Тези програмни средства подпомагат или реализират съставянето на тестови данни в съответствие с избрана стратегия на тестване.

Например при структурните стратегии на тестване проблемът е да се изберат тестови данни, които ще предизвикат изпълнението на неизпълнявана досега програмна част. Ясно е, че автоматичното генериране на тестове в този случай изисква сложен анализ с цел да се определи дали има осъществим път от началото на програмата до тази програмна част; от стойностите на кои променливи зависи изпълнението ѝ и т. н.

в) Помощни програмни средства

Те подпомагат определени стъпки от процеса на тестване — планиране, оценяване на постигнатото ниво на тестваност, реализиране на определен сценарий на тестване, документиране и др.

г) Интегрирани системи за тестване

Интегрираните системи (понякога наричани среди за тестване) могат да подпомагат няколко метода на тестване или да реализират изцяло избран метод. На тестване, като автоматизират планирането, генерирането на тестови данни, Управляемото изпълнение, анализа на получаваните резултати и оценяване на ефективността на тестването [2].

5.1.4. Осъществяване на тестването

Натрупаният практически опит показва, че резултатите от тестването установяват наличие на грешки и дефекти твърде късно — тогава, когато програмният продукт е

почти завършен. Доказано е, че усилията за отстраняване на грешките растат експоненциално в зависимост от дължината на времевия интервал t , определен от момента на допускане на грешката t_l до момента на откриването на грешката

$$E = k \cdot e^{-kt}$$

Затова се разработват специални методи и техники, позволяващи ранното откриване и отстраняване на грешки.

5.2. Съпровождане

5.2.1. Същност на съпровождането

Под *съпровождане* (*maintenance*) се разбира съвкупността от дейности, свързани с внасяне на промени във внедрен софтуер.

В зависимост от целите на модифицирането съпровождането може да бъде:

- коригиращо — за поправяне на установени грешки;
- адаптивно — за приспособяване към нова операционна или хардуерна среда;
- усъвършенстващо — за подобряване на съществуващи или добавяне на нови функционални възможности.

Статистическите данни показват, че 25% от усилията за съпровождане са за коригиращо, 50% за адаптивно и 25% за усъвършенстващо съпровождане. Това разпределение зависи от много фактори (тип и сложност на софтуера, приложна област и др.), но като цяло разходите за съпровождане са големи — между 65% и 75% от всички разходи. Много софтуерни организации са принудени да ограничават разработването на нови продукти поради голямата трудоемкост на съпровождането на съществуващите [3].

5.2.2. Осъществяване на съпровождането

Съпровождането може да се реализира от разработчиците или от други лица или организации въз основа на договор за съпровождане. Преимуществата на съпровождане от разработчиците е, че те познават добре проекта, функциите, съдържанието и структурата на програмите. Знаейки, че те ще отговарят и за съпровождането, още в процеса на разработване софтуерната система може да се създава по начин, който да улеснява внасянето на изменения. Сформирането на нова група по съпровождането пък осигурява по-обективна оценка на съществуващата система и възможност за оригинални идеи за модифицира-нетр. И в двата случая ролята на документацията е съществена за продължителността и трудоемкостта на съпровождането.

Основните обобщени етапи на съпровождането са:

- разучаване на съществуващия софтуер;
- модифициране на съществуващия софтуер;
- проверка на правилността на модифицирания софтуер.

5.2.3. Управление на софтуерните конфигурации

Под *софтуерна конфигурация* се разбира съвкупността от всички елементи, необходими за функционирането на даден ПП.

Това са първични текстове на програмите, обектен и изпълним код, командни файлове или процедури, необходими за свързване и изпълнение на програмната система, използвани системни файлове, помощни средства, файлове с данни и др. Изключително важна е съпровождащата документация, включваща описание на програмите, и експлоатационната документация, включваща ръководство за потребителя, ръководство за инсталација и др.

За съпровождането на всеки ПП трябва да се състави описание на софтуерната му конфигурация и на връзките между елементите ѝ, така че при всяка промяна в един елемент да може да се проследи кои други елементи са засегнати. Управлението на софтуерната конфигурация започва в началото на проекта и продължава в процеса на разработване [4].

При съпровождането се осъществява управление на заявките за изменение, контрол на версийте и управление на внасянето на изменениета.

Обикновено се поддържа стандартен формуляр — заявка за изменение, в който се описва исканото изменение и се обосновават причините за него. Специален експертен съвет разглежда постъпилите заявки и за всяка от тях определя дали да се отхвърли, дали да се включи в подготвяната нова версия или да се отложи за някаква следваща версия. Критерии за групиране изменениета на поредна версия могат да бъдат:

- а) подреждане на изменениета по технически или процедурни причини;
- б) подреждане по неотложност на извършване на промените;
- в) оценяване на необходимите ресурси за реализиране на изменениета;
- г) анализиране степента на отражение на промените върху останалите части от системата. Би трябвало да се минимизира броят на засегнатите от изменениета програмни части, като тези, от които зависи работоспособността на системата, да се променят само след доказана необходимост и наличие на ресурси;
- д) сходните промени да се групират в една версия.

Задължително е реализирането на приемственост между версийте. Това означава, че всяка нова версия трябва да реализира всички функциите на предишната и евентуално да предлага нови.

Трябва да се осигури и пълно съответствие между ПП и документацията му. Обикновено се поддържа бюлетин за изменениета. В него на достъпен за потребителите език се описват промените в новата версия. Ако броят на бюлетините е много голям, се препоръчва преиздаване на потребителската документация.

5.2.4. Управление на внасянето на изменения

Систематичното и управляемо внасяне на изменения преминава през следните пет етапа:

Eтап 1. Идентифициране на проблема и възлагане

Popълва се формуляр, в който се описват одобрените изменения с присвоения им приоритет. Оценяват се обемът и сложността на работата и се възлага на един или няколко изпълнители.

Eman 2. Проектиране на измененията

Анализират се възможните начини за осъществяване на измененията и се определя кои програмни части ще бъдат модифицирани.

Eman 3. Програмиране и тестване

През този етап съответните програми се изменят и се тестват локално.

Eman 4. Интегриране на програмната система и системно тестване

Извършва се регресионно тестване и пълни приемни изпитания, след успешното приключване на които версията се пуска за разпространение. През този етап потребителите се запознават с бюлетина за изменение и ако е необходимо, се провежда обучение за работа с новата версия.

Eman 5. Разпространение на новата версия

Всеки от потребителите на ПП може да получи новата версия и съответната документация. Конкретните финансови условия на разпространение зависят от съдържанието на новата версия и от регламентираните отношения между собственика на ПП и потребителите.

5.2.5. Разходи и цена на съпровождането

Изследвани са факторите, влияещи на разходите за съпровождането: приложна област, продължителност на използване на програмната система, степен на зависимост от външната среда. Някои от идентифицираните разходи са следствие от методите и средствата, използвани при разработването — език и стил на програмиране, тестване и валидиране на програмите, качество на програмната документация и др.

Белади и Леман [5] предлагат модел на процеса на съпровождане. Съгласно този модел, ако една програмна система се променя непрекъснато, то нейните обем и сложност нарастват. Причините за това са няколко. Първо, поправянето на едни грешки може да доведе до внасяне на нови. Второ, след всяко изменение системата се отклонява от първоначалния проект, който е обмислен и оценяван внимателно. Трето, съпровождането се извършва от специалисти с различна квалификация, които невинаги могат да разберат системата така, че да я променят оптимално.

Предложена е следната оценка за разходите на съпровождане:

$$M = p + K^{c-d},$$

където:

M са общите разходи за съпровождане на програмната система.

Стойността на p представя продуктивните усилия: анализ, проектиране, програмиране и тестване.

c е мярка на сложността, предизвикана от липсата на структурно проектиране и документиране. Тази сложност се редуцира с d — степента, до която Групата по съпровождане е запозната със софтуера.

Константата K е емпирична, като стойността ѝ зависи от средата. Тя се определя чрез регресионен анализ на разходите за съпровождане на реални проекти.

Ако системата е разработена без прилагане на основните принципи на софтуерните технологии, стойността на параметъра c ще бъде висока. Ако освен това се съпровожда без пълно разбиране на системата, стойността на d ще бъде ниска. В резултат разходите

за съпровождане ще нарастват експоненциално. Следователно намаляването им може да се постигне чрез осигуряване на характеристиката на качеството *съпровождаемост* в процеса на разработване и чрез предоставяне на достатъчно време за разучаване на системата, която ще се модифицира.

5.2.6. Автоматизирани средства, подпомагащи съпровождането

Някои от съществуващите софтуерни средства могат да се използват и при съпровождане. В зависимост от предназначението си те могат да се класифицират в следните групи:

- а) средства, улесняващи разучаването на програмите. Те реализират преформатиране в стандартен вид, създаване на справки за срещаните имена, статичен анализ на потока на данните и потока на управление и др.
- б) средства, улесняващи анализа и проектирането на внасяните изменения, например чрез прилагане на метрики върху оригиналната и променената програма;
- в) средства за управляемо внасяне на промените, като се следи за правата на достъп, автоматично се документират промените, управлява се последователността на осъществяване, регистрирането и оптималното извършване на модифицирането;
- г) средства за провеждане и анализ на резултатите от регресионното тестване;

Съществуват и среди за съпровождане, интегриращи няколко средства и поддържащи собствени бази от данни и механизми за управление на промените. Пример за такава среда е системата Lifespan, разработена от Yard Systems. Съхраняваните компоненти са програми, процедури, тестови данни или документи. Те имат номер на версията и са групирани в пакети. Позволява се внасянето на промени в копие на съхранявания компонент, с автоматично създаван коментар кой, кога и защо е променил компонентата и кои са компонентите, които трябва да се изследват, дали и как са засегнати от промяната.

5.3. Документиране

Разработването и използването на всяка софтуерна система е свързано със създаването на множество документи, които са с различно предназначение, структура и съдържание [6]. Най-общо тези документи могат да бъдат класифицирани в две групи:

- а) документи, свързани с процеса на създаване на софтуер — планове, графики, отчети, стандарти, протоколи от заседания, разменяни съобщения и др.

Тази документация зависи от стила на управление на проектите и се регламентира от действащи в конкретната софтуерна организация вътрешни правила и стандарти. Те определят каква е структурата на всеки документ, кой и кога го създава, какви са процедурите за утвърждаване, използване и променяне, къде и колко дълго се съхранява.

- б) документи, описващи създадения програмен продукт**

В зависимост от предназначението си тази документация може да бъде съпровождаща или експлоатационна.

Предназначенето на *съпровождащата* документация е да осигури информацията, необходима за разучаване и за внасяне на изменения в програмите. Тя може да включва:

- описание на функционалните и нефункционални изисквания към ПП;
- описание на архитектурата на софтуерната система — от какви основни компоненти се състои и какви са връзките между тях;
- за всяка компонента — описание на спецификациите и на детайлния проект;
- първичен текст на програмата. В някои организации има стандарт за т. нар. вътрешно документиране на програмите чрез коментари.
- описание на програмата, което включва предназначение, описание на логиката, структура, компоненти, използвани методи и алгоритми, входна и изходна информация, процедури за извикване и зареждане и др.

Предназначенето на *експлоатационната* документация е да осигури информацията, необходима за използването на софтуерната система от крайни потребители или администратори, които отговарят за функционирането на системата.

Документацията за администратора се състои от:

- инструкция за инсталациране. Тя съдържа указания, как да се инсталира системата в конкретна потребителска среда и какви са техническите изисквания към хардуерната конфигурация.
- ръководство за администратора. То описва връзките с други системи, обяснява извежданите съобщения и възможни действия при получаването им, определя същността и начина на извършване на някои сервизни функции, като архивиране, възстановяване на системата след срив, преконфигуриране и др.

От изключително значение е и документацията за крайния потребител. Съществуват различни стандарти за съдържанието на потребителската документация — например стандарт ANSI/IEEE Std 1063-1987. Потребителската документация може да се състои от две части — обучаваща и справочна. Обучаващата част е предназначена за начинаещи потребители, а справочната — за потребители с по-голям опит, които се нуждаят от информация за възможностите на системата. Справочната част обикновено представя основните функции, подредени по определен начин за бързо търсене — например по азбучен ред.

Стилът на изложение зависи от някои характеристики на потенциалните потребители, като образование, квалификация, ниво на компютърна грамотност и др. Ако в изискванията към системата има и описани характеристики на потребителя, то те трябва да се вземат предвид.

Могат да бъдат формулирани следните изисквания към потребителската документация:

- а) *Правилност*. Трябва да има пълно съответствие между функционирането на програмната система и описането ѝ в документацията. След всяка нова версия трябва да се обновява и документацията, като се проверяват и изложените примери, така че потребителят да не бъде насочван погрешно.
- б) *Пълнота и структурираност*. Да има подробно съдържание и дори каква последователност на четене да се следва от потребителите с различна квалификация и опит.
- в) *Стилът на изложение* да е ясен, недвусмислен и съобразен с терминологията в съответната област на приложение. Да се избягват тривиалните обяснения и примери.
- г) Ръководството за потребителя да бъде относително *затворена система*, съдържаща необходимата и достатъчна информация. Да се избягват сложните препратки

дори с цената на повтаряне на информация. Да няма препратки към труднодостъпни източници.

Всяка организация трябва да определи свои вътрешни стандарти за начина на създаване на документацията и за самата документация — от кои документи ще се състои и как ще се идентифицират те, каква да е структурата и съдържанието на всеки документ, какви правила при оформянето на документите ще се спазват (обща структура на документите, начин на оформление на страниците, номерация, използвани шрифтове, цветове и др.), каква ще е процедурата за внасяне на изменения.

В България съществуват стандарти от серията БДС ЕСГЮ. Те имат номера БДС 19.XXX-УУ, където XXX е номерът на стандарта, а УУ — годината на създаването му. Тези стандарти са от периода 1978—1980 година и са морално оstarели.

Съществуват автоматизирани средства, подпомагащи съставянето, оформянето и отпечатването на различните видове документи. Освен намаляване на времето и усилията за документиране преимущество на използването на такива средства е възможността за поддържане на пълно съответствие между последното състояние на софтуерната система и всички описващи я документи.

Литература

1. Манева, Н. Основни тенденции при разработването на средства за тестване на програми. АИТ и АС, №3, 1987, с. 77—84.
2. Sneed, H., K. Kirchoff. PRUFFSTAND — A Testbed for Systematic Software Components. Proc. INFOTECH State of the Art Conf. on Software Testing, 1978, London.
- 3.. Thomas, P., Practical Software Maintenance. NY Wiley, 1997.
4. James, M., Software maintenance. NY Prentice Hall, 1983.
5. Belady, L., M. Lehman. An Introduction to Growth Dynamics. In W. Freinberger (ed.) Statistical Computer Performance Evaluation, Academic Press, 1972.
6. Guidelines for the documentation of software in industrial computer systems (publ. by IEEE) 1987.

6. КАЧЕСТВО НА СОФТУЕРА

6.1. Общи понятия

Интуитивно е ясно, че всеки потребител желае да придобива и използва качествен софтуер. От това следва, че и стремежът на всеки разработчик е да създава софтуер с високо качество. Стимулът за това е не само чисто етичен, но има очевидни икономически причини. Възможността даден програмен продукт да бъде продаден, и то на по-висока цена, зависи силно от неговото качество. Теоретиците на софтуерните технологии и производство са разбирали важността на качеството на софтуера и в повечето определения на предмета на дисциплината, явно или не толкова явно, са включвали тази характеристика.

В контекста на този учебник и в съответствие с крайните цели на дисциплината разглеждането на всяко понятие, в дадения случай качеството на софтуера, трябва да води по възможност до конструктивни решения. Щом принципният стремеж на производителите на софтуер е да създават софтуер с високо качество, естествена цел е да се изгради някаква система или поне съвкупност от взаимообвързани мерки за *осигуряване на качеството*. За да стане обаче това, необходимо е преди това да се намерят достатъчно точни, обективни и ефективни методи за *измерване на качеството* на програмните продукти. Последното пък изисква най-напред да се изясни какво е това качество на софтуера.

Напоследък повечето автори са склонни да дават по-обща дефиниция на понятието качество, водени вероятно от убеждението, че всеки опит за по-подробно определение рано или късно ще доведе до намирането на контрапримери и следователно до опровергаване. Причините за това се крият в прекалената универсалност на понятието качество (дори ако се абстрагираме от философския му аспект), изключително широката му приложимост и уязвимост от практиката. Така че напоследък като че ли най-добре и от най-широк кръг се възприема определението, дадено в *International Standard Quality Vocabulary (ISO 8402-1986)*, още повече че то има в известен смисъл статут на стандарт:

Качеството е съвкупността от средства и характеристики на даден продукт или услуга, носители на способността му да отговори на явно или неявно указанi нужди.

Понякога изразът „указани нужди“ се заменя с привидно по-конкретното „нужди на потребителя“, но дори един не особено задълбочен анализ показва,

че това едва ли е стъпка към по-голяма яснота и конструктивност. Причината е в значителната размитост на понятието потребител и още по-голямата неопределеност на неговите нужди.

Така даденото определение може да се приложи и към качеството на софтуера и да послужи за добра основа при изграждането на ясен и конкретен възглед преди всичко с оглед на прагматичната задача за измерването му. Методологично погледнато,

естествената стъпка, водеща до тази цел, е създаването на модел на качеството на софтуера.

6.2. Модели на качеството на софтуера

6.2.1. Модел на Боем

Смята се, че първи сериозни изследвания по въпроса е направил Боем [1] през 1973 година, а по-късно, през 1978 година, ги е задълбочил с помощта на други автори [2]. Моделът на качеството на софтуера на Боем има йерархичен характер, но е сравнително слабо структуриран със своите реални две нива. Качеството на софтуера Боем свързва преди всичко с неговата *полезност* и възможност за лесно *съпровождане*. Първият аспект се определя от няколко *характеристики* — *надеждност*, *ефективност* и *използваемост* от гледна точка на потребителя човек. Вторият аспект зависи от други характеристики — *тестируемост*, *разбираемост* и *модифицируемост*. Освен това има още една, несвързана с двата аспекта характеристика, наречена *портабелност* (*мобилност*). Характеристиките от своя страна зависят от свойства на по-долно ниво. Тази зависимост вече не е чиста йерархия, защото не само че дадена характеристика се определя от две или повече свойства от по-долното ниво, но има свойства, които определят повече от една характеристика. Например надеждността се определя от три свойства — пълнота, точност и непротиворечивост, но от своя страна непротиворечивостта определя и характеристиката разбираемост. По-нататък идеята е да се оцени всяко свойство за конкретния програмен продукт чрез никаква обективна мярка (не непременно число, а по-скоро една от малък брой възможни степени в рамките на скала). Тази мярка се нарича *метрика*. Още тогава Боем е осъзнал, че по някакъв начин трябва да се отразява важността на метриката. Защото е ясно, че дадено свойство или характеристика е много важно за даден тип програмни продукти и не толкова — за други. Типичен пример е надеждността — за софтуер, управляващ полети на самолети или ракети, надеждността е една от най-важните характеристики, докато за един текстов редактор това едва ли е така. Още един жалон в работата на Боем е разбирането за необходимостта от автоматизирано оценяване на свойствата и характеристиките.

Естествено, като всяка пионерска работа и тази има своите недостатъци — не съвсем ясната структурираност, недостатъчната пълнота на множеството от характеристики, съсредоточаване почти изключително върху качеството на програмния код, а не върху цялостния програмен продукт, сравнително тясната експериментална база (програми на Фортран). Независимо от това следващите, по-съвършени модели несъмнено са били силно повлияни от идеите на Боем. Най-малкото развита е идеята за йерархичност, а и повечето от характеристиките, предложени от Боем, отново са включени, макар и понякога на различни нива.

6.2.2. Типичен йерархичен модел

Един завършен йерархичен модел на качеството на софтуера е предложен в [3]. Този модел е създаден с активното участие на самолетостроителната компания „Боинг“. Интересът на „Боинг“ към такъв тип разработки е леснообясним. Проектът, чийто резултат е моделът, е имал три основни задачи:

- развиване и утвърждаване на резултатите от предходни проекти на подобна тематика;

- разширяване на рамката на разглеждане на програмния продукт като такъв;
- разработване на методология за определяне и специфициране на изисквания към факторите, определящи качеството на софтуера.

Като основа на нашите разглеждания ще ни послужи друг йерархичен модел [4]. Той наподобява модела на „Боинг“, почива на същите принципи, но има и специфики, които идват само да покажат, че в една такава сложна и динамична област, каквато е създаването на софтуер, трудно се достига единствена гледна точка. Определящата причина да изберем [4] е участието на български специалисти в разработката на този модел и по-голямата му популярност в България.

Структура

Качеството се разглежда като йерархична структура. То се намира на най-високото ниво — **0**.

На следващото ниво — **1** — се намират *факторите*. *Факторът се определя като потребителски ориентирано свойство, представящо даден аспект на качеството на софтуера от гледище на потребителя*.

В зависимост от конкретния модел факторите могат да бъдат от 6 до 16. В разглеждания от нас те са 6 и са следните:

- гъвкавост,
- коректност,
- надеждност,
- съпровождаемост,
- удобство на използване,
- ефекти юст.

Ето как се дефинират горните фактори на качеството на програмния продукт.

Гъвкавост: Лекота на адаптиране към нови функционални условия, включително при изменение на областта на приложение или други условия на функциониране.

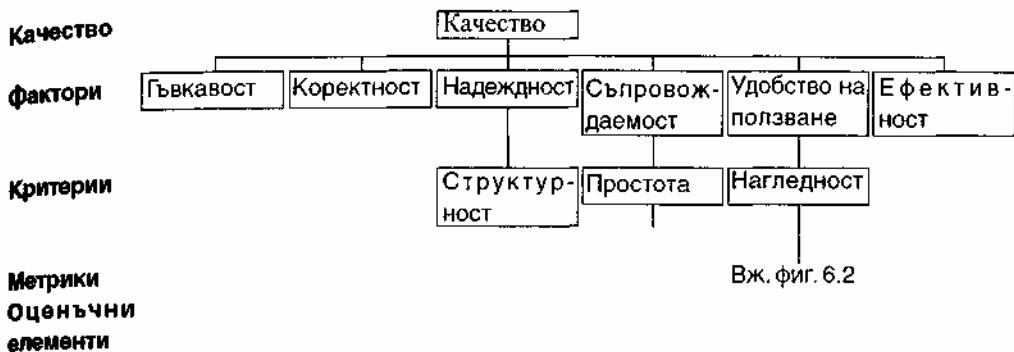
Коректност: Степен на съответствие на специфицираните алгоритми и други изисквания спрямо обработката на данни, както и спрямо потребителската документация.

Надеждност: Способност на програмния продукт да изпълнява зададените функции, предвидени в програмната документация, при отклонения, възникващи в средата на функциониране (апаратни и програмни отклонения и грешки).

Съпровождаемост: Възможност за отстраняване на отклонения и грешки **процеса** на експлоатация на програмния продукт и за поддържането му в актуално състояние.

Удобство на използване: Възможност за усвояване и експлоатация на програмния продукт с минимални усилия.

Ефективност: Пълнота и скорост на изпълнение на специфицираните функции в зададената изчислителна среда.



Фиг. 6.1 Качество и определящите го фактори

Вж. фиг. 6.2

На ниво 2 са *критериите*. *Критериите са софтуерно ориентирани свойства, представящи характеристики на програмния продукт.* Всеки **фактор** се определя от няколко критерия. Така например факторът *съпровождаемост* се определя от следните критерии:

- **структурност**: степен на сложност на построяването на частите на програмата в единно цяло в съответствие с принципите на структурното проектиране и програмиране (впрочем тук следва да припомним, че разглежданият **модел** е разработван в средата на 80-те години, когато обектната ориентираност беше далече от сегашната си популярност и престиж);
- **простота**: простота на построяването на програмите;
- **нагледност**: нагледност на представянето на текстовете на програмите, възможност за визуално или звуково изобразяване на хода на изпълнението.

На ниво 3 се позиционират *метриките*. В юерархичните модели *метри-ките са софтуерно ориентирани детайли на даден критерий*. Ще отбележим, че в някои от другите юерархични модели това ниво отсъства. Където ги има, метриките се определят от оценъчните елементи.

Ако продължим илюстрирането на юерархичната структура, можем да вземем **например** поддървото на критерия *нагледност*, определен от следните *метрики*.

- коментари към логиката на програмата
- оформление на текста на програмата
- възприета система за идентификация

Последното ниво — 4 — е това на *оценъчните елементи*. *Оценъчният елемент е елементарна характеристика на най-ниско ниво, която подлежи на количествено оценяване.* От метриките, определящи критерия *нагледност*, за пример нека вземем коментари към логиката на програмата. Тя се определя от следните *оценъчни елементи*:

- коментари към машиннонезависимите фрагменти на програмата
- коментари към машиннозависимите фрагменти на програмата
- коментари към входно-изходните точки



Фиг. 6.2. Поддърво на критерия нагледност

Определяне стойностите на оценъчните елементи

В рамките на йерархичните модели в процедурата за намиране на количествена оценка на качеството, разгледана по-нататък, директно се оценяват само оценъчните елементи. Методите, по които става това, могат да се класифицират по два признака:

А. По *начина на получаване* на информацията за програмния продукт:

- измерителен
- регистрационен
- органолептичен
- изчислителен

Б. По *източника на получаване* на информацията:

- традиционен
- експертен
- социологически

Ще проследим последователно дадените класификации.

Измерителният метод се състои в използването на програмни инструментални средства за определяне обема на програмата (например броя на редовете първичен код, броя на редовете коментари, броя на операторите и на опе-рандите, броя на изпълнимите оператори, броя на разклоненията в програмата, броя на входните и на изходните точки и др.), на времето на изпълнението на цялата програма или определени нейни клонове, на времето на реакция на програмата на определени входове, както и на други показатели.

Регистрационният метод се основава на информация, получена по време на изпитания или функциониране на програмния продукт, когато се регистрират определени събития, например време и брой на грешки, време на начало и край на работата.

Органолептичният метод е основан на използването на информация, получавана от човека в резултат на анализ на възприятията му чрез сетивните органи (зрение, слух) и се прилага при определяне на такива показатели, като удобство на използване, коректност и други подобни.

Изчислителният метод използва теоретични и емпирични зависимости (обикновено на ранни стадии на разработването), статистически данни, натрупвани при изпитанията, експлоатацията и съпровождането на програмния продукт. С помощта на изчислителния метод се определя точността на пресмятанятията, времето на реакция, необходимите ресурси и др.

В определени случаи може да се използва и комбинация от няколко метода.

Що се отнася до видовете източници на получаване на информацията, те се класифицират така:

— Стойностите на оценъчните елементи по *традиционнния* метод се определят в специализирани организации от звена по изпитания и изчисления. Такива са лаборатории, полигоны, центрове за изпитание на програмни продукти, стендове и др., както и отдели по програмиране, изчислителни центрове, служби за контрол на качеството и т. н.

— Определянето на стойностите на оценъчните елементи чрез *експертния* метод се осъществява от група експерти, компетентни в решаването на дадения тип задачи, на основа на техния опит и интуиция. Експертният метод се използва в случаите, когато оценката не може да бъде извършена чрез друг метод или другите методи са значително по-трудоемки. Препоръчва се експертен метод да се прилага при оценяване на елементи, свързани с нагледността, удобството на използване, пригодността на документацията, структурността, широтата на обхват на функциите.

— *Социологическият* метод се основава на обработката на специално подгответи анкети въпросници, на информация от изложби и конференции и т. н.

Стойностите на оценъчните елементи могат да попадат в следните видове *скали*:

— *интервална скала*, характеризираща се с относителни или абсолютни величини в даден интервал;

— *порядкова скала*, позволяваща ранжиране на стойностите на някои оценъчни елементи чрез сравняването им с определени опорни стойности;

— *номинална скала*, показваща само наличието на конкретното разглеждано свойство в дадения програмен продукт.

Процедура по оценяване

Целта на модела е конструктивна — чрез използване на вече дефинираната структура и като се следва точно определена процедура, да се определи за всеки даден програмен продукт (принадлежащ на определен тип софтуер) число (обикновено дефинирано в интервала [0,1]), което да характеризира качеството. Основната идея на процедурата за оценяване на конкретен програмен продукт се състои в следното (за нашите цели правим известно опростяване, което обаче не нарушава принципите на оценяването):

1. Приема се, че всяка характеристика на всяко ниво може да приема стойности в интервала $[0,1]$. Стойностите на всички *оценъчни елементи* се определят от експерти по един от изброените по-горе методи. При това експертите ползват някои най-общи указания, изгответи предварително. Когато се отнася за определяне на стойност чрез *измерителен, регистрационен или изчислителен* метод, се посочва точният начин за получаване на стойността (например чрез формула) и как така получената стойност, ако не е в интервала $[0,1]$, да се трансформира подходящо. В останалите случаи се дават по-общи указания, например от следния вид: коментарите към входно-изходните точки (един от оценъчните елементи) се оценяват с:

- 0, ако изцяло липсват;
- 0.33, ако ги има, но са твърде кратки и неясни;
- 0.67, ако ги има и са задоволителни;
- 1, ако са отлични (ясни, налични към всяка точка, с голяма обяснителна сила).

Стойностите на всички характеристики от по-горни нива се изчисляват като претеглени суми от стойностите на определящите ги характеристики от по-долното ниво.

2. Приема се също така, че на всяка характеристика на всяко ниво съответства тегло в интервала [0,1]. При това, както е обично, сумата от теглата на характеристиките, отнасящи се до една характеристика от по-горно ниво, е 1. Например, можем да предположим, че за разгледаното по-горе конкретно поддърво са определени такива тегла на ниво оценъчни елементи, отнасящи се до метриката *коментари към логиката на програмата*:

- 0.4 за коментари към машиннонезависимите фрагменти на програмата;
- 0.3 за коментари към машиннозависимите фрагменти на програмата;
- 0.3 за коментари към входно-изходните точки.

Всички теглови стойности, да ги наречем базови, се определят предварително от експерти и се отнасят до точно определен тип софтуерни продукти. Причината е, че дадена характеристика може да е изключително важна за даден тип софтуер и тогава тя трябва да получи високо тегло; същата характеристика може да не е особено съществена за качеството на друг тип софтуерни продукти и тогава базовите експерти ще й дадат относително по-ниско тегло. Впрочем по-горе дадохме такъв пример, отнасящ се до надеждността на различни типове софтуер.

3. Следователно при започване на оценката на качеството на конкретен програмен продукт от даден тип експертите разполагат с вече готовите:

- процедура за оценяване;
- указания за намиране стойностите на оценъчните елементи;
- теглата към всички характеристики на всички нива точно за *дадения тип* софтуер.

4. Експертите определят стойностите на всички оценъчни елементи, като използват структурата на модела и указанията и изследват оценявания програмен продукт.

5. Нека получените стойности на оценъчните елементи, определящи метриката M са e_1, e_2, \dots, e_n , а съответните им предварително зададени тегла са w_1, w_2, \dots, w_n . Тогава стойността на метриката M се изчислява по формулата:

$$M = e_1 * w_1 + e_2 * w_2 + \dots + e_n * w_n$$

6. Например, ако разгледаме вече познатото поддърво и за него експертите са определили следните оценки:

- 0.9 за коментари към машиннонезависимите фрагменти на програмата;
- 0.6 за коментари към машиннозависимите фрагменти на програмата;
- 0.8 за коментари към входно-изходните точки, то ще се получи:

$M = 0.9 * 0.4 + 0.6 * 0.3 + 0.8 * 0.3 = 0.36 + 0.18 + 0.24 = 0.78$ (теглата са определени по-горе в т. 2).

7. Същата схема на пресмятане се прилага за всяко от следващите нива.

7.1. След като всички стойност на метрики M са ни известни, за всеки критерий C прилагаме формулата:

$C = M_1 * w_1 + M_2 * w_2 + \dots + M_n * w_n$, където M_i са метриките, определящи критерия C .

7.2. Аналогично, след като всички стойност на критерии C са ни известни, за да получим стойността на всеки фактор F прилагаме формулата:

$$F = C_1 * w_1 + C_2 * w_2 + \dots + C_n * w_n,$$

Тук C_i са критериите, които определят фактора F .

7.3. И накрая, след като всички стойности на фактори F са ни известни, за да получим стойността на *качеството* Q прилагаме формулата:

$$Q = F_1 * w_1 + F_2 * w_2 + \dots + F_n * w_n$$

където F_i са факторите, които определят качеството Q .

Ще отбележим, че педантичната нотация би изисквала три индекса за точното експлициране принадлежността на една метрика (коя е поред в рамките на множеството, определящо критерия, кой поред е критерият и кой — факторът). Спестяваме ги за по-голяма прегледност и по-лесно четене. Несъмнено всеки би могъл да разбере какво означава обозначението M_{jk}^i .

По този начин получаваме числото Q . То остава в интервала $[0,1]$. Причината за това е, че всички стойности на тегла и на оценъчни елементи са в интервала $[0,1]$. Ясно е, че програмен продукт с $Q = 1$ е с максимално високо качество. В реалността стойност 1 не би трябвало да се достига, защото тя има смисъла на идеал за високо качество. Благодарение на нормализацията става възможно сравняването на получените стойности за Q на различни програмни про-Дукти, както и получаването на самостоятелна представа за качеството на отделен програмен продукт.

Оценка на модела и метода

Така описаният модел на качеството и свързаната с него процедура за оценка има следните *предимства*:

1. *Простота* на модела от структурна гледна точка. Йерархията е лесно обозрима структура, с ясно видими еднопосочни връзки. При сегашните разбириания за качеството на софтуера тази структура е адекватно негово отражение.
2. Структурата на модела съдържа в себе си и директно следващи възможности за *конструктивност*, т. е. за изграждане на процедура за оценяване. Видяхме как е направено това.
3. Твърде елементарно е да се създаде компютърна програма за пресмятане на оценката, доколкото действията са прости и еднообразни. Нещо повече, без такава *автоматизация* оценяването става твърде неефективно.
4. Крайният резултат е *едно-единствено число* в зададен интервал и това допринася за избягване на двусмислия и неясноти относно качеството на оценяване на програмен продукт.

Естествено, разглежданият йерархичен модел има и *недостатъци*.

1. Процедурата за оценяване съдържа твърде много елементи на *субективност*:

— указанията за това, как да се определят стойностите на оценъчните елементи, както се видя по-горе, се изготвят предварително от експерти;

— при самото определяне на стойностите на част от оценъчните елементи експертите, оценявачи конкретния програмен продукт, също не могат да не проявят субективност; оказва се, че твърде малка част от тези оценъчни елементи могат да бъдат оценени обективно (чрез измервания и изчисления); повечето получават стойности на основата на експертна оценка (ако искаме да сме съвсем точни, ще посочим, че от общо 257 оценъчни елемента в разглеждания модел 216, т. е. около 84% се оценяват на чисто експертна основа [5]);

— теглата на всички нива за всеки тип програмни продукти се определят предварително от експерти, следователно — субективно.

Естествено, съществуват методи, които позволяват от няколко субективни експертни оценки да се получи една — много по-малко субективна. Това не премахва изцяло проблема, но във всички случаи, ако се прилага, осъществява цялостната процедура.

2. Огромна *трудоемкост* поради посочената вече необходимост от многогранна и разнообразна работа на експертите по предварително определяне на тегла, на указания за оценяване, както и на отделен труд по определяне стойностите на оценъчните елементи за всеки конкретен програмен продукт.

3. Донякъде недостатък е и това, че макар моделът да е топологично универсален (т. е. грубо казано, йерархичната структура е винаги една и съща), всеки тип програмни продукти изисква свое *собствено множество от тегла*.

6.2.3. Класификационен модел

Основна идея

В [6] и [7] е предложен един друг модел на качеството на програмните продукти. Той се опитва да преодолее някои от недостатъците на йерархичните модели, преди всичко субективността в значителна степен, както и трудоемкостта. Естествено, тъй като всяко подобрение има своята цена, в този случай това е известна загуба на точност. Последната обаче, както ще се види, невина-ги е жизненоважна.

Основната идея на този метод произлиза от разбирането, че при оценката на качеството на даден програмен продукт не е толкова важно да се знае едно точно число, отразяващо това качество, а по-скоро да се придобие яснота за позицията на оценявания програмен продукт спрямо няколко известни еталона с добре познато качество. Следователно, ако сме в състояние да кажем за даден програмен продукт A, че той е приблизително толкова добър по качество, колкото добре известните и *отлични* по качество X и Y, както и че A не е в категорията на добре известните и *средни* по качество M и N, нито пък в категорията на също така добре известните и *лоши* по качество P и Q, то това би било напълно удовлетворително в повечето от случаите, в които се поставя проблемът за качеството на програмния продукт A. От това разбиране следва самият модел. Той се основава на т. нар. класификационни методи.

Математическа основа на модела

Нека да разгледаме даден тип програмни продукти (текстови редактори, програми за изчисляване на заплати и т. н.). За този тип определяме множество от характеристики: j_1, j_2, \dots, j_n . На всяка характеристика може да се присвоява стойност 1, ако конкретният програмен продукт притежава тази характеристика, 0 — ако не я притежава, и x — ако няма информация за това дали я притежава.

Нека да предположим още, че няколко продукта от този тип са добре известни и че стойността на всяка характеристика за всеки от тези продукти може да бъде определена. Тези продукти ще наричаме *еталони*. Интуитивно ясно е, че е желателно броят на x-овете да се намали, т. е. да работим с максимално пълна информация. По-нататък на основата на опита на експерти и потребители, еталоните се разбиват на няколко *класа* в зависимост от своето качество. На практика обикновено се определят два класа (добър и лош) или три (отличен, добър, лош). Няма обаче никакви принципни трудности, ако класовете са повече. По този начин всеки еталон е представен чрез вектор $E_i = (a_{i1}, a_{i2}, \dots, a_{in})$, където n е броят на характеристиките и a_{ik} може да взема стойности от множеството $\{0, 1, x\}$. Ако s е броят на класовете, то E принадлежи на точно един клас K_g , където $g=1, 2, \dots, s$.

Следователно сечението на всеки два класа е празно. Така дефинираните данни могат да се представят в таблица T_{mn} , обикновено наричана *обучаваща*.

Клас	Продукт	J_1	J_2	...	J_n
	E_1	a_{11}	a_{12}	...	a_{1n}
K_1
	E_{m1}	a_{m1}	a_{m2}	...	a_{mn}
	"				
	"				
	$E_{m_{s-1}+1}$	$a_{m_{s-1}+1,1}$	$a_{m_{s-1}+1,2}$...	$a_{m_{s-1}+1,n}$
K_s
	E_m	a_{m1}	a_{m2}	...	a_{mn}

Табл. 6.1. Обучаваща таблица

Създадена веднъж, таблицата е относително постоянна в рамките на да-Ден тип програмни продукти. Нека сега предположим, че е даден *нов* програмен продукт E . Определяме съответния му вектор $E = (a_1, a_2, \dots, a_n)$, т. е. определяме стойността на всяка от характеристиките на E . Основната идея на предлагания метод е чрез използване на обучаващата таблица T и на описание на E , зададено чрез горния вектор, новият програмен продукт E да бъде класифициран, т. е. отнесен към един от s -те предварително дефинирани класове. Това би ни дало информация за неговото качество, която в много случаи е напълно достатъчна.

В основата на метода стои търсенето на подмножества от признаки, които отразяват различията между класовете.

Тест на таблицата е такова подмножество от стълбове, че всеки два реда на подтаблицата, образувана от тези стълбове, които принадлежат на различни класове, се различават в поне един от стълбовете. **Неприводим** се нарича тест никое собствено подмножество на който не е тест. Ще го означаваме с **НТ**. Като пример нека разгледаме следната таблица, в която продукти 1 и 2 принадлежат на клас 1, 3, 4 и 5 — на клас 2, и останалите — 6, 7 и 8 — на клас 3:

Продукт/признак	1	2	3	4	5	6	7	8	9	10	11	12	13
E_1	1	1	1	1	1	1	0	1	1	1	1	0	1
E_2	1	1	0	1	-	1	0	1	1	0	0	0	0
E_3	1	1	1	1	0	1	1	0	0	1	0	1	0
E_4	1	1	1	1	0	1	1	1	1	1	1	1	0
E_5	0	1	0	0	1	1	1	-	0	0	1	-	0
E_6	0	0	1	0	0	0	0	0	0	-	0	-	0
E_7	0	1	0	1	0	1	0	1	0	-	-	-	0
E_8	0	0	0	1	0	0	0	1	0	-	-	-	0

Табл. 6.2. Примерна обучаваща таблица

Вижда се, че например стълбовете 7 и 9 образуват **тест**. Комбинацията (0,1) се среща само в първия клас, (0,0) — само в третия, а останалите две комбинации — (1,0) и (1,1) — само във втория.

Представителен набор за даден клас по някакво подмножество от стълбове е такава част от описание, която в подтаблицата, образувана от този стълбове, се среща в дадения клас, но не се среща в останалите класове. **Неприводим** се нарича представителен набор, никоя част от който не е представителен набор. Ще го означаваме с **НПН**. Ако в горната таблица разгледаме стълбовете 1,3 и 5, ще забележим, че описанието (0,0,0) е характерно само за третия клас (то се среща в E_7 и E_8 и в никой друг клас).

Забележете, че **тествовете са общи за цялата таблица** и различават всички класове едновременно, докато **представителните набори отличават само даден клас** от всички останали. Представителните набори са толкова по-добри, колкото кратността им е по-голяма (брой на появяванията им за дадения клас). Двете понятия — **НТ** и **НПН** — се характеризират с дължина — броя на признаките, които участват в тях.

Алгоритми

Създадени са няколко алгоритъма, които по различен начин, използвайки било **НТ**, било **НПН**, се опитват да класифицират даден обект (продукт) към някой от класовете на таблицата. Казваме „опитват”, защото има случаи, когато алгоритъмът не е в състояние да завърши успешно класифицирането. Това е една от причините да се използват няколко алгоритъма едновременно. Когато се получи резултатът от всеки от тях, се прилага някаква обобщаваща процедура, която дава окончателния резултат. Най-просто е да се извърши гласуване

(всеки алгоритъм има един глас) и класът, получил най-много гласове (т. е. този, към който най-много алгоритми са причислили класификацията продукт) да бъде обявен за клас на дадения продукт. Тук ще дадем един пример за такъв алгоритъм. Прилага се следната формула:

$$y_i(E) = \frac{1}{m_i - m_{i-1}} \sum_{(j_1, j_2, \dots, j_k) \in T} \sum_{i=m_{i-1}+1}^{m_i} a_{j_1}^{a_{j_1}} \dots a_{j_k}^{a_{j_k}}$$

Тук T е множеството на всички **НТ**, j_1, j_2, \dots, j_k образуват **НТ**

Резултатът $y_i(E)$ въщност показва броя гласове, които в рамките на този алгоритъм идват от класа 1. Ако се върнем към горната таблица пример и решим да класифицираме продукта

$$E = (1, 0, 0, 0, 1, 1, 0, 1, 1, 0, -, 0, 0),$$

ще видим, че се получава следният резултат:

$$y_1(E) = 1/2 (2_{(7,9)} + \dots) = 3 \quad y_2(E) = 1/3 (0_{(7,9)} + \dots) = 0 \quad y_3(E) = 1/3 (0_{(7,9)} + \dots) = 0$$

Поясняваме, че в изчислението за първия клас от други **НТ** идват още 4 гласа, поради което крайният резултат там е 3. Този резултат означава, че разглежданият алгоритъм класифицира изследвания продукт E към първия клас.

Пример

Един от първите реални примери, чрез които е проверяван и валидиран разглежданият класификационен метод, оценява програмни продукти за труд и работна заплата. Избрани са 9 такива продукта, разпределени в 3 класа — много добри, добри, слаби. Това разпределение е направено от експерти. Определени са и 22 характеристики и всеки от продуктите е описан като вектор по тези характеристики. По-нататък се

оказва, че 9 от признаките не са информативни, **зашто** имат еднакви стойности за всичките 9 продукта. Така остават следните 13 признака:

1. Минимални входни данни
2. Просто кодиране на входните данни
3. Степен на автоматичност
4. Приложимост при изменящи се условия
5. Оптималност на организацията на данните
6. Рационалност на интерфейса
7. Бързина на изпълнение
8. Независимост от операционната система
9. Лекота на експлоатация
10. Структурираност
11. Възможности за интерфейс с други продукти
12. Рационалност на потребителския език
13. Защита на данните

Описанията образуват дадената вече като пример таблица. При експериментите последователно всеки от продуктите е изваждан от таблицата и разпознаван чрез разработените 8 класификационни алгоритъма. Резултатите са твърде окуражителни. Подобни експерименти са правени и със значителен брой други типове програмни продукти — авторски системи, текстови редактори и др.

Литература

1. Boehm B. W. Software and its Impact: A Quantitative Assessment. Datamation, Vol. 19, No 5 May 1973, p. 49—59.
2. Boehm B.W. et al. Characteristics of Software Quality. North Holland. 1987.
3. Bowen T.P., G.B.Wigle, J.T.Tsai. Specification of Software Quality Attributes — Software Quality Evaluation Guidebook, RADC-TR-85-37, Vol.III, 1985.
4. Общая методика оценки качества программных средств. Межправительственная комиссия по сотрудничеству социалистических стран в области вычислительной техники. Бюллетень, выпуск 1(37), Москва, 1988.
5. Eskenazi A. The Problem of Objectivity in Software Quality Evaluation. Proceedings of the 19th Intl.Conf. with Summer School „Information Technologies and Programming”, Sofia 1994 p. 6—14.
6. Eskenasi A. Evaluation of Software Quality by Means of Classification Methods. J. of Systems and Software, vol.10, No 3, October 1989, p. 213—216.
7. Ангелова, В. Оценка на качеството на програмни продукти чрез класификационни методи. Кандидатска дисертация. София, 1987.

7. СОФТУЕРНИ МЕТРИКИ

7.1. Въведение

В тази глава ще обясним същността на софтуерните метрики и ще предложим класификация по няколко критерия. Ще опишем шест „класически“ метрики, измерващи характеристики на различни типове обекти — продукти, процеси и ресурси. Накрая ще разгледаме някои методологични проблеми на изграждане на система от метрики в конкретна софтуерна организация.

7.2. Измерването в софтуерното производство

Първата публикация, свързана със софтуерни метрики, излиза през 1968 г. и поставя проблема за количествено измерване качеството на програми. Интересът към този проблем е неотслабващ, за което свидетелстват ежегодните международни конференции на тази тема, големият брой книги и публикации в специализираните издания. Основната идея на прилагането на софтуерни метрики е да се измерват характеристики на обекти в софтуерното производство (спецификации, проекти, програми, документация и др.). Така може да се получава информация за качеството на тези обекти и по възможност да се прогнозират някои тенденции в процеса на разработване на софтуера, за които се смята, че зависят от изследваните характеристики и могат да бъдат управлявани.

Ще въведем някои основни понятия, необходими за по-нататъшното изложение, придържайки се основно към възприетата в [1] терминология.

Измерването е процес, при който в съответствие с определени правила на характеристики на изследвания обект се съпоставят стойности.

Ако тези стойности са числови, измерването се нарича *количествено*.

Мярка е числото, съпоставено при количествено измерване.

Мярката представя различните състояния на характеристиката, която се измерва.

Понятието метрика има две значения в областта на софтуерните технологии и производство:

- a) Друго име за мярката;
- б) Процедура за намиране и използване на мярката.

Измерването и получаваните мерки трябва да притежават следните свойства:

Обективност — получаваните мерки да не зависят от субекта, извършващ измерването.

Надежност — мярката да има необходимата различаваща способност и при повтаряне на измерванията при еднакви условия да се получават еднакви резултати.

Валидност — получаваните мерки да отразяват реално свойствата на измервания обект.

В [2] са формулирани следните три предпоставки за прилагането на дадена софтуерна метрика:

а) можем точно да измерваме някакво свойство на обект в софтуерното производство;

б) съществува връзка между това, което можем да измерим, и това, което бихме искали да знаем за съответното свойство;

в) тази връзка е осъзната, потвърдена (т. е. доказана е теоретично и експериментално) и може да бъде изразена чрез формула или в термините на съответен модел.

Следователно основните проблеми при създаване на софтуерни метрики са два:

— да се идентифицират съществени за изследване характеристики на софтуерни обекти и те да се дефинират в изчислими термини;

— да се определят гипът на използваната скала, допустимите операции върху метриките, статистическият смисъл на оценяването и ограничаването на грешките при измерване.

7.3. Класификация на софтуерните метрики

Изключително голямото разнообразие на съществуващите софтуерни метрики затруднява реалното им използване. Затова ще предложим класификация на метриките по няколко критерия.

А. Класификация в зависимост от *предназначението*

Най-често софтуерните метрики се използват за:

а) оценяване на разходите и усилията за разработване на софтуера;

б) измерване на производителността на разработчиците;

в) моделиране на качеството на софтуера и оценяването му;

г) измерване и прогнозиране на надежността на създаваните софтуерни системи;

д) изследване на някои програмни свойства.

Б. Класификация в зависимост от *целта* на прилагане на метриката Софтуерните метрики могат да бъдат:

а) за характеризиране (to characterize) — да разберем изследвания обект и да установим база за сравнение при бъдещи оценки;

б) за оценяване (to assess) — регистриране на текущо състояние на изследвания обект;

в) за прогнозиране (to predict) — предварителна оценка за бъдещо състояние на изследвания обект;

г) за подобряване (to improve) — натрупване на количествена информация за определяне на факторите, от които зависи качеството на продукта или ефективността на процеса.

В. Класификация в зависимост от *типа на изследвания обект* В софтуерното производство могат да се разграничават три основни типа обекти:

- а) процеси — дейности, свързани със създаването и използването на софтуера;
- б) продукти — резултати от процесите;
- в) ресурси — елементи, входни данни за процесите.

Всичко, което можем да измерваме, е характеристика на обект от горните три класа. Измерваните характеристики могат да бъдат вътрешни (дефинирани в термините на самия обект) или външни (дефинирани и измервани в зависимост от отношението на обекта към средата му).

Примери за обекти в софтуерното производство и съответните им вътрешни и външни характеристики са описани в таблица 1, структурата и някои данни на която са взети от [1].

Г. Класификация в зависимост от *начина на получаване на информацията*, въз основа на която се определят стойностите на измерваните характеристики:

- а) регистрационен метод — получаване на информацията по време на разработване или функциониране на ПП чрез регистриране на отделни събития — продължителност на сеанс, брой аварийни ситуации и описането им в документацията и др.;
- б) измерителен метод — информацията се получава чрез прилагане на специално разработени инструментални средства;
- в) възприятиен (органолептичен) метод — информация, получена след анализ на зрителни или слухови възприятия — използвани цветове, бързина на смяна на текстове, звукова сигнализация и др.
- г) изчислителен метод — използване на теоретични или емпирични зависимости, изведени въз основа на статистическите данни, натрупани през различни фази на жизнения цикъл на ПП.

Първи са се появили метриките, изследващи програми. Обяснението е, Че програмите са най-важният продукт, създаван във всеки софтуерен проект. Сравнително лесно дори за неспециалист в областта на софтуерните Метрики е идентифицирането на програмни свойства, които да бъдат изследвани с определена цел. За този вид метрики е възможна следната класификация:

- а) в зависимост от начина на изследване — статични и динамични.

При статичните метрики се анализира само текстът на програмата, докато за динамичните метрики е необходимо изпълнението ѝ.

б) в зависимост от изследваната програмна характеристика — сложност (структурна, текстуална или алгоритмична), модулност, тестируемост, независимост от средата, модифицируемост и др.

в) в зависимост от нивото на декомпозиране — изследване на отделна програмна част или на програмна система като цяло. Метриките от втората група са много по-сложни, защото се изследва потокът на данните и потокът на управление в многокомпонентни системи.

ОБЕКТИ	ВЪТРЕШНИ ХАРАКТЕРИСТИКИ	ВЪНШНИ ХАРАКТЕРИСТИКИ
ПРОДУКТИ		
Спецификации	Размер Модулност Коректност Функционалност Възможност за повторно използване	Изчерпателност Съпровождаемост Разбираемост.
Проекти	Размер Модулност Свързаност	Качество Сложност Съпровождаемост
Програми	Размер Алг. сложност Тестируемост	Надежност Приложимост Съпровождаемост
Тестови данни	Размер Ниво на тестване	Качество Изчерпателност
Документация	Обем Пълнота	Разбираемост Адекватност
ПРОЦЕСИ		
Специфициране	Продължителност Трудоемкост Брой изменения в изискванията	Качество Разходи Стабилност
Детайлно проектиране	Продължителност Трудоемкост Брой открити грешки в спецификациите	Разходи
Тестване	Продължителност Трудоемкост Брой открити грешки в спецификациите	Разходи Изчерпателност Систематичност
РЕСУРСИ		
Кадри	Възраст Възнаграждение	Производителност Професионален опит
Екип	Численост Структура	Продуктивност Качество
Софтуер	Цена Размер	Полезност Надеждност
Хардуер	Цена Изч. възможности	Надежност на функциониране

Таблица 1

Една метрика се нарича *адитивна*, ако резултатите от прилагането ѝ за програмна система могат да се получат чрез сумиране на резултатите от прилагането ѝ върху съставящите я програмни части.

7.4. Примери за софтуерни метрики

Ще опишем някои софтуерни метрики, които се смятат за „класически“. Те се споменават най-често в специализираната литература и са с висок степен на валидност, т. е. полезността от прилагането им е потвърдена теоретически и/или експериментално. Описанието е кратко и цели само да се добие представа за съответната метрика, а в цитираните източници могат да се намерят останалите подробности.

Метриките ще представим по следната схема:

I. Описание

То включва вида на метриката по някои от критериите за класификация, препратка към литература за допълнителна информация, описание на самата метрика и оценъчните елементи, които използва, възможности за автоматизация на процедурата за измерване и др.

II. Приложимост — за какво могат да се използват получаваните от метриката резултати.

III. Метрика за размера на програма

I. Описание

Метриката е статична и може да се прилага както за отделен програмен модул, така и за програмна система.

За мярка на размера на програма се избира броят програмни редове в изходния текст на програмата.

$L = L_1 + L_2$ където

L — общ брой редове;

L_1 — коментарни и празни редове;

L_2 — същински програмни редове.

Метриката е адитивна.

II. Приложимост

1. Броят на грешките, усилията за разбиране, поддържане и съпровождане са правопропорционални на размера на програмите [3, 4].

2. В зависимост от размера (на ниво програмна единица) и на размера и броя модули в програмна система програмите могат да се класифицират като

прости, средно сложни, сложни и свръхсложни и за всяка категория да се формулират мерките за осигуряване на качеството, да се оценят обхватът и сложността на тестването, съпровождането и др.

Примерна класификация е дадена в [3, 4]. Границите стойности могат да се определят експериментално или да се задават за всеки софтуерен проект. З. L е най-проста мярка за извършената от програмиста работа.

M2. Метрика на Маккейб за структурна (цикломатична) сложност

I. Описание

Метриката на Маккейб [5] е за програми (обекти от тип „продукт“). Тя е статична и измерва сложността на потока на управление в програмата. Тази метрика е една от най-често цитираните в литературата.

За прилагане на метриката трябва да се построи управляващият граф на програмата, в който върховете са отделните оператори (или линейни участъци) и два върха са свързани с ребро, ако има предаване на управление между съответните оператори

Тогава

$$V(G) = e - n + 2 * p, \text{ където:}$$

$V(G)$ е цикломатичната сложност, представляваща максималния брой линейно-независими пътища в програмата;

e — брой ребра в управляващия граф;

n — брой върхове в управляващия граф;

p — брой на свързаните компоненти в графа.

Разработени са инструментални средства за автоматично изчисляване на цикломатичната сложност, като при това се установяват и някои структурни неправилности — недостижими програмни части, пресичащи се цикли и други.

Недостатъци на метриката са, че не отчита дълбината на линейните участъци и нивото на влагане на управляващите структури. Не се отразяват и между-модулните връзки.

Разширение на тази метрика е метриката на Гонг—Шмид [6], която предлага цикломатичната сложност да се изчислява по формулата

$$C(G) = V(G) + E,$$

където E представя обобщената степен на вграждане на управляващите структури една в друга.

II. Приложимост

Метриката предлага мярка за вътрешномодулната сложност, която определя леснотата на разбиране и усвояване на програмите и точността на внасяне на изменения в тях.

Препоръчва се да се проектират модули с цикломатична сложност $V < 10$ като се смята, че модулите със сложност до 10 са прости, до 20 — средно сложни, от 30 до 50 — сложни, а при сложност над 50 трябва да се обмисли преструктуринг (разделяне на няколко модула).

M3. Метрика на Холстед за текстуална сложност

I. Описание

Статична метрика за програма, написана на произволен език за програмиране. Предложена е от Холстед през 1976 г. [7] и е метриката, за която има наи-много публикации.

Програмата се разглежда като състояща се от активни елементи (оператори) и пасивни елементи (операнди). Операндите са константи или променливи, използвани при реализацията на алгоритъма в конкретна програма. Операторите са елементи, влияещи върху стойностите или наредбата на операндите.

За прилагане на метриката се определят:

n_1 — брой различни оператори;

n_2 — брой различни операнди;

N_1 — общ брой оператори;

N_2 — общ брой операнди.

Чрез тези оценъчни елементи се изчисляват:

Речник: $n = n_1 + n_2$

Дължина на програмата: $N = N_1 + N_2$

Изчислена дължина на програмата:

$N = n_1 * \log_2 n_1 + n_2 \log_2 n_2$

Обем на програмата: $V = (N_1 + N_2) * \log_2 (n_1 + n_2)$

Трудност на програмата: $D = (n_1 / 2) * (N_2 / n_2)$

Ниво на програмата: $L = 1 / D$

Усилия за създаване на програмата: $E = V * D$

Преброяването на n_1 , n_2 , N_1 и N_2 , както и изчисляването на метриките за гореописаните свойства, може да се извърши от инструментално средство (специализиран лексически анализатор).

II. Приложимост

1. Могат да се оценяват усилията за създаване на програмата. В [8] се предлага за оптимална сложност на програма стойността $E_{op} = 60\ 800$ и за максимална $E_{max} = 129\ 600$.

Сравняването на E за конкретна програма с тези две стойности дава възможност да се класифицират усилията за разработването ѝ като малки, средни или големи.

2. Може да се оцени времето за програмиране чрез използване на индекса на Страуд за интензивност на интелектуалните занимания, който за програмирането има стойност $S = 18$. Така

$$T = E/S = E/18$$

3. Гордън доказва [9], че усилията за създаване на нова програма са сравними с усилията за разбиране на съществуваща. Този факт често се използва за предварителна оценка на усилията за съпровождане в случаите, когато то не се осъществява от разработчиците.

4. В [8] се предлагат формули за предварително оценяване на броя на откриваните грешки:

$B_1 = V / 3000$ — брой грешки, откривани при изпитанията; $B_2 = 4 * B_1 = V / 750$ — общ брой грешки, открити през жизнения цикъл на програмата.

5. В [10] се обосновава възможността усилията за тестване да се оценяват чрез произведението $NX * E$, където NX е индекс за ниво на влагане (индикатор за необходимия брой тестове), а E е мярката на Холстед за усилията на разработване.

Някои от предлаганите мерки не са очевидни и за разбирането им е необходимо вникване в създадената от Холстед теория, но от прагматична гледна Точка е важно, че тази метрика е с изключително високо ниво на експериментално валидиране. Ясно дефинираният смисъл на четирите оценъчни елемента, сравнително простата и лесна за автоматизиране процедура и множеството допълнително пресмятани мерки правят от метриката на Холстед задължителен елемент на всяка използвана в практиката метрична система.

M4. Метрика на Рехенберг за технологична сложност

I. Описание

Метриката на Рехенберг [11] е опит за преодоляване на едностраничността на съществуващите метрики за сложност. Тя е статична, комплексна метрика която може да се прилага за програми или програмни системи, написани на произволен език за програмиране от високо ниво.

Рехенберг предлага следната формула:

$$CC = SC + EC + DC, \text{ където:}$$

CC е комплексната сложност;

SC — операторка сложност;

EC — сложност на използваните изрази;

DC — сложност на данните.

Въведени са и относителни мерки за сложност. Ако NS е броят на операторите в програмата, то съответните относителни мерки са:

$$RCC = CC / NS$$

$$RSC = SC / NS$$

$$REC = EC / NS$$

$$RDC = DC / NS$$

Относителните мерки дават възможност да се сравняват програми с различна дължина.

Операторната сложност SC на програмата е сума от операторните сложности на съставящите я оператори. За всеки тип оператор е въведена мярка за сложността му. Например операторът за присвояване има сложност 1, операторите за цикъл — 3, операторът goto — 5 и т. н. Влагането на операторите се отчита чрез коригиращия коефициент k = 1.5, с който се умножава мярката за сложност на всеки вложен оператор.

Сложността на изразите EC се пресмята чрез мерки за сложността на използваните в израза операции. Например операциите „+“ и „—“ и операциите за сравнение „>“, „=“, „<“ имат мярка за сложност 1. Операциите „*“ и „/“ имат мярка за сложност 2, логическите операции и/или имат стойност 3 и т. н.

Сложността на данни измерва разстоянието между декларирането и използването на данните. Мярката за сложност DC за програмата е сума от мерките за сложност на използваните променливи (локални и глобални).

II. Приложимост

По описаните в [11] примерни таблици за операторната сложност, сложността на изразите и сложността на данните лесно може да се настрои метриката така, че да може

да се използва за програми, написани на език от високо ниво, като се отразят и конкретните потребителски виждания за всяка от разглежданите мерки.

Мярката на Рехенберг може да се използва за прогнозиране на усилията за тестване и съпровождане чрез сравняване на мерките на новоразработвала софтуерна система със съществуваща, за която тези усилия са документирани.

M5. Метрика за четимост

I. Описание

Обект на изследване са различни видове документи — ръководства за потребителя, материали за обучение, спецификации и др. [12]. Предлаганата мярка е $F = 0.4 * (lm + plw)$,

където

lm е средна дължина на изречение, т. е. брои думи/брой изречения; plw — относителен дял на дългите думи, т. е.

$$plw = \frac{\text{брой дълги думи}}{\text{общ брой думи}} * 100$$

В българския език е прието дълги да се наричат думите с над 3 срички.

II. Приложимост

Мярката на четимост F отразява трудността на текста.

Боем [13] предлага мярката F да е

$10 < F < 12$ за обикновените делови документи и

$12 < F < 16$ за спецификации, документация и научни отчети.

Има и модификация на метриката за текстове на немски език, която да се използва за проверка на разбираемостта на софтуерната документация.

Съществуват много софтуерни метрики (библиографията в [1] е почти изчерпателна) и затова интерес представлява изследването на връзките между тях. Изключително важен е резултатът от сравняване на някои метрики за програми. Оказва се, че могат да прилагат три основни метрики — за размер, метриката на Рехенберг за относителна сложност и метриката на Маккейб за структурна сложност, а много други мерки могат да се изведат от резултатите, получени само от тези метрики. Засега това е хипотеза след статистически експерименти, която си струва да бъде изследвана и с формални методи.

7.5. Методологични проблеми на използването на софтуерните метрики

Конструирането на нови метрики, доказване на валидността на съществуващи, създаване на модели и др. са обект на изследване от научните работници или преподаватели в тази област. Но все още прилагането на метриките в реални софтуерни проекти е ограничено.

Някои от основните причини за това са:

а) Участниците в софтуерните разработки нямат достатъчно добра теоретическа подготовка, за да оценят полезността на софтуерните метрики. Някои метрики (особено тези за процеси) изискват създаването на сложни модели и съответна организационна структура, осъзнаването и реализацията на които може да продължи с години.

б) Немислимо е използването на метрики в реалните сложни софтуерни проекти, без да са налице инструментални средства, автоматизиращи процедурите на измерване. Съществуващите немного такива инструментални средства обикновено са свързани с точно определена хардуерна и операционна среда.

M2. Метрика на Маккейб за структурна (цикломатична) сложност

I. Описание

Метриката на Маккейб [5] е за програми (обекти от тип „продукт“). Тя е статична и измерва сложността на потока на управление в програмата. Тази метрика е една от най-често цитираните в литературата.

За прилагане на метриката трябва да се построи управляващият граф на програмата, в който върховете са отделните оператори (или линейни участъци) и два върха са свързани с ребро, ако има предаване на управление между съответните оператори

Тогава

$$V(G) = e - n + 2 * p, \text{ където:}$$

$V(G)$ е цикломатичната сложност, представляваща максималния брой линейно-независими пътища в програмата;

e — брой ребра в управляващия граф;

n — брой върхове в управляващия граф;

p — брой на свързаните компоненти в графа.

Разработени са инструментални средства за автоматично изчисляване на цикломатичната сложност, като при това се установяват и някои структурни неправилности — недостижими програмни части, пресичащи се цикли и други.

Недостатъци на метриката са, че не отчита дължината на линейните участъци и нивото на влагане на управляващите структури. Не се отразяват и между-модулните връзки.

Разширение на тази метрика е метриката на Гонг—Шмид [6], която предлага цикломатичната сложност да се изчислява по формулата

$$C(G) = V(G) + E,$$

където E представя обобщената степен на вграждане на управляващите структури една в друга.

II. Приложимост

Метриката предлага мярка за вътрешномодулната сложност, която определя леснотата на разбиране и усвояване на програмите и точността на внасяне на изменения в тях.

Препоръчва се да се проектират модули с цикломатична сложност $V < 10$ като се смята, че модулите със сложност до 10 са прости, до 20 — средно сложни, от 30 до 50 — сложни, а при сложност над 50 трябва да се обмисли преструктурiranе (разделяне на няколко модула).

М3. Метрика на Холстед за текстуална сложност

I. Описание

Статична метрика за програма, написана на произволен език за програмиране. Предложена е от Холстед през 1976 г. [7] и е метриката, за която има наи-много публикации.

Програмата се разглежда като състояща се от активни елементи (оператори) и пасивни елементи (operandи). Operandите са константи или променливи, използвани при реализацията на алгоритъма в конкретна програма. Операторите са елементи, влияещи върху стойностите или наредбата на operandите.

За прилагане на метриката се определят:

n_1 — брой различни оператори;

n_2 — брой различни operandи;

N_1 — общ брой оператори;

N_2 — общ брой operandи.

Чрез тези оценъчни елементи се изчисляват:

Речник: $n = n_1 + n_2$

Дължина на програмата: $N = N_1 + N_2$

Изчислена дължина на програмата:

$N = n_1 * \log_2 n_1 + n_2 \log_2 n_2$

Обем на програмата: $V = (N_1 + N_2) * \log_2 (n_1 + n_2)$

Трудност на програмата: $D = (n_1 / 2) * (N_2 / n_2)$

Ниво на програмата: $L = 1 / D$

Усилия за създаване на програмата: $E = V * D$

Преброяването на n_1 , n_2 , N_1 и N_2 , както и изчисляването на метриките за гореописаните свойства, може да се извършва от инструментално средство (специализиран лексически анализатор).

II. Приложимост

1. Могат да се оценяват усилията за създаване на програмата. В [8] се предлага за оптимална сложност на програма стойността $E_{op} = 60\ 800$ и за максимална $E_{max} = 129\ 600$.

Сравняването на E за конкретна програма с тези две стойности дава възможност да се класифицират усилията за разработването й като малки, средни или големи.

2. Може да се оцени времето за програмиране чрез използване на индекса на Стraud за интензивност на интелектуалните занимания, който за програмирането има стойност $S = 18$. Така

$$T = E/S = E/18$$

3. Гордън доказва [9], че усилията за създаване на нова програма са сравними с усилията за разбиране на съществуваща. Този факт често се използва за предварителна оценка на усилията за съпровождане в случаите, когато то не се осъществява от разработчиците.

4. В [8] се предлагат формули за предварително оценяване на броя на откриваните грешки:

$B_1 = V / 3000$ — брой грешки, откривани при изпитанията; $B_2 = 4 * B_1 = V / 750$ — общ брой грешки, открити през жизнения цикъл на програмата.

5. В [10] се обосновава възможността усилията за тестване да се оценяват чрез произведението $NX * E$, където NX е индекс за ниво на влагане (индикатор за необходимия брой тестове), а E е мярката на Холстед за усилията на разработване.

Някои от предлаганите мерки не са очевидни и за разбирането им е необходимо вникване в създадената от Холстед теория, но от прагматична гледна Точка е важно, че тази метрика е с изключително високо ниво на експериментално валидиране. Ясно дефинирианият смисъл на четирите оценъчни елемента, сравнително простата и лесна за автоматизиране процедура и множеството допълнително пресмятани мерки правят от метриката на Холстед задължителен елемент на всяка използвана в практиката метрична система.

M4. Метрика на Рехенберг за технологична сложност

I. Описание

Метриката на Рехенберг [11] е опит за преодоляване на едностраничността на съществуващите метрики за сложност. Тя е статична, комплексна метрика която може да се прилага за програми или програмни системи, написани на произволен език за програмиране от високо ниво.

Рехенберг предлага следната формула:

$CC = SC + EC + DC$, където:

CC е комплексната сложност;

SC — операторка сложност;

EC — сложност на използваните изрази;

DC — сложност на данните.

Въведени са и относителни мерки за сложност. Ако NS е броят на операторите в програмата, то съответните относителни мерки са:

$$RCC = CC / NS$$

$$RSC = SC / NS$$

$$REC = EC / NS$$

$$RDC = DC / NS$$

Относителните мерки дават възможност да се сравняват програми с различна дължина.

Операторната сложност SC на програмата е сума от операторните сложности на съставящите я оператори. За всеки тип оператор е въведена мярка за сложността му. Например операторът за присвояване има сложност 1, операторите за цикъл — 3,

операторът `goto` — 5 и т. н. Влагането на операторите се отчита чрез коригирання коефициент $k = 1.5$, с който се умножава мярката за сложност на всеки вложен оператор.

Сложността на изразите EC се пресмята чрез мерки за сложността на използванието в израза операции. Например операциите „+“ и „—“ и операциите за сравнение „>“, „=“, „<“ имат мярка за сложност 1. Операциите „*“ и „/“ имат мярка за сложност 2, логическите операции и/или имат стойност 3 и т. н.

Сложността на данни измерва разстоянието между декларирането и използването на данните. Мярката за сложност DC за програмата е сума от мерките за сложност на използванието променливи (локални и глобални).

II. Приложимост

По описаните в [11] примерни таблици за операторната сложност, сложността на изразите и сложността на данните лесно може да се настрои метриката така, че да може да се използва за програми, написани на език от високо ниво, като се отразят и конкретните потребителски виждания за всяка от разглежданите мерки.

Мярката на Рехенберг може да се използва за прогнозиране на усилията за тестване и съпровождане чрез сравняване на мерките на новоразработана софтуерна система със съществуваща, за която тези усилия са документирани.

M5. Метрика за четимост

I. Описание

Обект на изследване са различни видове документи — ръководства за потребителя, материали за обучение, спецификации и др. [12]. Предлаганата мярка е $F = 0.4 * (l_m + pl_w)$,

където

l_m е средна дължина на изречение, т. е. брои думи/брой изречения; pl_w — относителен дял на дългите думи, т. е.

брой дълги думи $pl_w = \text{-----} * 100$

общ брой думи В българския език е прието дълги да се наричат думите с над 3 срички.

II. Приложимост

Мярката на четимост F отразява трудността на текста.

Боем [13] предлага мярката F да е

$10 < F < 12$ за обикновените делови документи и

$12 < F < 16$ за спецификации, документация и научни отчети.

Има и модификация на метриката за текстове на немски език, която да се използва за проверка на разбираемостта на софтуерната документация.

Съществуват много софтуерни метрики (библиографията в [1] е почти изчерпателна) и затова интерес представлява изследването на връзките между тях. Изключително важен е резултатът от сравняване на някои метрики за програми. Оказва се, че могат да прилагат три основни метрики — за размер, метриката на Рехенберг за относителна сложност и метриката на Маккейб за структурна сложност, а много други мерки могат да се изведат от резултатите, получени само от тези метрики. Засега това е хипотеза след статистически експерименти, която си струва да бъде изследвана и с формални методи.

7.5. Методологични проблеми на използването на софтуерните метрики

Конструирането на нови метрики, доказване на валидността на съществуващи, създаване на модели и др. са обект на изследване от научните работници или преподаватели в тази област. Но все още прилагането на метриките в реални софтуерни проекти е ограничено.

Някои от основните причини за това са:

а) Участниците в софтуерните разработки нямат достатъчно добра теоретическа подготовка, за да оценят полезността на софтуерните метрики. Някои метрики (особено тези за процеси) изискват създаването на сложни модели и съответна организационна структура, осъзнаването и реализацията на които може да продължи с години.

б) Немислимо е използването на метрики в реалните сложни софтуерни проекти, без да са налице инструментални средства, автоматизиращи процедурите на измерване. Съществуващите немного такива инструментални средства обикновено са свързани с точно определена хардуерна и операционна среда.

Метриките, измерващи програми, зависят силно и от езика, на който са написани програмите, което прави невъзможно създаването на универсални средства. От прагматична гледна точка интерес представлява следният въпрос: Как една софтуерна организация, осъзнала полезността на софтуерните метрики, може да изгради и внедри подходяща за нея система от метрики?

На основата на изследванията в областта на софтуерните метрики и обобщавайки [1], може да се предложи следната постъпкова процедура:

1. Формулирайте целта, която искате да постигнете с въвеждането на система от метрики (вж. част 3. от класификация А).
2. Изберете свързваните с тази цел софтуерни обекти и съответните им характеристики.
3. Проучете (или възложете проучването на специалист) какви софтуерни метрики съществуват за избраните характеристики.
4. Съставете съвкупност от тези метрики, за които може да се осигурят (чрез закупуване или разработване) инструментални средства, автоматизиращи процедурите на измерване.
5. Регламентирайте и осъществете система за натрупване и анализ на данните от прилагане на метриките и определете контролен срок за експеримента.
6. След изтичане на срока въз основа на експертна оценка на резултатите от прилагане на метриките преустановете експеримента или съставете нова съвкупност от метрики чрез отпадане на някои, добавяне на нови или модифициране на съществуващи. Във втория случай се върнете към стъпка 4.

Литература

1. Fenton N. Software Metrics — A Rigorous Approach. Chapman & Hall, London, 1991
2. Kitcheman B. Software metrics in Software Reliability Handbook (ed. by RRook), London, Elsevier, 1990.
3. Йенсен, Р, Ч.Тониз. Технология на програмирането. Техника, София, 1987.
4. Липаев В. Качество программного обеспечения. М., Финанси и статистика, 1983.
5. McCabe T. A complexity measure. IEEE Transactions on Software Engineering, SE-2, 1976, pp. 308—320.
6. Gong, H., M.Schmidt. A complexity Measure Based on Selection and Nesting. ACM SIGMETRICS, Nov. 1983.
7. Halstead M.H. Elements of Software Science. New York, Elsevier, 1977.
8. Hocker, H. et al. Comparative Descriptions of Software Quality Measures. GMD-Studien Nr.81,1984.
9. Gordon R.D. Measuring Improvements in Program Clarity. IEEE Trans, on SE, SE-5(2), 1979, pp. 79—90.
10. Budde et al. Untersuchungen ueber Massnahmen zur Verbesserrung der Software Production, Teil 1, Bericht Nr. 130 der Gesellschaft fuer Mathematik und Datenverarbeitung. Munchen-Wien, Oldenbourg Verlag, 1980.
11. Rechenberg P. Ein neues Mass fuer die softwaretechnische Komplexitaet von Programme¹¹-Informatik Forsch. Entw. Nr. 1,1986, pp. 26— 37.
12. Gunning R. Technique of Clear Writing. New York, Me Graw Hill, 1968.
13. Boehm et al. Characteristics of Software Quality. TRW Series of Software Technology, voU. North Holland, 1978.

8. УПРАВЛЕНИЕ НА КАЧЕСТВОТО

8.1. Проблемът за управление на качеството

В глава 6. бяха разгледани проблеми на *качеството на софтуерния продукт*. Производството на софтуер, както видяхме в глава 2. и специално в 2.3.8., е резултат на *процес*. Този процес има свои характеристики, които могат да имат различна степен на съвършенство. Следователно може да се говори за *качество и на софтуерния процес*. Близко до ума е, че при по-добро качество на софтуерния процес по принцип трябва да се очаква и по-добро качество на софтуерния продукт. Например, ако процесът на планиране и създаване на тестовите данни за определен програмен продукт се извърши грижливо и компетентно, т. е. с високо качество, може да се очаква, че продуктът ще има по-висока стойност на фактора надеждност.

Следователно *осигуряването на качеството на софтуера (OKC)* не може да не се свърже с планирането и прилагането на подходящи мерки през целия производствен цикъл. Това е изразено много точно в [1] по следния начин:

„Качеството е едновременно философия и съкупност от ръководни принципи, които са основата на една *непрекъснато подобряваща се организация*, интегрираща:

фундаментални техники за управление,

постоянни усилия за усъвършенстване,

технически средства, всичко това в рамките на един дисциплиниран и целенасочен метод.“

Следствие от тази синтезирана мисъл впрочем е, че качеството не може да бъде единствено задача, специално възложена на дадено лице или дори група, а Трябва да се разглежда и като приоритетна отговорност на всеки, участващ в Разработването на софтуерния продукт. Следователно методите и конкретните мерки на OKC трябва да са такива, че да мотивират, ангажират и задължават всеки да работи за качеството на продукта.

OKC е тясно свързано с жизнения цикъл на програмния продукт, по-точно с всяка негова фаза. Планирането му следва да се извърши още в началните фази и да обхваща всички следващи фази. Всеки междинен продукт, резултат от Дадена фаза, заедно със своите характеристики следва да бъде точно дефиниран. На тази основа следва да се предвидят начини за обективно установяване доколко удовлетворително е реализиран този междинен продукт (за крайния продукт казаното се подразбира). За да се осъществи тази дейност по един систематичен и ефективен начин, теорията препоръчва да се изработи *програма за осигуряване на качеството на софтуера (ПОКС)*.

8.2. Компоненти на програмата за осигуряване на качеството

8.2.1. фактори

Няколко фактора оказват важно влияние върху ПОКС [7]. Без да се впускаме в подробности, ще ги изброим:

- изисквания към *графика*;
- разполагаемия *бюджет*;
- технологичната *сложност* на продукта;
- предполагаемия *размер* на продукта;
- относителния *опит* на разработчиците;
- хардуерните и софтуерните *ресурси*, предвидени за процеса на разработване;
- изискванията на *договора* за възлагане.

В момента на създаването на ПОКС трябва да се анализира и установи доколко всеки от горните фактори (а евентуално и някои други) ще й окаже влияние.

8.2.2. Прегледи

Под преглед (review) се разбира дисциплинирана групова дейност, насочена към изследване на продукт или процес. Ефективното изпълнение на прегледа изисква умело съставяне на групата с оглед комбиниране различните необходими квалификации. Отличават се 3 вида прегледи:

А. *Пробег (walkthrough)*. Това е неформален преглед на софтуерния продукт. Обикновено не се подчинява на строги правила. Прилага се най-често върху първичния код на междинните продукти.

Б. *Инспекция (Inspection)*. Това е дисциплиниран формален преглед на всякачъв тип продукти.

В. *Проверка на конфигурацията (Configuration Audit)*. Много често крайното приемане ча софтуерния продукт се основава на редица проверки на конфигурацията. Целта им е да установят доколко крайният продукт удовлетворява изискванията, формулирани първоначално. Тези проверки се делят на 2 категории:

В.1. *Функционални*. Основната цел на функционалните проверки е да преодолеят ефекта от многобройните тествания и съответни корекции. Напълно е възможно при установяване на дадена грешка в процеса на разработване и последващото нейно отстраняване да е била внесена друга грешка. Колкото по-дълго се разработва даден продукт, толкова повече опасността от такива „вторични“, неотстранени грешки нараства. Крайната функционална проверка цели да покаже пълната липса на такива остатъчни грешки.

В.2. *Физически*. Тази категория проверки се съсредоточава върху съответствието на продукта с изискванията на договора по отношение на документацията и сроковете. Те се правят след функционалните.

8.2.3. Оценяване

Оценка се прави обикновено от един специалист. Целта ѝ е да се установи съответствие на всички характеристики на всеки продукт с формулираните изисквания. По същество това е функция по контрол на качеството на продукта. Тя обаче осигурява информация и за процеса на разработване. Поради тези причини всички дейности по оценяване трябва да се планират подробно и резултатите им да се използват възможно най-пълноценно. За отделните фази следва да се предвидят следните примерни дейности по оценяване.

1. *Изисквания към продукта.* Дори и в използвания модел на жизнения цикъл да няма точно такава фаза, не може на един първоначален етап да не се формулират тези изисквания, така че те да отразяват на едно първо, но достатъчно точно приближение възгледите на потребителя за продукта. При планирането следва да се предвиди преглед и оценка на:

- плана за разработване на продукта;
- софтуерните стандарти;
- плана за управление на софтуерната конфигурация;
- плана за осигуряване на качеството;
- спецификацията на изискванията към продукта;
- спецификацията на изискванията към интерфейса.

2. *Общо проектиране.* Както известно, на този етап изискванията се конкретизират и уточняват, а така също потребителският възглед започва да се третира от гледна точка на реализацията. Съответни на това са и оценките на:

- всички ревизирани производствени планове;
- плана за тестването на софтуера;
- ръководството за оператора;
- ръководството за потребителя;
- ръководството за диагностика,

като последните 3 визират всъщност плановете за тези документи, които очевидно в този ранен стадий не могат да бъдат направени по същество.

3. *Подробно проектиране.* Резултатите от този етап са подробни спецификации, на основата на които може да бъде извършено програмирането (кодирането) на продукта. Тук се предвижда оценяване на:

- текущо ревизираните планове — отново; документа — изход от подробното проектиране;
- документа — проект на интерфейса;
- документа — проект на базата данни;
- тестовите примери за проверка на отделните модули; — тестовите примери за проверка на интегрираните модули; — описание на процедурите по тестване;
- ръководството за програмиста;
- останалите ръководства в новото им състояние.

4. Програмиране и тестване на отделните модули. Както е известно, на този етап наред с програмирането самите изпълнители тестват своите реализации. През цялото време се извършват оценки на:

- плановете, доколкото търсят промени — както на всеки етап;
- написания първичен код;
- резултатите от тестването на отделните модули;
- всички ръководства.

5. Интегриране и тестване. Написаните и проверени отделни модули започват да се интегрират, което изисква оценяване на:

- текущо ревизираните планове;
- резултатите от тестването на интегрирането;
- актуализирания първичен код;
- описанието на приемните тестове;
- всички ръководства.

6. Тестване на крайния продукт. Тук се оценяват:

- текущо ревизираните планове;
- всички ръководства;
- актуализираният първичен код;
- документът за описание и управление на версията;
- докладът за тестване на крайния продукт (системата).

8.2.4. Типове оценки

Добре е известно, че софтуерът за военни цели изисква особена надеждност и други високи качества. Поради тези причини при създаването на такъв софтуер се полагат особено сериозни мерки за осигуряване на качеството, като специално се набляга на оценяването. Във връзка с това Министерството на от branата на САЩ има приет стандарт — *DOD-STD-2168 Software Quality Program*. В него се фиксираят *типовете оценки*, които следва да се правят при производството на всеки програмен продукт. Поважните от тях следват:

- съблюдаване на изисквания формат и стандарти за документацията;
- съответствие с изискванията на договора за разработка;
- вътрешна непротиворечивост;
- добра разбираемост;
- система за лесно намиране на пътя до всеки документ;
- съответствие на продукта с придружаващите го документи;
- коректно извършен анализ на изискванията, проектиране и програмиране (тук следва да се отбележи, че всъщност се прави оценка не на атрибути на продукта, а на качества на процеса);
- правилно разпределение на ресурсите — по време и памет;

- адекватно проведено тестване за съответствие на продукта с изискванията;
- адекватно съставени тестови примери;
- пълнота на тестването;
- пълнота на регресивното тестване (този проблем беше разгледан погоре в 8.2.2., т. В1 — тук се им предвид, че едни и същи тестове, проведени в начални стадии и довели до отстраняване на грешки, следва да се провеждат отново на по-късни етапи с цел избягване на грешки, резултат от корекции);
- съответствие и непротиворечивост между дефинициите на данните и тяхното използване.

8.2.5. Управление на конфигурацията

Управлението на софтуерната конфигурация обхваща методи и технологии за иницииране, оценяване и управление на измененията в софтуерния продукт след пускането му в експлоатация. Необходимите промени се правят не само в първичния код, но така също в документацията — вътрешна (съпровождаща) и потребителска, в отчетите за тестване, в отчетите за откритите грешки. Следят се и се документират последователно създаваните версии и движението им сред различните потребители.

Общо взето, при по-малки проекти, реализирани от малки фирми, не се отделят специални усилия управлението на конфигурацията да се върши по един системен, добре формализиран и документиран начин. Това впрочем е лесно обяснимо. Но ако един екип от 3 души е в състояние да помни историята на развитието на даден продукт наизуст или с малко неформални документи, то от известно място нататък (по отношение на мащаба на продукта и на колектива) такъв подход е невъзможен. Ето основните функции по управление на конфигурацията, които са решаващи за запазване качеството на продукта след влизането му в експлоатация:

- поддържане целостността на продукта;
- напълно контролирано управление на промените;
- управление и контрол на версийте;
- планиране на управлението на конфигурацията.

8.2.6. Отчитане на грешните

Добре известна истина е, че и в най-прецизно тествания софтуер след пускането му в експлоатация се откриват непрекъснато грешки. Все още сме много далече от възможността да се докаже математически строго липсата на грешки в дадена програма. За програми, писани на процедурни езици, има поне теоретическа яснота, как това би могло да стане, за обектно ориентираните — дори и това не е ясно.

Поради тази причина поддържането на ефективна система за регистриране на грешките с оглед последващото им отстраняване е абсолютна необходимост. Такава система трябва да е действена в следните насоки:

1. *Идентификация на грешките.* Всяка идентифицирана грешка трябва да се опише ясно и точно. Може да се опише както поведението на продукта в Дадената ситуация, така и реалният дефект в софтуера. Във всички случаи описанието трябва да бъде разбираемо както за хора, които не са се занимавали с Правенето на продукта, така и с оглед на това, че обикновено е минало вече Време от разработването и дори лица, които са участвали пряко в работата, Постепенно са загубили подробна представа за направеното. Впрочем така подготвената информация е необходима не само за конкретното отстраняване на хрешката, но и за по-системни анализи на типовете грешки.

2. *Анализ на грешките.* Трябва да се документира сериозността на грешката и трудността на нейното отстраняване. Това ще позволи правилно заделяне на необходимия ресурс, определяне на приоритет и график за корекция. Обикновено грешките се откриват сравнително лесно, често те просто се набиват на очи, но тъй като отстраняването им невинаги е лесно, налага се да се вземат управленски реше-

ния, свързани с приоритета им. Впрочем отделянето във времето на анализа от корекцията обуславя разглеждането им като отделни и независими операции.

3. *Корекция на грешките.* Коригирането, освен че се извършва по същес тво, следва и да се документира. Описанието на корекцията трябва да съдържа:

- разказвателно описание на корекцията;
- списък на засегнатите модули от продукта;
- пълна идентификация според създадената система на засегнатите документи;
- евентуални промени в тестовите процедури в резултат на корекцията

4. *Въвеждане на корекцията в експлоатация.* На практика не е възможно всяка направена корекция да отива незабавно при потребителя. Затова обикновено се прави работен екземпляр, в който постепенно се натрупват направени корекции. В определен момент, резултат на управленско решение, се създава нова модификация на продукта, включваща всички направени корекции (а така също подобрения и нови функционалности, ако е имало такива) и тя се предлага официално на потребителите. Задължително се регистрира в коя модификация (или версия) всяка корекция е включена.

5. *Регресивно тестване.* Вече беше казано, че всяка корекция крие потенциална опасност от създаване на нови грешки. Изследвания показват, че тази опасност е около 20%. Най-разумната възможност за отстраняването им е в момента на откриването им те да бъдат третирани като стандартни грешки и да бъдат отстранявани по вече описаната схема. Във всички случаи обаче описанието на регресивния тест трябва да включва:

- списъка на отново тестваните компоненти;
- върху коя версия/модификация е направено тестването;
- индикация, дали тестването е било успешно или неуспешно.

6. *Категоризация на грешките.* Анализът на грешките би се улесnil, ако те се систематизират с оглед бъдещи анализи. Близко до ума е, че категоризирането ще е най-успешно, ако се направи в момента на отстраняването на всяка грешка. Възможни признания за категоризация са:

- тип на грешката — от изискванията, от проектирането, от програмирането, от тестването;
- приоритет на грешката — критична, некритична, козметична;
- честота на грешката — повтаряща се, неповтаряща се.

8.2.7. Анализ на тенденциите

Това е пасивна и по-второстепенна дейност, но тя може да насочи към полезни корективи. Състои се в анализиране на определени страни от софтуерния процес и изготвяне на съответни отчети.

1. *Количество на грешките.* Възможно е да се събират данни за количеството грешки както за целия период на разработването, така и за отделни крат-

ки периоди. При това е препоръчително грешките да се класифицират подходящо, например по функционални групи или по отговорни специалисти. При по-продължително събиране на подобни данни (в рамките на повече проекти) възможно е след статистическа обработка да се фиксираят някакви критични граници които биха указали например доколко е смислено да се обучават повече, или да се оставят на постовете им различните отговорници.

2. *Честота на грешките.* Честотата на грешките следва да се свързва с конкретна единица — тестова процедура, програмен модул, дял от спецификация. Също след статистически обработки могат да се правят изводи за латентни грешки в тези единици.

3. *Сложност на програмните модули.* Известни са немалко метрики, които позволяват обективно измерване на сложността на програмните единици. При установяване на по-голяма от някаква критична стойност, може да се пристъпи към опит за намаляване на сложността на тази единица.

4. *Честота на компилациите.* Въпреки че на пръв поглед тази характеристика изглежда тривиална, Де Марко е показал, че програмни модули, които са били компилирани често при създаването им, след това се компилират често и при интеграцията и системното тестване. Така че би било оправдано за програмна единица, за която е установено, че при кодирането е била компилирана много често, да се определи задължително процедура по оценяване.

8.2.8. Проследимост

Принципът за *проследимостта* означава, че за всяка единица, създадена по време на разработването на софтуера, трябва да може да се проследи от каква друга единица е получена. Така например за даден алгоритъм, въплътен в програма, следва да може да се види резултат на какво решение е по време на проектирането или на създаването на изискванията. Благодарение на това се улесняват оценките и преди всичко отстраняването на грешките.

8.2.9. Планиране на ПОКС

Както всяка друга дейност при производството на софтуер програмата за осигуряване на качеството е също обект на грижливо предварително планиране. Практическият съвет в това отношение е да се ползва аналогичен план от Предходна разработка. Най-малкото, с което може да бъде полезен подобен План, е, че няма да бъдат пропуснати задължителните елементи.

8.2.10. Социални фактори

Все по-често се обръща внимание на социалните елементи, начина на общуване на ръководителите с подчинените и с клиентите, мотивацията на разработчиците. Доколкото

осигуряването на качеството е особено силно свързано с общуването между членовете на екипа, често при противопоставящи роли с необходимо да се обърне внимание и на този елемент от софтуерния процес.

1. **Точност.** Абсолютната точност на представянето на данните от всяка оценка или друга дейност по осигуряването на качеството е задължителна. Практиката показва, че най-ефективната защита на контролираните срещу искания на контролиращите, например за извършване на промени, е позаване на неточност на данните. При това обикновено се цитират прецеденти и доколкото почти винаги времето за подробно разглеждане и доказване на представените данни не е достатъчно, такова позаване се оказва достатъчно. Единственият начин за противопоставяне е просто да не се създава нито един прецедент за неточност.

2. **Авторитет.** Обикновено отговорниците по осигуряване на качеството получават значителни пълномощия от ръководството на фирмата. Както и в други дейности обаче, знае се, че истинският авторитет е много по-малко резултати на административни мерки, отколкото на доказвана непрестанно лична компетентност.

3. **Пола.** По принцип ефективната работа по осигуряване на качеството води до обща полза за всички участници в процеса на разработване. От друга страна обаче, в краткосрочен и индивидуален план много често това не е така защото на даден програмист например му се налага да прави корекции, които смята за ненужни и които му отнемат допълнително време и усилия. Още по-характерен и направо типичен пример са исканията към отделните изпълнители, свързани е изготвяне на документация и спазване на стандарти. В този смисъл ръководителят и другите експерти по осигуряване на качеството имат за задача да мотивират засегнатите от мерките изпълнители, като ги убеждават в общата полезност на такива мерки.

4. **Комуникации.** Както вече се каза, осигуряването на качеството в голяма степен като схема е натрупване и разпространяване на информация. Формите за това са обикновено речеви и писмени в различни модификации. Откъдето следва, че отговорните за осигуряване на качеството трябва да са комуникативни личности, умеещи добре да се изразяват говоримо и писмено.

5. **Постоянство.** Не е възможно да се запази доверието, ако концепциите и произлизашите от тях решения се променят често. В такива случаи скоро идва момент, от който нататък нареджданията просто не се изпълняват в очакване на последваща промяна.

6. **Отмъстителност.** Лице, отговарящо за осигуряване на качеството, нерядко ще се сблъска с волни или неволни нарушения. Те биха му дали възможност при следващи случаи да се изкуши да се възползва от създалото се предимство пред нарушителя. Ясно е, че такава злонамереност е вредна за общата атмосфера и за ефективността на работата във фирмата.

8.3. Оценяване на софтуерните процеси

8.3.1. Методологията SEICMM

Изграждането на такава организация на процесите, която да осигурява ефективното и навременно създаване на качествен софтуер се нуждае и от непрекъснато обективно оценяване на достигнатото равнище. В края на 80-те години [2], [3] в Института по софтуерни технологии в Питсбърг (САЩ) (Software Engineering Institute (SEI), Carnegie

Melon University, Pittsburgh) се обявяват конструктивни изследвания в тази насока, които по-късно се развиват, прилагат в практиката и оказват силно влияние върху теоретиците и практиците в цял свят. Основният продукт на тези изследвания е моделът за оценяване на зрелостта — **Capability maturity model — CMM** [4]. Този модел е *предназначен за:*

- подобряване на софтуерните процеси;
- оценяване на софтуерните процеси, при което специално подгответи експерти определят текущото състояние на софтуерните процеси в организацията;
- оценяване от подгответи експерти на способността на потенциални изпълнители на даден софтуерен проект.

Оценяването се извършва по точно определена схема на основата на **150 въпроса**, отговорите на които са „да“ или „не“. Пример за такъв въпрос е: „Има ли във вашата фирма формално организирана система за осигуряване на качеството?“

Подобряването се извършва с помощта на точно и подробно определени и структурирани действия в планирането, технологията и управлението на разработването и съпровождането на софтуера.

CMM е получил *широко разпространение*, защото:

- основан е на *реалната практика*;
- отразява *най-добрите постижения* на тази практика;
- съобразява се с *нуждите* на участниците в софтуерния процес и неговото оценяване и подобряване;
- *документиран* е добре;
- *достърен* е за широката публика. *Структура*

CMM се състои от 5 *нива на зрелост (maturity levels)*. Нивото на зрелост е добре дефинирана развиваща се платформа, насочена към достижането на зрял софтуерен процес.

Всяко ниво на зрелост се състои от *ключови области на обработка (key process areas)*, с изключение на ниво 1. Всяка ключова област съдържа група свързани дейности, които, изпълнени съвместно, водят до постигане на целите, смятани за существени за това ниво на зрелост. Например една от ключовите области на обработка на ниво 2 е „Планиране на софтуерния проект“.

Всяка ключова област се състои от 5 секции *общи характеристики (common features)*: ангажираност за изпълнение, способност за изпълнение, изпълнявани дейности, измерване и анализ и верификация на приложението. Тези общи характеристики показват дали прилагането и формалното установяване на ключовата област на обработка е ефективно, повторяемо и трайно.

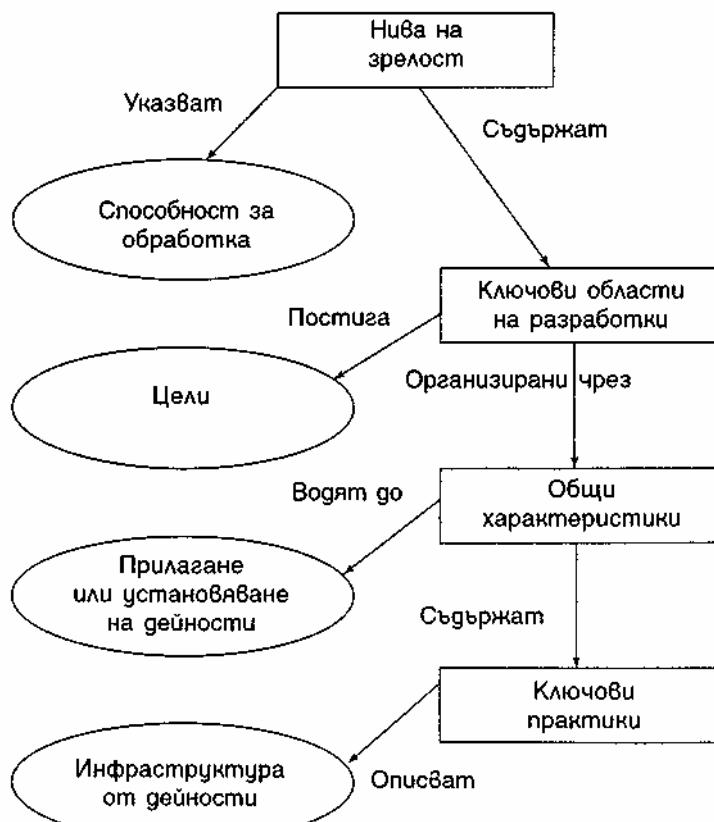
На следващото ниво са т. н. *ключови практики (key practices)*, чрез които се постигат целите на ключовите области. Ключовите практики описват инфраструктурата и *дейностите*, които допринасят за ефективното прилагане и Формалното установяване на съответната ключова област. Например една от ключовите практики на ключовата област „Планиране на софтуерния проект“ е: „Планът за разработването на софтуерния проект се създава в съответствие с документирана процедура.“

Схематично това е показано на фиг. 8.1.

Нива на зрелост

Ниво 1 се нарича *начално (initial)*. Организация на ниво 1 не осигурява стабилна среда за разработване и съпровождане на софтуер. В такива фирми управлението не е на здрава основа. Дори да има планирани процедури, в момента когато настъпи криза по отношение на срокове или ресурси, плановете се изоставят и се преминава изключително към програмиране (кодиране) и тестване. Успехът зависи само от опитността и квалификацията на ръководителя и членовете на екипа. Общата способност на организациите на ниво 1 е непредсказуема, каквото са и всички елементи на софтуерния процес — графици, бюджет, качество на продукта и др. Поради това не може да се предскаже и ефективността на такава организация.

Ниво 2 се нарича *повторяемо (repeatable)*. В организацията е установена политика за управление на софтуерния проект и процедури за прилагане на тази политика. Планирането и управлението на нови проекти се основава на опита от предишни проекти. Показател за достигането на това ниво е формалното установяване на ефективни процедури за управление. Това позволява на организацията да повтаря успешните дейности от проект на проект. Може обаче да се случи някои от дейностите (практиките) да не се повторят. По принцип, за да бъде един процес *ефективен*, той трябва да е приложим, документиран, измерим, с възможности за усъвършенстване и с обучени за приложението му кадри. Ръководството на всеки проект следи разходите и графиците. Дефинирани са стандарти и има стремеж към съблудаването им.



Фиг. 8.1. Схема на СММ

Ниво 3 се нарича *определен (defined)*. Тук стандартният процес за разработване и съпровождане на софтуер в организацията е документиран, включително технологичните и управленските процеси, като при това те се интегрирали. Така стандартизираният процес в рамките на СММ се нарича *стандартен софтуерен процес* на организацията.

Налична е специална група, отговорна за този процес. Действа програма за обучение на членовете на колектива в съответствие с изпълняваните от тях задачи. Съществува процедура за настройване на стандартния софтуерен процес към нуждите на всеки конкретен проект.

Накратко, на това ниво процесите са стандартни и непротиворечиви поради стабилността и повторяемостта на технологичните и управлениските дейности. Цената, графиците и функционалността са под контрол и качеството на софтуера се следи.

Ниво 4 се нарича *управляемо (managed)*. На това ниво организацията установява количествено измерими критерии за качество и се стреми към постигането им — както за софтуерните процеси, така и за софтуерните продукти. Производителността и качеството се измерват за важните софтуерни процеси през цялото време в рамките на специална програма. Всеобхватна база данни се ползва за събиране и анализ на параметрите на софтуерните процеси. За измерването им са предвидени инструментални средства. При излизането на тези параметри от определени граници може да се установи дали това е случайно явление, или са необходими коригиращи действия. При преминаването към нова област на приложение или нов инструментариум се пресмята и контролира рисъкът.

Накратко, организациите на това ниво са предсказуеми, защото работят с измерими процеси и в дефинирани измерими граници. Предсказуеми са както високото качество на процесите и продуктите, така и тенденциите в развитието му. Нарушаването на определените количествени граници подлежи на коригиране.

Ниво 5 се нарича *оптимизиращо (optimizing)*. На това ниво цялата организация е съсредоточена към непрекъснато подобряване на процесите. Тя има средствата за идентифициране на силните и слабите страни на процесите с цел предотвратяване на дефекти. Разполага се с данни за ефективността на софтуерния процес, които се използват за анализ на цената и ползата от въвеждането на нови технологии или изменения в него. Технологични инновации се идентифицират и евентуално се въвеждат в практиката. Дефектите се анализират до установяване на причините и избягване на повтарянето им както в текущия, така и в бъдещи проекти.

В табл. 8.1. наред с кратките определения на петте нива са дадени и данни за реалното разпределение на организациите (фирмите) по тези нива на основата на извършени изследвания в САЩ.

Ниво според СММ	Честота	Определение
1 = начално	75.0%	Примитивни и случайни процеси
2 = повторяемо	15.0%	Някои стандартизиранi методи и контроли
3 = определено	8.0%	Добре структурирани методи, добри резултати
4 = управляемо	1.5%	Висока сложност с повторна използваемост
5 = оптимизиращо	0.5%	Съответствиа на теорията, нови методи

Табл. 8.1. Разпределение на нивата на СММ

Ключови области на обработка

Вече беше казано какво представляват ключовите области на обработка. Тук ще изброим тези области така, както са разпределени по нива.

Ниво 2:

1. Управление на софтуерната конфигурация
2. Осигуряване качеството на софтуера

3. Управление на договорите с подизпълнителите
4. Проследяване на софтуерния проект
5. Планиране на софтуерния проект
6. Управление на изискванията (на потребителя)

Ниво 3:

1. Групови прегледи
2. Координация между групите
3. Технология на софтуерния продукт
4. Интеграция на управлението и технологиите
5. Програма за обучение
6. Дефиниране на софтуерния процес
7. Фокусиране върху организацията на процесите

Ниво 4:

1. Управление на качеството на софтуера
2. Управление на количественото оценяване на софтуерните процеси

Ниво 5:

1. Управление промените в процесите
2. Управление промените в технологиите
3. Предотвратяване на дефектите

Цели

Целите обозначават обхвата, границите и намеренията за всяка ключова област. Те, както се каза, се реализират чрез ключови практики. Когато дадена ключова практика се настройва към конкретен проект, целта е тази, която служи като ориентир, доколко настройката е запазила същността на практиката в рамките на областта.

Ключови практики

Всяка ключова практика се изразява с едно изречение, често следвано от по-подробно описание. Характерно за тях е, че те не отговарят на въпроса, как да се постигне дадена цел, а описват какво следва да се прави.

Пример

За илюстрация ще изберем *една от 18-те ключови области* и ще дадем в схематичен, съкратен, но достатъчно пълен вид нейното описание.

Ниво 3: Програма за обучение

Предназначенето на програмата за обучение е да развие умения и знания у членовете на колектива, така че те да изпълняват задачите си ефективно.

Програмата за обучение включва преди всичко идентификация на нуждите от обучение и след това осигуряване на такова обучение. В някои случаи обучението се извършва неформално (в течение на изпълняването на задачите чрез ръководство от страна на по-опитни членове на екипа), в други то се организира формално във вид на курсове или ръководено самообучение.

Целите на програмата за обучение са:

1. Да се планират дейностите по обучението.
2. Да се осигури обучение за уменията и знанията, необходими за изпълнение на управленските и технологичните роли.
3. Всяко лице, за което това е необходимо, да получи нужното му обучение.

Ключовите практики за осъществяване на програмата за обучение са:

1. Всеки софтуерен проект разработва и поддържа план за обучение, в който са указаны нуждите от обучение.
2. Планът за обучение се разработва и актуализира в съответствие с документирана процедура.
3. Обучението се извършва в съответствие с плана за обучение.
4. Курсовете за обучение, подгответи в организацията, се разработват в съответствие с вътрешни стандарти.
5. Съществува процедура, която се ползва и с чиято помощ се установява доколко дадено лице притежава или е придобило необходимите за неговата работа умения и знания.
6. Поддържа се архив за протичането на обучението.

8.3.2. Методологията BOOTSTRAP

СММ получава разпространение по целия свят. Все пак в някои отношения тя не удовлетворява потребителите в Европа. Основната причина е, че в Европа водещо значение имат стандартите по качеството ISO 9000—9004, които на практика се пренебрегват от СММ. Поради тази причина в рамките на европейската програма ESPRIT като проект 5441 в началото на 90-те години се разработва методологията BOOTSTRAP. Тя се основава на:

1. СММ като модел
2. Въпросника на СММ, чрез който се извършва оценката
3. Стандартите за качество ISO 9000, 9001, 9000-3.

Авторите на BOOTSTRAP определят *целите* й по следния начин:

- на основата на най-добрите софтуерни практики да се създаде средство за оценяване способността на дадена организация за ефективен софтуерен процес;
- да се отразят признатите технологични софтуерни стандарти;
- да се гарантира повторяемостта и устойчивостта на оценката;
- във всяка оценявана организация да може да се идентифицират слабите и силните страни на софтуерните процеси;
- да се осигури подкрепа за планиране на подобренията и съответстващи положителни резултати;
- да се подпомогне увеличаването на ефективността на процесите при Прилагането на стандартни изисквания.

База на BOOTSTRAP са:

1. *Процесът на оценяване.* Той се разглежда като част от цялостното подобряване. Резултатите от оценката са основният ресурс за създаване на план за подобряване. Оценката се прави на основата на модел на процесите.

2. *Моделът на процесите.* Аналогично на СММ и тук има нива на зрелост (макар да не се ползва точно този термин). Нивата са:

- ниво 0: непълен процес;
- ниво 1: изпълняван процес;
- ниво 2: управляван процес;
- ниво 3: установлен процес;
- ниво 4: предсказуем процес;
- ниво 5: оптимизиращ процес.

Идентифицирани са всички процеси и са структурирани във вид на дърво. Това дърво схематично е дадено на фиг. 8.2. Както се вижда от нея, коренът на дървото е цялостната **BOOTSTRAP**. На следващото ниво има 3 клона:

- *Организация*
- *Методология*
- *Технология*

Организацията и технологията за разбити съответно на 3 и 4 процеса. Като пример за организационен процес може да се посочи управлението на човешките ресурси, а на технологичен — обновяване на технологията.

Методологията се състои на следващо ниво от:

- процеси, *зависими от жизнения цикъл* (например анализ на изискванията към системата, проектиране на архитектурата на системата и т. н. до съпровождане и накрая отмиране);
- процеси, *независими от жизнения цикъл*, разбити на още едно ниво;
- „*метапроцесни*“ дейности — дефиниция на процесите и подобряване на процесите.

Накрая независимите от жизнения цикъл процеси се разделят на 3 групи:

- *управление* (например управление на проекта, управление на качеството, управление на риска);
- *вътрешно поддържане* (например документиране, управление на конфигурацията, верификация, валидация);
- *поддържане на потребителя* (например управление на нуждите иисканията на потребителите, доставка, експлоатация на софтуера).

3. *Въпросниците.* Оценката се основава, както при СММ, на отговорите, събрани по фиксирани въпросници. Тук те са два. Първият се отнася до организацията на софтуерните процеси във фирмата, а вторият — до отделните проекти. Това е отражение на разбирането, че в рамките на една софтуерна единица (фирма или дори нейно подразделение) може да има съществени разлики. Между другото това дава възможност да се сравняват екипите, работещи по отделни проекти.

4. *Оценяването и представянето на резултатите.* За да може подобряването на процесите и тяхното планиране да се осъществи на основата на оценките, необходимо е

те да са надеждни и да представят адекватно оценяваната фирма. Това става благодарение на това, че:

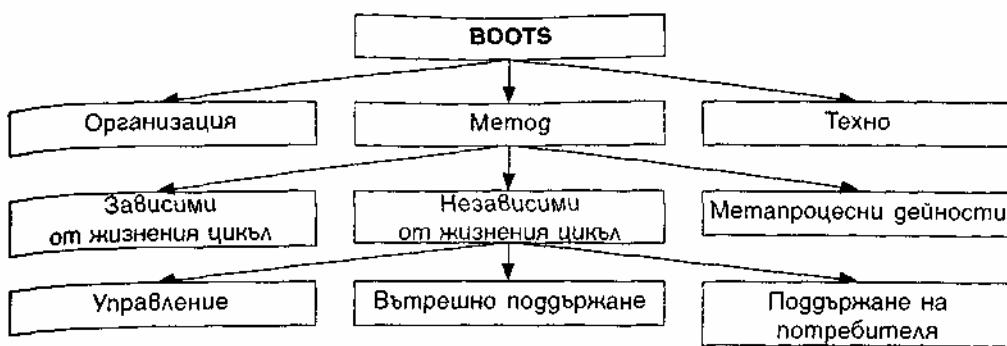
- оценителите имат еднаква подготовка и ползват единна методология — това се гарантира от специална система за акредитация;

- прилаганите правила за оценка са строго дефинирани.

Всеки атрибут на качеството на всеки процес се оценява по четиристепенна скала. След това по схема за агрегиране на тези оценки се получават оценки за нивата на зрелост — за фирмата и за отделните проекти.

5. *Указанията за подобряване на процесите.* Тези указания подпомагат идентифицирането на процесите, които са с най-съществено значение за по-

стигането целите на фирмата. След това най-висок приоритет за подобряване се дава на процесите с най-ниска оценка и най-голямо влияние. Оценява се рисъкът от неприлагането на мерки за подобряване. Ако това не е направено до момента, извършва се подготовка за постигането на изискванията на стандартите за качество ISO 9000—9004.



Фиг. 8.2. Схема на BOOTSTRAP

6. *Базата данни.* Всички данни от оценки автоматично се натрупват в централна база. Тя има за задача:

- да съхранява данни за резултатите от извършените оценки в Европа с цел да се създаде постепенно картина на софтуерната индустрия в Европа;

- да подпомага позиционирането на всяка оценявана фирма в рамките на съответния сектор от софтуерната индустрия в Европа.

8.4. Стандарти за качеството на софтуера

В областта на качеството на софтуера има най-разнообразни стандарти, отличаващи се по степен на общност, предназначение, степен на готовност за приложение, разпространение и др. Нито е възможно, нито е смислено тук да се разглеждат всички те. Поради това ще направим кратък преглед с цел създаване на обща представа, а след това ще обърнем повече внимание на тези, без прилагането или поне познаването на които създаването на съвременен софтуер не е препоръчително. Изрично ще отбележим, че някои от тях не визират точно качеството на софтуера, а по-общи или други проблеми на неговото създаване. Независимо от това те представляват интерес, защото самото прилагане на стандарти при производството на софтуер е едно от изискванията за осигуряването качеството на процеса и на продукта.

8.4.1. Национални и други стандарти с ограничено действие

В областта на *отбраната*, както е известно, особено се държи на стандартизацията, а и въпросът за качеството на софтуера е от първостепенно значение. Министерството на отбраната на САЩ освен споменатия по-горе в 8.2.4. стандарт *DOD-STD-2168 Software Quality Program* прилага и *DOD-STD 2167A Defense System Software Development*. Подобни стандарти има и в НАТО. Общи стандарти за качеството са *NATO AQAP-1, AQAP-13, AQAP-14*, като AQAP-13 дава допълнителни изисквания за софтуера, а AQAP-14 съдържа указания за оценяване на съответствието на продукта с изискванията. Има данни, че тези стандарти се ползват активно във Великобритания, по-малко в другите европейски страни — членки на НАТО, и никак — в САЩ. Що се отнася до AQAP-1 той се смята за еквивалентен на *ISO 9001* и вече е изместен от него.

В областта на *авиацията* САЩ имат специален стандарт *STD 018a-1987 Computer Software Quality Program Requirements*. Негов аналог в Европа е *ES4 PSS-05 Software Engineering Standards*.

Ядрената енергетика има също своите специални стандарти: *IAEA-1987 QA for Computer Software, BS 5882-80 Specification for a Total Quality Assurance Programme for Nuclear Power Plants*, прилаган във Великобритания. *ASME N45.2.11 Quality Assurance Requirements for the Design of Nuclear Power Plants* канадската серия от стандарти *Q396* за програми за осигуряване на качеството за критични и некритични приложения, испанският стандарт *UNE73-404* за осигуряване на качеството на информатични системи в ядрени инсталации.

Немалък брой страни имат също свои стандарти за качеството на софтуера. За Франция, Ирландия и Германия например те вече губят значението си във връзка с преминаването към единна система на стандарти в рамките на Европейския съюз. В Австралия и Нова Зеландия се прилага стандартът *AS3563 — Standard for Software Quality Management Systems*. Той се смята за по-добър за приложение от аналогичните от серията ISO 9000, като при това, ако дадена система за осигуряване на качеството удовлетворява единия от тях, тя съответства на практика и на втория.

8.4.2. Международни стандарти за софтуера

В световен мащаб съществува специално тяло, упълномощено да създава международни стандарти, отнасящи се до софтуера. То е образувано като общ орган на Обединения технически комитет 1 (JTC1) на Международната организация за стандартизация ISO и на подкомитет 7 (SC7), принадлежащ на Международната електротехническа комисия IEC. Нарича се ISO/IEC JTC1/SC7 Software Engineering.

Подготовката на даден стандарт се подчинява на строго формализирана процедура, чиято цел е да се отчете мнението на всички заинтересовани страни, да се постигне съгласие между тях, да се избегнат по възможност всякакви грешки, да се спазят стандартите относно формата и съдържанието. Обикновено в началните стадии на изработването на даден стандарт материалите за него са с ограничен достъп, но от един момент нататък стават достъпни и могат да бъдат намерени в Интернет. Обикновено даден стандарт е в състояние TR (Technical Report) — технически доклад, или IS (International Standard) — международен стандарт.

В момента има около 30 стандарта в областта на софтуера, но броят им нараства. Някои от тях са в състояние на технически доклади, но други са в сила. Нещо повече — за приетите има предвидена специална процедура за ревизия, която се провежда по

определен план. Сред тези стандарти ще споменем следните, пряко свързани с качеството на софтуера:

IS 9126-1 Information Technology — Software quality characteristics and metrics Part 1 — Quality characteristics and sub-characteristics

TR 15504-K: Information Technology — Software Process Assessment —

Part k (тук *k* е число от 1 до 9, т. е. това са 9 части на един стандарт, които тгетират различни аспекти на оценяването на софтуерния процес — увод, концепции, речник, указания и т. н.)

IS 12119 Information Technology — Software packages quality requirements and testing

IS 14598-5: Information Technology — Software Product Evaluation — Part 5: Process for evaluators

IS 14598-1 Information Technology — Software Product Evaluation — Part 1: General Overview.

8.4.3. Стандартите от серията ISO 9000

В последните години непрекъснато нараства значението на стандартите от серията ISO 9000. Във връзка с процесите на интегриране на България към Европейския съюз (ЕС) все по-важно става съблудоването им (съответно сертифицирането) от тези български фирми, които имат интерес да изнасят продукция в ЕС.

Серията ISO 9000 се появява през 1987 година. От 1992 година са задължителни за продажбата на продукти в ЕС. Основната ѝ цел е да даде стандарти и указания за създаване на такава организация на производство и обслужване, че получените в резултат продукти и услуги да са качествени.

Със софтуера имат пряка връзка следните от тези стандарти.

ISO 9001 е модел за осигуряване на качеството, състоящ се от 20 групи изисквания за качество. Този модел се отнася до организации, които проектират, разработват, произвеждат, инсталират и обслужват продукти. ISO очаква организациите да прилагат този модел, като създават и развиват съответна система за качеството.

ISO 9004-2, част 2, представлява указания за качеството на услугите, които са приложими към обслужването и на софтуерните продукти.

ISO 9000-3 представлява указания за приложението на ISO 9001 по отношение на разработването, доставката и съпровождането на софтуера.

За да илюстрираме тези указания, ще дадем *пример* с втората от 20-те групи указания — изисквания към качеството на системата:

- разработете план за контролиране на качеството на проектите за разработване на софтуер;
- разработете план за качеството навсякъде, където имате нужда да контролирате качеството на конкретен проект, продукт или договор;
- вашият план за качеството трябва да обяснява как имате намерение да създадете вашата система за качество, така че да е приложима към конкретния проект, продукт или договор;

— разработете подробни планове за качеството и процедури за контрол на Управлението на конфигурацията, на верификацията на продукта, на валидацията на продукта, на нестандартни продукти, на коригиращи действия.

Трябва да се признае, че тези указания имат прекалено общ характер и се Нуждаят от задълбочено конкретизиране и детализиране, за да станат реално приложими. Много е вероятно това да е причината за пренебрежването им в софтуерната индустрия в САЩ. Последното има и друго обяснение. То се илюстрира със следния цитат от [8]: „Единственият видим ефект от прилагането на ISO [9000—9004], изглежда, е цената на самия процес на сертифициране по ISO, изразходованото време за това сертифициране и нарастващото на обема на книжната документация, свързана с качеството.“

Литература

1. Sehulmeyer G.G. Zero Defect Software. New York, McGraw-Hill, 1990.
2. Humphrey W.S. Characterizing the Software Process: A Maturity Framework. IEEE Software, March 1988, p. 73—79.
3. Humphrey W.S. (ed.). Managing the Software Process. SEI (USA), Addison-Wesley, New York, 1989, 489 pp.
4. Paulk M. C, Weber C. V, Garcia S.M., Chrassis M.B., Bush M., Key Practices of the Capability Maturity Model for Software, Version 1.1, TR CMU/SEI-93-TR-025, Software Engineering Institute (USA), Carnegie Mellon University, Pittsburgh, 1993.
5. Jones C. Assessment and Control of Software Risks, Prentice-Hall, Englewood Cliffs, 1994.
6. BOOTSTRAP Team, „BOOTSTRAP: Europe's Assessment Method“, IEEE Software, July 1993, pp. 93—95.
7. Brown B.J., Assurance of Software Quality, SEI Curriculum Module SEI-CM-7-1.1, Carnegie Mellon University, Software Engineering Institute, 1987.
8. Jones C, Applied Software Measurement, McGraw-Hill, New York, 1997.

9. ОРГАНИЗАЦИОННО УПРАВЛЕНИЕ

9.1. Въведение

Според Б. Боем целта на софтуерните технологии е осигуряване на *ефективен* процес на разработване на *висококачествен* софтуер. Основна техника за преодоляване на стихийността в създаването на ПП е мениджърският подход. Разглеждането му е актуално заради все още тревожните данни за незавършване (изобщо или в срок) на софтуерните проекти, на превишаване на предварително определените ресурси за разработване или за разпространение на ПП, които затрудняват потребителите със сложността и ненадеждното си функциониране. По публикувани данни през 1998 г. 26% от софтуерните проекти са преустановени, а 46% са завършили с превишаване на планираните ресурси за разработване.

За съжаление не е възможно автоматично пренасяне в софтуерното производство на полезни мениджърски практики от материалното производство или други области поради:

- същността на софтуера. Програмните продукти не са чисто материални, а са продукт на интелектуални усилия, професионални умения, прилагане на научни знания и др.
- уникалността на всеки софтуерен проект, която се обуславя от естеството и сложността на конкретно решавания проблем. Затова могат да се определят само най-общите рамки, а реалното протичане на проекта се управлява с избири според случая (*ad hoc*) техники. Опитът от предишни проекти е полезен, но не може да бъде директно използван.

9.2. Основни понятия

Извършваните в една организация дейности могат да се разделят в две групи: продължителни, повтарящи се, стандартни и рутинни дейности или еднократни за създаване на нещо ново, уникално. Вторият вид дейности се реализират чрез *проекти*. Всеки проект има определена *цел*, която трябва да се постигне с крайни и ограничени *ресурси* (време, финансови средства, екип от специалисти). Обикновено изискванията се уточняват постепенно и поради уникалността на проекта участниците в него могат да разчитат на опита си, но трябва да са в състояние да разрешават и нововъзникнали проблеми, които е трудно да се предвидят отнапред. *Целта* на управлението на проекти е те да се осъществяват систематично и контролирано чрез постъпкови процедури. *Ефективни проекти* са тези, които водят до повишаване на производителността на участниците и осигуряване на качествени крайни продукти и услуги. Резултатите от тях са постигнати в определен срок и с оптимално използване на ресурсите. Предпоставки за ефективни проекти са [1]:

- създаване на благоприятна среда за работа (техническа осигуреност, подходящ психологически климат, ергономичност);

- формулиране на ясни, точни и непротиворечиви изисквания за работата, която трябва да се свърши;
- управляване на проекта чрез планиране и контрол;
- прилагане на ефективни, проверени (теоретично или експериментално) и стандартизириани процедури;
- осигуряване на адекватни ресурси, определени след предварителна оценка и прогнозиране;
- мотивиране на участниците в проекта. Задачите, които се възлагат, да бъдат точно определени, съобразени със знанията, квалификацията и опита на конкретния изпълнител. Да има свободни и добре организирани комуникации, регламентирани поощрения и възможности за личностна изява.

9.3. Управление на софтуерни проекти

9.3.1. Методология на управлението на проекти

Управлението на софтуерните проекти може да се декомпозира на управление на четири основни обекта (т. нар. four P's: people, product, process and project).

Управлението на *човешките ресурси* е преди всичко признаване на ролята на човешкия фактор в разработването и използването на софтуера (вж. глава 14.). С развитието на софтуерната психология в последните години стандартната за софтуерните системи двойка „хардуер-софтуер“ се разшири с трети елемент, наречен reopleware. От гледна точка на мениджмънта това са дейностите по подновяване и подбор на кадрите, повишаване на квалификацията, атестиране и професионално развитие, организация на работата и на работното място, обучение за работа в екип и т. н.

Управлението на *продукта* е преди всичко определяне на предназначението, основните функции и изисквания към новия продукт още в началото на проекта, за да се оценят предварително необходимите ресурси за реализацията му.

Управлението на *процеса на разработване* включва планиране и ефективно осъществяване на всички дейности. В зависимост от предназначението, обхвата и начина на организирането им дейностите могат да бъдат *основни* (проектиране, програмиране, тестване) или *допълнителни* (осигуряване на качеството, управление на софтуерните конфигурации и измерване). Избраният в софтуерната организация *модел на жизнения цикъл* определя осъществяването на основните дейности, общите характеристики на които се настройват за отразяване на специфичните особености на всеки проект. Глобалните дейности са едни и същи за всички проекти и за организирането им могат да се прилагат общоприети стандартни методики.

Основната цел на управлението на *проекта* е да се държат нещата под контрол. Всеки проект има определена начална и краяна дата, цел и обхват, определено и документирано съотношение цени-ползи, добре дефиниран краен резултат, ясни и точни критерии за завършване. Управлението на проекти е в съответствие с избрана *методология* — съвкупност от принципи, подходи, методи, процедури правила и указания, изясняващи как се осъществява определен проект и какви са взаимоотношенията му с други обекти и субекти. Методологията е *формална*, ако е

представена в писмен вид и се прилага систематично в дадена организация. Преимуществата на прилагане на формална методология са, че тя:

- подпомага избирането на проекти;
- позволява разпределение на ресурсите и контролирането на риска;
- управлява проекта чрез дефиниране на контролни точки;
- улеснява комуникациите между участниците чрез стандартизириани процедури и терминология;
- подпомага счетоводно-финансовите дейности;
- осигурява създаване на необходимата документация по време на проекта, а не след завършването му;
- подобрява качеството на създавания ПП чрез специфициране на проверки и задължително спазване на определени стандарти;
- намалява разходите за разработване чрез стандартизиране на основните процедури;
- улеснява натрупването на статистическа информация, която да се използва за прогнозиране на разходите и усъвършенстване на някои процедури при реализация на следващи проекти;
- дава възможност за сравнителен анализ на различни проекти;
- осигурява систематичност и управляемост на проектите.

9.3.2. Управленска структура

Традиционната *управленска структура* е йерархична, като нивата на йерархията, видът и броят на ръководителите на всяко ниво зависят от големината на софтуерната фирма и размаха на дейностите в нея. В общия случай се разграничават две нива: ръководители на фирмата (*senior managers*) и ръководители на проекти (*project managers*). Ръководителите на фирмата отговарят за:

- определяне на дългосрочните и краткосрочни бизнеспрограми, т. е. кои проекти ще се осъществяват и при какви условия;
- сключване на договори за разработване;
- назначаване на ръководители на проекти;
- ръководство на основни фирмени дейности, като фирмено планиране, Маркетинг, научноизследователска и развойна дейност, подбор и развитие на кадрите и др.

Ръководителите на проекти отговарят за:

- подготовката и съставянето на офертата за разработката;
- оценка на стойността и продължителността на проекта и необходимите за осъществяването му ресурси;
- сформиране на екип(и) за проекта и оценяване на работещите;
- съставяне на планове (бюджет, индивидуални планове) и графики за проекта;
- текущо управление на проекта (проследяване, отчитане в контролни точки);

— представяне на проекта и развитието му пред външни или горестоящи инстанции.

Практиката показва, че успехът на проекта зависи в голяма степен от качествата на ръководителя му. Затова са разработени и специални процедури за подбор на ръководителите на проекти. Обикновено се предпочитат лица с информатично или бизнесобразование с допълнителна специализация в областта на софтуерните технологии. Практическият опит в разработването на софтуер е съществено предимство. Професионалните изисквания се допълват от изисквания за лидерски качества, комуникационни умения и индивидуални характеристики, като говорчивост, способност за оценяване на нови идеи и създаване на подходящ микроклимат в екипа и т. н. Личната мотивация за превръщане на разработчик в ръководител на проект е освен по-високото заплащане приемането на предизвикателствата и отговорностите на нов проект, издигане в юрархията на софтуерната организация, чувството на удовлетворение от вземане на решения, водещи до успешно завършване на проекта, и не на последно място — упражняване на власт.

9.4. Планиране

Планирането обхваща дейностите по съставяне и проследяване на всички видове планове, т. е. разработване на планове и процедури за реализирането им, възлагане на задачи и проверка на изпълнението им. Планирането е основна управлена функция и не е предсказание, какво ще стане, а какво бихме искали да стане. Ръководството на фирмата определя дали и какво да се планира. Основни възражения срещу планирането са, че отнема много време, че е скъпо и ограничава гъвкавостта; че се основава на предположения, които може и да не са верни, че потиска творческия подход и доколкото всеки проект е уникален, не може да се използва рационално опитът от предишни проекти. В подкрепа на планирането се посочва, че то:

- подпомага осъществяването на интегрирани, целенасочени действия и резултати;
- служи за координация и контрол на всички дейности;
- увеличава ефективността, като предотвратява повторно извършване на определени действия;
- осигурява „прозрачност“ на управлението на проектите.

В зависимост от целите и предназначението си плановете се делят на общи и конкретни.

Общите планове се отнасят до софтуерната организация като цяло и са целева програма, стратегически и тактически план.

Целевата програма (*The Mission*) отговаря на въпроса, защо съществува софтуерната организация. Най-често целта е разработване на софтуер, който да се разпространява с цел да се реализира печалба. Но съществуват и софтуерни организации с идеална цел (например софтуерни отдели на научни институти), които разработват софтуер за популяризиране на нови научни постижения, създаване на *freeware* или *shareware* за издигане на авторитета на съответния научен институт и др. Специални са целите на разработване на софтуер за военните, за космическите изследвания и др.

Стратегическият план отговаря на въпроса, какво да създава софтуерната организация. Този план се създава за относително по-дълъг период от време (обикновено за 5 години) и определя в какви приложни области ще бъдат разработваните ПП, дали и как ще се разширява дейността на организацията, какви ще са нейните основни и

странични дейности (например съпровождане, сертифициране, документиране на софтуер и др.).

Тактическият план отговаря на въпроса, как ще се осъществяват планираните софтуерни проекти (в каква последователност, с какви ресурси, в каква среда на разработване). Той обикновено се съставя за една календарна година.

Конкретните планове се отнасят до отделен софтуерен проект. Освен плана за проекта се съставят бюджет, календарни планове и графици за групите, реализиращи проекта, както и индивидуални планове за всеки участник.

Съществен конкретен план е *бюджетът*, описващ разпределението на средствата за разработване. Съставянето на бюджета се подчинява на следните правила:

— съставя се от ръководителя на проекта със съдействието на финансист, запознат с действащата нормативна уредба;

— представлява писмен документ, който се утвърждава от ръководството на фирмата и спазването му е задължително. Препоръчва се регламентирането на процедура за внасяне на изменения, защото реалните проекти понякога са толкова големи и сложни, че е невъзможно в началото на проекта да се извърши прецизно определяне и разпределение на финансовите средства.

— бюджетът описва необходимите средства и начина на осигуряването им. Разходите са описани по видове, наречени пера. Основни пера са: за трудови възнаграждения (заедно с дължимите социални и здравни осигуровки и данъци), за закупуване или наемане на техника, за поддържане на средата за разработване (наем за помещението, ток, парно, вода, чистене и поддръжка), за консумативи (дискети, хартия, тонер), за командироувъчни (в случаите, в които възложителят или внедрителят са в друго населено място), за допълнителни услуги (въвеждане на начални данни, съставяне, превод и/или редактиране на документация, наемане на външни експерти). Обикновено средствата са „боядисани“ и не се допуска безконтролното им прехвърляне от едно перо в друго.

— ако проектът се осъществява в условия на инфлация и ще продължи повече от година, се препоръчва съставянето му в някакви условни единици, които са сравнително постоянни или добре регламентирани — например в долари или евро, в минимални заплати и т. н.

— препоръчва се да се предвидят контролни точки от проекта, в които да се анализира изпълнението на бюджета и ако е необходимо, да се актуализира.

9.5. Анализ и управление на риска

Под *risk* се разбира потенциален проблем, който може да възникне и да застраши успешното завършване на проекта. Всеки риск се характеризира с *неопределеност* (може да се появи или не) и *вредност* — винаги има нежелани последици или загуби. В зависимост от това, какво засягат, рисковете се делят на *рискове на проекта, технически* или *бизнесрискове*. В [3] е дадена и класификация на рисковете в зависимост от степента на прогнозируемост. *Очаквани* са рисковете, които могат да се идентифицират чрез анализ на проекта; *предвидими* са рисковете, които могат да се опишат въз основа на опита от предишни проекти, и *непредвидими* са рисковете, които просто се случват.

Основната идея на управлението на риска е да се анализират някои от най-вероятните рискови ситуации и да се направят опити за избягването им или за намаляване на последиците от тях. Допълнителните разходи за изследване на рисковите фактори се компенсират с предотвратяването на значителните щети които биха се

получили, ако рискът е действителен. Управлението на риска се осъществява в три стъпки: идентифициране, оценяване и ограничаване на възможните последици. За всеки риск тези стъпки са последователни, но дейностите за управление на различните рискове се преплитат (например някои рискове се контролират, докато нови рискове се идентифицират и оценяват).

Идентифицирането на риска е систематичен опит да се установи какво може да застраши успешната реализация на проекта. Рисковите фактори могат да се разделят на две групи: *общи* за всички софтуерни проекти и *специфични* за даден проект. Основна техника за идентифициране на риска е съставянето на въпросници, свързани с характеристики на основните застрашени обекти. Например за обекта проект въпросите се отнасят до характеристиките обхват, размер и продължителност; за обекта продукт — спецификации, сложност, уникалност и специални изисквания; за обекта персонал — организация, състав и мотивираност на работната група, качества на ръководителя на проекта; за обекта процес на разработване — процедура за избор на проекти, методология и стандарти за работа и др. За всеки риск се определя и степента му на влияние върху проекта.

Оценяване на риска — данните за риска се систематизират и се преобразуват в информация за вземане на решения. За всеки риск се определя вероятността и последиците от риска [3], въз основа на които се определя и приоритетът му.

Предотвратяване на последиците от риска изисква планиране на управлението на риска — определянето на рисковете с най-висок приоритет, които ще се наблюдават, и свързаните с тях действия за предотвратяването им. По време на разработването се следят идентифицираните рискови фактори и се натрупва информация за всеки от тях, за да се установи дали и как се променя вероятността за настъпване на рисковата ситуация.

Управлението на риска е свързано с допълнителни ресурси, но някои проблемни ситуации могат да се избегнат, като се спестят средствата за възстановяване на щетите след настъпването им. Политиката на следене на риска поддържа увереността на ръководителя, че всичко е под контрол, и увеличава шансовете за успешно завършване на проекта.

9.6. Организация на софтуерните проекти

След изясняване на основните параметри на проекта — съдържание, трудоемкост, продължителност и рискове — работата по проекта трябва да се организира така, че той да се вмести в определените срок и ресурси [4]. За тази цел се съставя и се следи изпълнението на календарен план-график, представлящ разработването на ПП във времето.

Съставянето на графици преминава през следните етапи:

1. Декомпозиране на всички дейности, необходими за реализацията на проекта, на множество от задачи.
2. Определяне на зависимостите между задачите. Някои задачи могат да се изпълняват само последователно, защото междинният продукт, създаван от една задача, е необходим за започването на друга задача. Други задачи са независими и могат да се изпълняват паралелно, т. е. едновременно.
3. Определяне на трудоемкостта на всяка задача в избрана мерна единица (например в човекодни) и дефиниране на начална и крайна дата за извършването ѝ.

4. Разпределяне на работата между членовете на групата така, че във всеки момент ресурсите за извършване на провеждащите се в този момент задачи да не надвишават отпуснатите за проекта ресурси.
5. Определяне на персонални отговорници за всяка задача.
6. Определяне на получуваните изходи от всяка задача.
7. Дефиниране на контролни точки, при достигането на които ще се анализират и оценяват получените междинни продукти.

Като пример ще разгледаме съставянето на график за дадена фаза на софтуерен проект. Нека за тази фаза да са определени три стандартни задачи, изискващи едни и същи умения и квалификация на разработчиците. Въз основа на информация от предишни проекти е определена трудоемкостта на всяка от задачите. Данните от предишни проекти показват, че около 60% от усилията в тази фаза са продуктивни. Следователно от 8 работни часа само 5 ще са продуктивни.

Тези предварителни оценки са представени в таблицата на фиг. 9.1.

Задачи	Трудоемкост (човекочаса)
Задача 1	100
Задача 2	150
Задача 3	150

Фиг. 9.1.

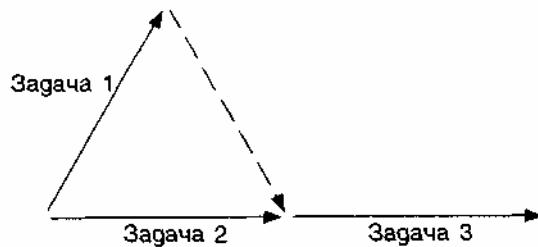
Ръководителят на проекта оценява особеностите на проекта и планира следната трудоемкост на задачите:

Анализирайки същността на задачите, ръководителят на проекта установява, че първите две задачи могат да се изпълняват паралелно, но третата може да започне едва след завършването на първите две.

Задача	Трудоемкост (човекочаса)	Трудоемкост (човекодни)
Задача 1	90	18
Задача 2	165	33
Задача 3	170	34

Фиг. 9.2.

Наредбата на задачите е следната:

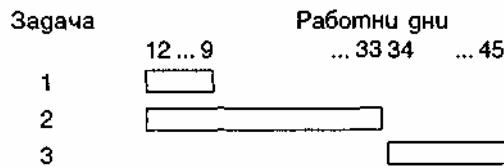


Фиг. 9.3.

Първоначално ръководителят на проекта разпределя работата на разработчиците по задачите по следния начин:

Задача 1	18 човекодни	2 души	9 дни
Задача 2	33 човекодни	1 човек	33 дни
Задача 3	34 човекодни	3 души	12 дни

Съответната Гант-диаграма показва, че за цялата фаза ще са необходими 45 дни.



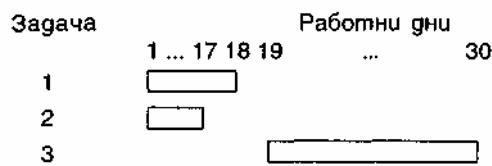
Фиг. 9.4.

Веднага се виждат два основни недостатъка на този график — фазата продължава тръдре дълго и средната производителност е около 40%. Затова ръководителят на проекта решава да промени разпределението на разработчиците по задачи, като определя 1 човек за първата задача, двама души за втората и трима за третата:

Задача 1	18 човекодни	1 човек	18 дни
Задача 2	33 човекодни	2 души	17 дни
Задача 3	34 човекодни	3 души	12 дни

Съответната Гант-диаграма на Фиг. 9.5. показва, че за цялата фаза са необходими 30 дни и средната производителност вече е 60%.

Този график е по-добър, защото продължителността на проекта е 30 дни (при 29 дни от предварителната оценка) и производителността е близка до очакваната.



Фиг. 9.5.

Освен съставянето на графици организирането на проекта включва и проверка, дали те се спазват. Проследяването на проекти става чрез регламентиране на система за контрол. Контролират се основните параметри на проекта — размер на междинните продукти, производителност на разработчиците, цена на разработката, продължителност. Текущото състояние се сравнява с планираното и при несъответствия се планират коригиращи действия. Основен принцип на контрола на проекта е, че се контролира работата, а не работещите. Контролът се основава на проверки за свършената работа по всяка задача и осъществяване на обратна връзка. Реализира се чрез последователност от проверки, чрез които се натрупват контролни данни — отчети, справки, протоколи, експертни оценки и др. Основни техники за събиране на данни са провеждане на интервюта, прилагане на инструментални средства, реализиращи процедурите на измерване на зададени метрики, или преглеждане на материали, свързани с реализацијата на проекта. Данните се проверяват за правилност и непротиворечивост (чрез сравняване с исторически данни, анализ от експерти или чрез проверка на процедурите за събирането им). Проверените данни се използват за оперативно управление на проекта. Съществено е, че контролът трябва да обхваща стандартните дейности и ситуации, а не изключенията. Нивата на контрол са обикновено четири: организация, отдел, проект, работна група. За всяко ниво на контрол са определени съответни процедури.

9.7. Примерен план на софтуерен проект

Изложените по-горе управленски концепции и принципи се илюстрират чрез съставяне на план на проекта, който след утвърждаването си е задължителен документ, направляващ систематичната и организирана реализация на проекта. Предлаганата в [4] структура на плана е следната:

ПЛАН

I. Въведение

А. Предназначение на документа

Б. Цели на проекта

Цели

Основни функции

Изисквания към функционирането на продукта

Управленски и технически ограничения

II. Оценка на проекта

А. Исторически данни, използвани за оценката Б. Техники за оценяване

В. Оценки

III. Рискове за проекта А. Анализ на риска

Идентифициране на рисковите фактори Оценка на риска

Б. Управление на риска

Техники за избягване на рисковите ситуации

Процедури за следене на рисковите фактори

IV. Управление на проекта чрез календарни планове и графики

А. Декомпозиране на работата по проекта на основни задачи Б. Съставяне на мрежов график

В. Разпределение на ресурсите

V. Ресурси

А. Човешки ресурси

Б. Хардуерни и софтуерни ресурси

В. Специални ресурси

VI. Организация на екипа, работещ по проекта А. Структура и състав на работната група

Б. Проследяване и отчитане на извършената работа

VII. Проследяване и контролиране на проекта**VIII. Приложения****9.8. Правила за осъществяване на софтуерни проекти**

Натрупаният практически опит дава възможност да се формулират основни правила, спазването на които би повишило ефективността на реализираните проекти. Списъкът от правила не е подреден, няма претенции за изчерпателност и може да се разглежда като споделяне на know-how.

1. *Комуникации.* Регламентират се начините на комуникации между договарящите се страни и всички участници в проекта — форма и честота на провеждане, организиране и документиране.

2. *Определяне на крайния срок.* Задава се продължителността на проекта в дни или месеци. Обикновено се определя датата, до която проектът трябва да бъде завършен.

3. *Възлагане на проекта* — само чрез писмен договор, подписан и одобрен от договарящите се страни.

4. *Подизпълнители.* Изпълнителят може да възложи части от проекта на трета страна само след изричното съгласие на възложителя въз основа на под-договор, и то само ако са спазени изискванията към процеса на разработка и към създавания продукт.

5. *Осигуряване на необходимата документация.* Възложителят предоставя или съдейства за осигуряването на документация, материали, справки за законови или ведомствени разпоредби и други. Всички материали се връщат след завършване на проекта и съдържанието им е фирмена тайна.

6. *Финансова дисциплина.* Изпълнителят трябва да документира всички разходи. Възложителят има право да изпрати финансови ревизори, на които се предоставят всички необходими документи. Ревизията завършва с ревизионен акт.

7. *Задължения на изпълнителя:*

- да извърши съответната работа в договорения срок, обем и качество, изпълнявайки всички изисквания, определени в договора и приложенията към него;
- да спазва законите и действащата нормативна база, а при възникване на особени ситуации да уведомява възложителя;
- да реализира проекта ефективно.

8. *Конфиденциалност.* Фирмената информация, до която изпълнителят получава достъп при реализацията на проекта, не може да бъде разгласявана.

9. *Публични изяви,* свързани с новоразработвания продукт, да се правят с цитиране на възложителя и само с изричното му съгласие.

10. *Индустриална и интелектуална собственост.* Всички първични данни и материали (диаграми, схеми, проекти, отчети, статистики, изводи и др.) са собственост на възложителя. Изпълнителят може да има копия от тях, за да изпълнява проекта, но не може да ги използва и разпространява без изричното съгласие на възложителя. Изпълнението на проекта не може да наруши чужди авторски права.

11. *Независимост.* По принцип изпълнението на проекта не може да се обвързва с цели и средства на трета страна. При доказана зависимост договорът може да бъде прекратен еднострочно от възложителя, който освен това може да потърси правата си чрез съда.

12. *Работна група.* Определя се в началото на проекта. Структурата и съставът ѝ се описват в плана на разработката и се променят само със съгласието на възложителя в случаите на невъзможност за участие или доказана некомпетентност. Работната група може да се разширява, за да се спази определен срок, но ако разширението не е съгласувано и одобрено от възложителя, е за сметка на изпълнителя.

13. *Проблеми.* Възложителят оказва съдействие за разрешаване на процедурни, бюрократични или други трудности, възникнали при осъществяване на проекта.

14. *Отчети.* Формата, броят и съдържанието на отчетите се определят в началото на проекта и се описват в плана за реализацията му. Отчетите могат да бъдат планирани или извънредни. След завършване на проекта се изготвя заключителен отчет, който освен финансов анализ е и обзор на проблемите, възникнали при осъществяване на проекта.

15. *Закъсняване на проекти.* Санкциите се описват в договора и могат да бъдат прекратяване на договора, продължаване за сметка на изпълнителя или плащане на неустойки.

16. *Удължаване на срока.* Причини могат да бъдат формулиране на нови Изисквания, неизпълнение на задължения на възложителя или непредвидени обстоятелства. Препоръчва се предварително уведомяване и след разглеждане от Двете страни — сключване на допълнително споразумение.

17. *Изменения на характеристики на проекта.* Съгласуват се с изпълнителя, като се договарят измененията в цената и сроковете, които се отразяват в анекс към договора.

18. *Ползване на отпуск.* Ако не е договорено друго, препоръчително е да не се разрешава отпуск на членове на екипа, ако продължителността на проекта е под 6 месеца. При по-дълги проекти може да се разрешава, като периодът на отпуска се определя така, че да не се наруши планираното изпълнение на проекта.

19. *Плащания.* Плащанията не могат да надхвърлят договорената стойност на проекта. Схемата за плащанията се приема в началото на проекта и не подлежи на промяна. Броят на междуинните плащания зависи от продължителността на проекта, като

е прието поне 20% да се изплащат след окончателното приемане на проекта. В условия на инфлация се допуска индексиране на цените по предварително договорена формула.

20. *Прекратяване на договора.* Договаря се при какви случаи е възможно едностренно прекратяване, с какъв срок на предизвестие и с какви финансови последици.

21. *Особени ситуации.* При възникване на извънредни обстоятелства, възпрепятстващи изпълнението на задълженията на някоя от страните, се подписва допълнително споразумение за последствията.

22. *Разрешаване на противоречия.* Противоречията и споровете се разрешават чрез преговори, а ако не се постигне съгласие — по съдебен път.

Литература

1. Page-Jones, Practical Project Management, Dorset House, 1985.
2. Pressman, R. Software Engineering — A Practitioner's Approach. R.S. Pressman & Associates, Inc. 2000.
3. Charette, R., Software Engineering Risk Analysis and Management, McGraw-Hill/Intertext, 1989.
4. Spinner, M., Elements of Project Management: Plan, Schedule and Control. Prentice-hall, Inc. Englewood Cliffs, New Jersey, 1981.

10. ЦЕНА НА СОФТУЕРА

10.1. Необходимост и цели

Всеки ръководител на софтуерен проект би желал във възможно най-ранен момент от жизнения цикъл на разработвания програмен продукт да знае колкото се може по-точно какви разходи ще трябва да се направят до завършването му и колко ще продължи целият процес.

С този проблем теоретиците започват да се занимават още през 60-те години и в резултат се появява и първият метод за оценяване разходите по произвеждането на даден програмен продукт — SDC, 1965 г. Последват го още няколко метода, основани на съответни модели, докато се стига до 1981 година, когато се появява COCOMO, предложен от Боем [1].

Тази разработка се смята за фундаментална — от една страна, тя намира реално приложение в практиката, а от друга — поставя на здрави основи последващите изследвания по цената и по-общо — по икономиката на разработването на софтуер.

Оказва се обаче, че процесът на намиране цената на разработване води до резултати, полезни и в други насоки [2]:

- *избор на проект за реализация* — очевидно един от решаващите фактори при такъв избор е цената на бъдещата разработка;
- *определяне състава на колектива разработчик*, което също е в пряка зависимост от установената цена и, обратно — изменяйки състава (по брой, квалификация, опит и др.), можем пряко да влияем върху цената на разработката;
- *определяне на маркетинговата политика* по отношение разработвания софтуер — както е известно, цената е един от основните компоненти на т. нар. маркетингов микс (вж.11.4.2.);
- *оценяване работата на членовете на колектива* — това може да става на основата на сравнение на предварително получените оценки с крайните резултати от работата.

Един от много важните приноси на Боем в споменатия труд [1] е, че той формулира множество от критерии, които следва да удовлетворява даден модел, респективно метод, за установяване цената на разработване на софтуерни продукти.

10.2. Критерии

Критериите, формулирани от Боем, са следните.

10.2.1. Определеност

Този критерий означава точно и обективно определяне на началните данни и понятия, както и на крайните резултати и количествените характеристики. Доколкото всеки метод на оценка се основава на някакъв модел на жизнения цикъл, то е много важно да се знае кои са точно фазите му и какво е тяхното съдържание, какви функции се изпълняват по време на всяка фаза и в какво точно се състои всяка от тях. Смята се, че

една от причините моделът СОСОМО да е получил широка известност и приложение е тъкмо отличното спазване на този критерий.

10.2.2. Точност

Има се предвид съответствието между предсказаната от модела цена на разработване и реално получилата се накрая. Изглежда, че наред със същността на метода от голямо значение е определеността и точността на входните данни, както и квалификацията на експертите, които правят оценката.

Има различни начини за повишаване на точността. Колкото и авторите да се опитват да направят модела и метода си универсален, по правило той работи най-добре за определен клас програмни продукти. При всеки опит за прилагането му извън този клас точността на резултатите рязко се влошава и тогава се прави опит за настройка — промяна на базовите уравнения, добавяне на нови фактори, премахване на нерелевантни такива.

Има твърде много изследвания, които показват недобра точност на немалко от методите дори при прилагането им към „подходящи“ за метода продукти. Така например в [4] на основата на 15 големи проекта с прилагането на 4 различни модела е показано, че при определянето *продължителността* на няколко (вече завършени) проекта грешката се е движила между 85% и 772%. Макар и не така разочароваващи, грешки са показани и в точността на предсказването на *цената* на разработван софтуер.

10.2.3. Обективност

Както и при други оценявачи процедури, важен критерий е постигането на възможно най-голяма обективност, или иначе казано — максимално избягване на субективния фактор. Навсякъде, където се очаква лична оценка от страна на експерти, се изготвят предварително указания, въвеждат се тегла, определянето на чито стойности се формализира в рамките на възможното, предоставят се еталони.

10.2.4. Детайлност

По принцип колкото един модел е по-подробен, толкова той е по-адекватен на реалните обекти и процеси и следва основаният на него метод да даде поточни резултати. За съжаление по-дълбоката детайлност изисква както повече ресурси (време, хора, пари) в етапа на разработване на модела, така и при прилагането му в конкретни случаи. При това положение се търси компромис между цената на извършваната оценка и желаната степен на точност. В споменатия вече СОСОМО този проблем е решен с помощта на предлаганите 3 версии на модела — базова, междинна и детайлна.

10.2.5. Устойчивост

Този критерий е въведен от Боем с цел отделяне на неподлежащи на оценка разработки поради съществуващи граници на приложение на метода. Така например неустойчив се оказва моделът DOTY, при който започват да се наблюдават силни отклонения в оценките при продукти с над 10 000 реда първичен код. Самият Боем признава, че неговият модел СОСОМО също може да се покаже като неустойчив при оценяване на малки проекти — до 2 000 реда първичен код. Една от причините за това

явление е, че всички модели са създавани на основата на изследване на големи и много големи проекти и уравненията, чрез които се пресмятат оценките, се базират на тях. Впрочем има правен опит — моделът РЗ (Programmer's Personal Planner), предназначен за относително малки проекти — до 18 000 първични реда и до колектив от максимум 3 души. Подобен модел представлява особен интерес днес, при непрекъснато растящите нужди от малки програмни продукти за персоналните компютри.

10.2.6. Област на приложение

Както казахме, няма универсален модел за определяне цената и продължителността на разработване на софтуерен продукт. Следователно за всеки модел трябва да може ясно да се определи областта на приложение. Така например PRICE S е предназначен за оценка на аерокосмически софтуер, знае се, че методът на функционалните точки работи най-добре за информационни системи и бизнесприложения. Областта на приложение на някои модели е по-широва благодарение на възможността определени коефициенти в уравненията за оценка да бъдат предварително променяни. Отново пример е СОСМО, както ще се види по-нататък.

10.2.7. Конструктивност

Всеки модел трябва да дава възможност да бъдат анализирани и разбрани получените резултати. Както се видя, в известни случаи даден модел води до по-ниски или по-високи резултати от реално получените накрая. Винаги обаче трябва да бъде възможно тези отклонения да бъдат обяснени. Целта е конструктивна — подобряване на модела.

Към тази характеристика спада и още една особеност. Моделът трябва да бъде конструктивен и в смисъл, че допуска получаването на различни резултати в зависимост от това, на какъв фактор потребителят придава по-голямо значение. С други думи, моделът може да предлага различни алтернативи според това, дали се държи на скъсяване на сроковете (разбира се, в допустимите граници) за сметка на ползването на повече или по-скъпа работна сила или пък точно обратното.

10.2.8. Простота на прилагане

Този критерий определя степента на трудност на разбирането и получаването на входните данни, както и степента на трудност на изпълнение на процедурите по оценяване. По принцип по-добра точност се получава при по-голяма детайлност, която обикновено се реализира йерархично. В този случай обаче нараства значително трудоемкостта, а и множеството фактори също усложняват прилагането на метода. Ясно е, че става дума за критерий, който е в пряка зависимост с други критерии — точност, определеност. Тази бележка впрочем може да се обобщи — десетте разглеждани критерия са в голяма степен зависими и често подобрене в един от тях води до влошаване в други.

10.2.9. Предсказуемост

Този критерий засяга проблема за практическото използване на моделите. Ясно е, че стремежът на всеки потребител е да получи възможно най-точна оценка колкото се

може по-рано в процеса на разработване на софтуер. По-нататък, при разглеждането на модела СОСМО ще видим, че той се основава на броя редове първичен код. За съжаление това не е толкова лесно предсказуем параметър, т. е. дори и много голям опит не може да гарантира сравнително точно определяне на този брой в началната фаза на жизнения цикъл на софтуерния продукт. Значителна част от изследователските усилия са били и продължават да бъдат съсредоточени тъкмо на този проблем: като входни данни — основа на съответния метод да се избере точно и раннопредсказуем фактор.

10.2.10. Икономичност

Това изискване е очевидно, не и то като много от вече изброените има компромисен характер. От една страна, желателно е входните данни за оценката да бъдат по-малко на брой и лесно измерими, от друга — да не са прекалено малко, така че да доведат до фатална неточност на крайните резултати. С други думи — желателна е икономичност, но не и на всяка цена. При отделни модели е забелязано, че когато се преминава от една област на приложение към друга, някои от факторите се оказват в новата област толкова взаимосвързани, че отпадането на някои от тях става възможно. По този начин се повишава икономичността на модела.

10.3. Моделът на Боем СОСМО

10.3.1. Цели и основни идеи

СОСМО произлиза от Constructive Cost Model.

Основната му цел е за всеки планиран софтуерен проект да се оцени *цена-та и срокът* на разработване.

Основополагащите му идеи е използването броя редове първичен код.

Първоначално предложеният модел е усъвършенстван в различни направления — чрез въвеждане на 3 нива на подробност, чрез отличаване на 3 типа на разработване, чрез въвеждане на коригиращи кофициенти за определящите параметри.

10.3.2. Същност на модела

За оценяване на трудоемкостта на даден софтуерен проект се прилага формулата:

$$\text{ЧМ} = 2.4 \times \text{ХРПК}^{1.05},$$

където ЧМ означава брой човекомесеца

ХРПК означава хиляди реда първичен код.

За оценяване продължителността на разработване на софтуерния проект формулата е:

$$B = 2.5 \times \text{ЧМ}^{0.38},$$
 където B е срокът на разработване в месеци.

Формулите се прилагат при следните *предположения*:

- а) редовете първичен код (т. е. тези, които се пишат на някакъв език за програмиране):
- се броят без коментарните редове;
 - принадлежат на крайния продукт (а не на междинни негови версии);
 - не включват използваните стандартни програми;
- б) включват се само фазите проектиране, програмиране и оценка, включително усилията по управлението и документирането по време на тези фази;
- в) не се включват обучението, планирането и инсталиранието на софтуера при потребителя;
- г) включва се трудът на проектантите и програмистите, но не и този на компютърните оператори, висшите ръководители и секретарките;
- д) смята се, че един човекомесец е от 19 дни, или 152 часа;
- е) предполага се, че никакви сериозни промени не се правят в продукта след одобряването на документа, който съдържа изискванията към него;
- ж) двете страни — потребителят и разработчикът — се предполага, че са Добросъвестни през цялото време.

10.3.3. Пример и следствия

За илюстрация да дадем един прост пример.

Да предположим, че в резултат на предварителна експертиза бъдещият софтуерен продукт се оценява на 32 000 реда първичен код. Като приложим двете формули, ще получим следните числови резултати:

$$\text{ЧМ} = 2.4 \times 32^{1.05} = 91 \text{ (човекомесеца)} \quad B = 2.5 \times 91^{0.38} = 14 \text{ (месеца)}$$

От тези базови резултати могат да се получат още две важни характеристики:

Производителност: $32\ 000 / 91 = 352$ реда първичен код за човекомесец

Екип. $91 / 14 = 6.5$ человека.

Тълкуванието на последната бройка е ясно — един или повече специалисти от екипа ще бъдат заети с проекта не през всичките 14 месеца на разработката.

Правени са изследвания за това, как се изменя производителността с промяната на размера на проекта. Боем предлага едно условно разделяне на проектите на малки, междинни, средни и големи (колкото и възражения да предизвика една такава класификация). За тези 4 категории са получени следните данни, показани в табл. 10.1:

Тип проект	ХРПК	ЧМ	B	Екип: брой	Производителност
Малък	2	5,0	4,6	1,1	400
Междинен	8	21,3	8,0	2,7	376
Среден	32	91,0	14,0	6,5	352
Голям	128	392,0	24,0	16,0	325

Табл. 10.1. Типове проекти по големина

Намаляването на производителността с нарастването на размера на проекта е естествена — тя се обяснява с увеличаването на броя на взаимодействията между растящия брой членове на колектива разработчик, всяко от които изисква допълнителни усилия и следователно отнема от времето за директно създаване на първичния код.

10.3.4. Усъвършенстване на модела

Първото усъвършенстване на дотук изложения базов модел е въвеждането на 3 типа софтуерни проекти — *разпространен*, *полунезависим* и *вграден* (макар и немного сполучливи, това са използвани вече в нашата литература преводи на оригиналните термини *organic, semidetached, embedded*). За всеки от тях формулата е различна. В табл. 10.2. всеки тип се характеризира кратко, илюстрира се с примери и му се съпоставя предложената от Боем формула.

Следващото усъвършенстване е свързано с установяването на факта, че все пак редовете първичен код не биха могли да са единственият параметър на установената формула. Преминава се към по-сложни модели — *междинен (intermediate)* и *детайлен (detailed)*, в които се отчитат допълнителни фактори-Те заедно с възможните *им рейтинги* са дадени в табл. 10.3.

Тип	Характеристика	Примери	формула
Разпространен	Разработва се от малка група в познатите условия на собствената фирма	Прости системи за управление на запаси или производствени процеси	$ЧМ = 2.4 \times ХРПК^{1.05}$
Полунезависим	Има междинно положение между разпространен и вграден тип	Системи за обработка на съобщения, нови операционни системи	$ЧМ = 3.0 \times ХРПК^{1.12}$
Вграден	Софтуерът работи свързан с апаратура, друг софтуер и изчислителни процеси	Авиационни или радиоелектронни системи	$ЧМ = 3.6 \times ХРПК^{1.20}$

Табл. 10.2. Типове софтуерни проекти

Оценявани атрибути	Много нисък	Нисък	Нормален	Висок	Много висок	Свръхвисок
Атрибути на продукта						
Изисквана надеждност	0,75	0,88	1,00	1,15	1,40	
Размер на базата данни		0,94	1,00	1,08	1,16	
Сложност на продукта	0,70	0,85	1,00	1,15	1,30	1,65
Атрибути на компютъра						
Ограничение по бързина			1,00	1,11	1,30	1,65
Ограничение по оперативна памет			1,00	1,06	1,21	1,65
Изменяемост на виртуалната машина		0,87	1,00	1,15	1,30	
Цикъл на обръщение към компютъра		0,87	1,00	1,07	1,15	
Атрибути на изпълнителя						
Квалификация на проектантите	1,46	1,19	1,00	0,86	0,71	
Опит в приложната област	1,29	1,13	1,00	0,91	0,82	
Квалификация на програмистите	1,42	1,17	1,00	0,86	0,70	
Опит от работа с компютъра	1,21	1,10	1,00	0,90		
Опит с езика за програмиране	1,14	1,07	1,00	0,95		
Атрибути на проекта						
Прилагане на съвременни методи	1,24	1,10	1,00	0,91	0,82	
ползване инструментални средства	1,24	1,10	1,00	0,91	0,83	
Ограничения в срока на разработка	1,23	1,08	1,00	1,04	1,10	

Табл. 10.3. Атрибути и рейтинги

При извършването на оценката експертите следва да се ръководят от тази таблица и за конкретния проект трябва да определят за всеки атрибут съответния рейтинг. Получените стойности се заместват в формулата, която вече е придобила по-сложен вид:

$$\text{ЧМ} = k \times \text{ХРПК}^P \times A_1 \times A_2 \dots \times A_{15},$$

където ЧМ и ХРПК са вече познатите величини,

к и р са коефициенти, които са различни за различните типове софтуер, както се вижда и от табл. 10.2.

А, са рейтингите на атрибутите

Разликата в междинния и детайлния модел е в начина на получаване на крайната оценка. Основната разлика е в това, че при детайлния вариант се прилагат различни коригиращи коефициенти (рейтинги) за всяка фаза от производствения процес. Посложен е и процесът, чрез който от оценките на отделните модули се получават оценки за подсистемите и от тях — за цялостния продукт. При междинния вариант нивото модул се игнорира.

10.3.5. Критика на „редове първичен код“

Основните критики към СОСОМО се свеждат до следните две:

- Няма нито стандарт, нито единно виждане за това, какво е „ред първичен код“.
- Много е трудно дори за експерти с голям опит да предскажат достатъчно точно в ранен етап на разработването броя на редовете първичен код.

По въпроса за редовете първичен код се изтъкват следните проблеми:

1. Какво всъщност да се разглежда — *физически или логически редове*. Физическият край на даден ред се получава с натискането на клавиша Enter, което води до генериране на съответни служебни символи за край на реда. Логическият край е някакъв ограничител, зависещ от конкретния език за програмиране, например двоеточие, запетая или друг точно определен знак. Езици, които позволяват написването на няколко оператора на един ред, могат да дадат неколократна разлика при двата вида броене. Същото може да се случи в обратна посока и когато (за прегледност или по други причини) един логически оператор се разполага върху 2 или повече реда. Показателно е едно изследване [7], което показва, че в САЩ 35% от ръководителите на проекти броят физическите редове, 15% броят логическите, а останалите 50% въобще не смятат за уместно да борят редовете първичен код.

2. Друг елемент на несигурност внасят различните от семантична гледна точка *типове редове*. Почти всеки процедурен език включва 4 типа първични редове:

- *изпълними оператори*, чрез които се задават различни операции (логически, аритметични, вход/изход и др.);
- *определения на данни*, чрез които се дефинират различните типове данни;
- *коментари*, които дават разясняваща информация;
- *празни редове*, които се ползват за повишаване прегледността на програмата.

Установено е, че в дадено типично бизнесприложение около 40% от общия брой оператори за изпълними, 35% са определения на данни, 15% са коментари и 10% — празни редове. При системния софтуер (например операционни

системи) 45% са изпълними оператори, 30% са определения на данни и отново 15% са коментари и 10% празни редове.

Правени са изследвания сред създателите на софтуер и отново е установено разлике в разбиранятията — 10% броят само изпълнимите редове, 20 % броят изпълнимите редове и определенията на данни, 15% добавят коментарите, а 5% броят дори и празните редове. Както вече беше казано, 50% въобще не броят редовете първичен код.

3. Голям проблем е *повторно използваемият код*. По съвсем общи оценки един програмист на традиционните процедурни езици за програмиране Фор-тран, С, КОБОЛ, средно копира от други програми между 20 и 30% код в своите програми. За обектно ориентирани езици като Smalltalk, C++ този процент достига до 50. Има отделни софтуерни фирми, които са организирали специални библиотеки от модули за повторно използване и там въпросният процент достига дори 75. При това положение спорът се върти около това как да се брои даден повторно използван модул:

- да се брои при всяко негово появяване;
- да се брои само веднъж независимо от броя появявания;

— въобще да не се брои, доколкото този модул не е разработван за оценявания проект.

Изследванията показват, че в САЩ първата алтернатива се прилага от 25% от професионалистите, втората — от 20%, третата — от 5%. За останалите 50% вече се разбра, че не боят.

4. Друга област на несигурност е как да се ползват редовете първичен код при приложения, писани на *повече от един език*. Има данни, че около една трета от софтуера в САЩ е написан на повече от един език за програмиране. Колкото и да изглежда невероятно, дори около 1996 година най-срещаните комбинации от езици все още са били следните:

- КОБОЛ заедно с някакъв език за обработка на заявки за търсене от типа на SQL;
- КОБОЛ и език за дефиниция на данни от типа на DL/1;
- КОБОЛ и някакъв специализиран език;
- С и Асемблер;
- АДА и Асемблер.

Доколкото, както стана вече ясно, няма стандарти и единство дори при броенето в приложения с един език, още по-малко може да се очаква такова единство при софтуер, написан на повече от език.

5. Има и някои *допълнителни*, по-малки проблеми от рода на това да се включва или изключва временно поставен в програмата код, код от типа мак-**рос**, код, свързан с управлението на заданието в рамките на операционната система. Трудности предизвиква и разликата в *стила* на програмистите. Преди време в IBM направили малък експеримент, като възложили на 8 програмисти Да напишат първичен код по зададена спецификация. Разликата в броя редове първичен код (преброени, естествено, по една и съща фиксирана методика) между-ДУ най-големия и най-малкия била петкратна.

Независимо от критиката по горните 5 пункта моделът СОСОМО продължава да има своите привърженици и да се прилага. Причината е в особената сложност и динамика на софтуера и неговото създаване и невъзможността, по-йе за момента, да се предложи напълно адекватен метод и модел. Това е и причината да са създадени и да продължават да се предлагат и да се усъвършенстват немалък брой методи, независимо от огромните усилия, необходими за създаването, валидирането и прилагането им.

10.4. Метод на функционалните точки

10.4.1. История и мотиви

Методът на функционалните точки е предложен през 1979 г. от Олбрихт [5]. След това самият той в съавторство [6], както и много други учени работят усилено и продължително върху усъвършенстването му, практическото му приложение, сравнението му с други модели и методи.

Основният мотив на автора са недостатъците на фактора брой редове първичен код. Както видяхме, той е основата на няколко модела, сред които най-представителен е СОСОМО. Като най-съществени негови недостатъци вече изтъкнахме трудната му ранна

определяемост и липсата на единно виждане за същността му. Поставяйки си за цел да намери по-добър фактор в това отношение (ясен и лесен за определяне в ранен стадий на производствения процес), Олбрихт достига до понятието „функционална точка“. На основата на този базов обект той изгражда модел и метод и по-късно ги подобрява. Главната му идея е, че усилията за разработването на даден софтуер (а следователно и цената му) се определят от неговата функционалност. Последната може да бъде измерена на основата на въпросните функционални точки. Доколкото функционалните точки (resp. техният брой и вид) могат да бъдат определени с помощта само на някои първоначални проектни документи от типа на съглашение за изискванията и външна спецификация, това означава и ранно получаване на желаните оценки.

Всъщност в цитираната вече [5] Олбрихт сам формулира в явен вид 5 цели, които се стреми да постигне със своето предложение за модел и метод:

- да се използват външните характеристики на софтуера;
- да се третират средства, важни за крайния потребител;
- да може да се прилага в ранна фаза на производствения процес;
- да може да се свърже лесно с икономически оценки;
- да има независимост от редовете първичен код.

10.4.2. Същност на модела

Моделът се основава на 5 функционални типа. Това са 5 непресичащи се класа от обекти с точно определени функции. Допълнително се определят по 3 нива на сложност — просто, средно, сложно — за всеки тип. Тези типове са следните:

Външен входен тип (External Input Type) — такъв е всеки потребителски входен управляващ поток или поток от данни, който пресича външната граница на измерваното приложение и който добавя или изменя данни в даден вътрешен логически тип. Този тип може да бъде:

- прост, когато съдържа малък брой типове данни и когато тези данни се отразяват върху малък брой вътрешни логически типове;
- междинен, когато не може да се определи еднозначно нито като прост, нито като сложен;
- сложен, когато съдържа значителен брой типове данни и когато тези данни се отразяват върху голям брой вътрешни логически типове; при това проектирането на външния входен тип изисква значителни допълнителни усилия, тъй като трябва да се вземат предвид проблемите на човеко-машинния интерфейс.

Типичен пример за този тип са входните екрани. Въвежданите чрез тях данни „навлизат“ в приложението и променят един или повече вътрешни логически типа.

Външен изходен тип (External Output Type) — такъв е всеки потребителски изходен управляващ поток или поток от данни, който напуска външната граница на измерваното приложение. Като примери могат да се посочат различни видове съобщения до потребителя или изходни отчети. Класифицирането на този тип на трите нива на сложност е аналогично на предходния тип. Допълнително за видовете отчети се ползват следните определения:

- прост — отчетът съдържа една или две колони, данните почти не се преобразуват при извеждането им;

- междинен — отчетът съдържа много колони, налични са междинни суми по нива;
- сложен — резултатите в отчета се получават след сложни преобразувания на данните, налага се комбиниране на данни от много и сложни файлове, налага се да се вземат мерки за постигане на по-добра ефективност на обработките.

Вътрешен логически файлов тип (Internal Logical File Type) — такава е всяка съществена група от потребителски данни или управляща информация. Типични примери са логическите файлове и въобще всяка логическа група от данни, обособена като такава поради потребителска представа и която се генерира, използва и поддържа от приложението. Класифицира се на три нива по следния начин:

- прост — малко типове записи, малък брой типове елементи от данни, не се изискват усилия за постигане на ефективност или за съхраняване и възстановяване;
- междинен — типът не може еднозначно да се определи като прост или сложен;
- сложен — голям брой типове записи и типове елементи от данни, изискват се специални грижи за осигуряване на по-висока ефективност, както и за процедури по запазване и възстановяване.

Външен интерфейсен файлов тип (External Interface File Type) — това е файл, който се предава или се ползва съвместно от две или повече приложения. За да се определят трите нива на сложност, се ползват дефинициите за вътрешния логически файлов тип.

Външен справочен тип (External Inquiry Type) — всяка комбинация вход/ изход, при която входът предизвиква незабавен изходен резултат. Такива комбинации се реализират например в дадена информационна система, когато потребителят формулира и въведе заявка и получи от приложението съответния отговор. Класификацията на трите нива се определя така:

- входната част се класифицира съгласно определенията за външния входен тип;
- изходната част се класифицира съгласно определенията за външния изходен тип;
- сложността на външния справочен тип е равна на по-голямата от така определените две сложности.

10.4.3. Процедура по пресмятането

Преброяват се елементите на всеки от упоменатите 5 функционални типа. След това за всеки елемент от всеки функционален тип се определя нивото на сложност съгласно дадените вече определения.

С така получените данни се попълва следната таблица (табл. 10.4.):

Тип	Наименование	прост	междинен	сложен	Общо
IT	Външен Входен	X3	X4	X6	
OT	Външен изходен	X4	X5	X7	
FT	Вътрешен логически файлов	X7	X10	X 15	
EI	Външен интерфейсен файлов	X5	X7	X 10	
QT	Външен справочен	X3	X4	X6	
FC	Общо ненастроени ф.точки				

Табл. 10.4. Таблица за пресмятането

Обозначението „X a" в трите колони за сложностите означава, че получената за тази клетка бройка трябва да се умножи по теглото „a". Точките в колоната „Общо" означават, че на това място следва да се попълни сумата от получените в предходните 3 колони числа.

Като се сумират 5-те числа от най-дясната колона, се получава общият брой *ненастроени функционални точки FC* за изследваното приложение.

Следващата стъпка хронологично е предложена по-късно. Тя е предизвикана от осъзната от авторите необходимост от съществени уточнения и до-настройки на така получения брой *FC*.

Разглеждат се изброените по-долу *14 характеристики* и за всяка от тях се определя степента ѝ на влияние върху оценяваното приложение. *Допустимите стойности* за всяка характеристика са от 0 до 5 и имат следния смисъл:

- 0 — не е налична или не влияе въобще
- 1 — незначително влияние
- 2 — умерено влияние
- 3 — средно влияние
- 4 — значително влияние
- 5 — силно влияние.

Самите *характеристики* са следните:

1. Данните и управляващата информация, използвани в приложението, се изпращат или получават по *комуникационни линии*.
2. В приложението има *разпределена обработка* на данни.
3. За приложението е важно достигането на *висока ефективност*.
4. Приложението ще се експлоатира върху силно *натоварена операционна конфигурация* — хардуер и софтуер.
5. *Интензивността на транзакциите* е висока.
6. Наличен е *интерактивен режим на въвеждане* на данни и на управляваща информация.
7. Цели се *ефективност* от гледище на *потребителя*.
8. Наличен е *интерактивен режим на актуализирането* на данните.

9. Логиката на обработките е *сложна* (многообразни логически и математически уравнения, необходимост от обработка на много изключения и др.).

10. Програмният код трябва да е *повторно използваем* (reusable).

11. Цели се *лесно инсталиране*.

12. Цели се *лесна експлоатация* (ефективна начална процедура, съхраняване, възстановяване и др.).

13. Приложението може да се ползва за *разнообразни потребители*.

14. Приложението е *гъвкаво* и лесно се модифицира.

Определя се величината *PC* (processing complexity) като *сума* от 14-те определени стойности. Очевидно тя е в интервала [0,70].

На нейна основа се определя коригиращият коефициент *PCA* (processing complexity adjustment) по формулата

$$\text{PCA} = 0.65 + (0.01 \times PC)$$

Ясно е, че PCA ще бъде в интервала [0.65, 1.35].

Крайният резултат — настроеният брой функционални точки *FP* (*Function Points Measure*), се получава по формулата:

$$FP = FC \times PCA$$

Лесно се вижда, че коригиращият коефициент PCA всъщност коригира първоначалния резултат с +/- 35%.

10.4.4. Използване на модела

При наличието на достатъчно статистически данни не е трудно да се определи цената на създаване на една функционална точка. Оттук лесно може да се пресметне и очакваната цена на разработване на даден софтуерен продукт.

Изследванията, свързани с функционалните точки, са довели и до други много интересни резултати, свързани, най-общо казано, с икономиката на софтуера.

Едно такова изследване според [7] показва какъв е приблизителният брой *функционални точки*, притежавани от определен тип организация в САЩ. Ясно, че е възможно да се прегледа с приемливо равнище на точност наличният в дадена фирма софтуер и за всеки продукт да се определи броят функционални точки. Ето каква картина се получава за различните групи (табл. Ю.5.):

По подобен начин са получени данни и какво количество функционални точки използват избрани групи от потребители в САЩ. Резултатите са дадени в Табл. 10.6.:

Както и авторът на [7], следва да отбележим изрично, че данните от тези 2 таблици могат да съдържат значителни отклонения. Основни причини са пионерският им характер и динамиката на някои от професиите (почти е сигурно например, че в областта на телекомуникациите консумацията на софтуер е нараснала значително през последните години). Независимо от това, вижда се обещаваща Методология за изследване на различни аспекти на икономиката на софтуера.

Тип организация (фирма)	Брой финкц. точки
Малка местна банка	125 000
Средна търговска банка	350 000
Голяма международна банка	450 000
Средна животозастрахователна компания	400 000
Голяма животозастрахователна компания	550 000
Голяма телефонна оперативна компания	450 000
Голяма телефонна производствена компания	600 000
Средна производствена компания	200 000
Голяма производствена компания	375 000
Голям производител на компютри	1 650 000

Табл. 10.5. Брой функционални точки по типове организации

Тип професия	Брой използвани функционални точки
Служител в туристическа агенция	35 000
Служител по резервациите в авиолиния	30 000
Аерокосмически инженер	25 000
Електроинженер	25 000
Телекомуникационен инженер	20 000
Банков служител по кредитите	15 000
Софтуерист	15 000
Машинен инженер	12 500
Ликвидатор 8 застрахователна компания	5 000

Табл. 10.6. Брой функционални точки по професии

10.5. Други модели

За да дадем представа за пътищата, по които са се развивали разглежданите тук модели, ще направим кратък преглед на още някои от тях.

10.5.1. Doty

Този модел наподобява COCOMO. Основава се на 2 формули:

$$\text{ЧМ} = 5.288 \times \text{ХРПК}^{1.047} \text{ за ХРГЖ} < 10$$

$$\text{ЧМ} = 2.060 \times \text{ХРПК}^{1.047} \times f_1 \times f_2 \times \dots \times f_{14} \text{ за ХРПК} \geq 10,$$

където обозначенията имат същия смисъл, както при COCOMO, а f_i са подобни атрибути, но само с по 2 възможни стойности.

10.5.2. SPQR

Отново е инспириран от **COCOMO**. Основава се на 20 добре дефинирани и 25 недостатъчно добре дефинирани фактора, определящи цената на разработване на софтуера и на производителността. Доведен е до търговски продукт, но

не е достатъчно добре документиран. От потребителя се изисква да отговори на 100 въпроса, свързани със софтуерния проект. На тяхна основа се установяват стойностите на входните параметри. Авторът на модела Кейпърс Джоунс твърди, че с 90% вероятност моделът дава оценка в рамките на точност от +/- 15%.

10.5.3. ESTIMACS

Този модел с автор Рубин е съсредоточен преди всичко върху разработването. Претенциите му са също за точност в границите на 15%. Включва следните модули:

1. Оценител на усилията за разработване на системата. Основава се на отговорите на 25 въпроса относно разработвания софтуер, средата на разработване и др. За пресмятанятията използва база от исторически данни за вече разработени проекти.
2. Оценител на разходите за колектива разработчик. Входни данни са получената по т. 1 оценка, данни за квалификацията и производителността на разработчиците, разбивка на заплатите по нива. И тук се ползват исторически данни.
3. Оценител на хардуерната конфигурация. На входа се подават данни за изискванията на софтуерния продукт към операционната среда, очаквания брой и тип на транзакциите, а като изход се получава оценка за необходимата хардуерна конфигурация.
4. Оценител на риска. На основата на отговори на 60 въпроса относно размера на проекта, структурата му и използваната технология се изчисляват характеристики на риска.

10.5.4. BANG

Авторът на този модел е една от най-известните фигури в областта на софтуерните технологии — Де Марко [8]. Методът е от типа на функционалните точки, но се знае, че е правен независимо от Олбрихт. Докато моделът и методът на функционалните точки на Олбрихт е насочен най-вече към информационни системи и бизнесприложения, целта на BANG е системният софтуер и този за научни приложения. Основните елементи, които се установяват и броят при този метод, са:

1. Функционални примитиви (функционална единица, която не може повече да бъде декомпозирана).
2. Модифицирани функционални примитиви.
3. Елементи от данни.
4. Входни елементи от данни. 5.-Изходни елементи от данни.
6. Елементи от данни в паметта.
7. Обекти (objects или entities).
8. Връзки (relationships),

както и няколко фактора, свързани със спецификата на системите в реално време, каквито са между другото и операционните системи (преходи, състояния на чакане).

Това значително по-широко множество от фактори дава основание този модел да се разглежда като надмножество над модела на Олбрихт. Естествено, тази характеристика прави метода по-трудно приложим, което се е отразило и на далече по-малкия му брой потребители.

10.6. Оценяване при обектна ориентираност

Оценяването на разходите и продължителността на разработване на софтуер, създаден с обектно ориентирани методи, изиска специално внимание. По този въпрос много показателна е [9], където се излагат резултати от сравнително изследване на софтуер, създаден с различни средства, някои от които обектно ориентирани. Интересна е приложената технология, доколкото данните за езика Ада9х се е наложило да бъдат симулирани, а за други езици — да бъдат получени чрез трансформация на основата на аналогични проекти. Самият софтуерен продукт е в областта на телекомуникациите

Показано е, че модели, свързани с броя първични редове код, не могат да бъдат прилагани. Едновременно с това обаче се демонстрира, че методът на функционалните точки е приемливо приложим. Поради това, че е трябвало да се сравняват обектно ориентирани с процедурни езици, не е било възможно прилагането на специално създадения за обектно ориентирани приложения [10] модел и метод *MOOSE* (Metric for object-oriented system environment). Основният извод от изследването е, че обектно ориентираните езици осигуряват по-висока производителност от процедурните. Към това трябва да се направи бележката, че изводът засяга преди всичко програмирането, интегрирането, тестването и управлението, в много по-малка степен проектирането и по никакъв начин анализа, формулирането на изискванията и документирането. Направени са изследвания и произтичащи от тях заключения, че при обектно ориентираното програмиране намирането на грешки в процеса на разработване става по-резултатно, отколкото при процедурните езици. Резултатите относно производителността се виждат от табл.10.7. Числата в клетките са в човекомесеци.

Език	Проект	Кодиране	Тестване	Управление
Асемблер	61,5	317,0	295,0	108,0
C	61,5	117,0	159,0	57,0
Паскал	61,5	51,0	97,0	39,0
Ада 83	61,5	35,0	82,0	35,0
Ада9х	61,5	21,0	67,0	30,0
C++	44,5	9,0	50,0	23,0

Табл. 10.7. Езици и производителност

Освен споменатия вече модел *MOOSE* известна е и една опростена версия на метода на функционалните точки, предложена в [11] (авторът Й Снийд е известен с това, че е ръководил създаването на една от първите големи CASE системи). В тази версия е елиминирана чисто функционалната част и се разглежда частта, която моделира структурните класове. Основната цел е както максимално доближаване до спецификата на обектната ориентираност, така изтегляне приложението на метода във възможно най-

ранен момент. Признава се, че точността при такъв подход силно страда, но се изхожда от презумпцията, че една ранна, донякъде неточна оценка е за предпочтитане пред една точна, но късна. Засега приложението на метода на Снайд е силно ограничено.

Известно е и едно предложение на Боем [12], основано на модела и метода COCOMO 2.0, наречено *обектни точки* (*object points*). Тези точки се генерират чрез пребояване на екрани, отчети, определен тип модули, после се сумират с подходящо определени тегла и получената сума се донастройва с оглед свойства за повторно използване. За съжаление и този метод трудно може да бъде приложен преди завършването на фазата на проектиране.

За стъпка в положителна посока се смята предложението в [13] модел. В него на основата на предварително създаден бизнесмодел (такъв може да се прави за бизнесприложения и особено се препоръчва за приложения, работещи в Интернет, свързани с електронна търговия) се прилага методът *System Meter*, за който се твърди, че дава правдоподобни оценки за трудоемкостта на разработването.

Литература

1. Boehm B.W., Software Engineering Economics, Prentice Hall, Englewood Cliffs, N.J., 1981.
2. Lederer A.L., J.Prasad, Software Management and cost estimating error, The Journal of Systems and Software, 50 (2000), p. 33-- 42.
3. Русеков А., Ресурсные модели для оценки софтверных расходов, Анализ-91, Книга-8, Интерпрограма, 1991.
4. Kemerer C, An empirical validation of software cost estimation models, Communication of the ACM 30(5), 416—429
5. Albrecht A.J., Measuring application development productivity, Proc. IBM Applications Development Symp., Monterey, CA, Oct. 14—17, 1979, p.83.
6. Albrecht A.J., J.E.Gaffney, Software Function. Source Lines of Code, and Development Effort Prediction: A Software Science validation, IEEE Trans. Softw.Eng. SE-9, 6(Nov. 1983), 639—648.
7. Jones C, Applied Software Measurement, McGraw-Hill, New York, 1997.
8. DeMarco T, Developing a Quantifiable Definition of Bang, Controlling Software Projects, Yourdon Press, New York, 1982, p. 92—110.
9. Jones C, The Economics of Object-Oriented Software, American Programmer, October 1994, p. 28—35.
10. Kemerer, C.F., MOOSE: Metrics for Object-Oriented Systems Environments, Proc. of ASM93 Conference, Orlando, FL, November 1993.
11. Sneid H., Calculating Software Costs using Data (Object) Points, SES, Ottobrunn, Germany. 12. Boehm B.W., B.Clark, E.Horowitz, Cost models for future life cycle process: COCOMO 2.0, Ann. Software Engineering, 1(1), p.1—24.
13. Moser S, B.Henderson-Sellers, VB.Misic, Cost estimation based on business models, The Journal of Systems and Software, 49(1999), p. 33—42.

11. МАРКЕТИНГ НА СОФТУЕРА

11.1. Общи съображения

Както става ясно от изложението дотук, софтуерът се разглежда винаги като продукт, който рано или късно трябва да се реализира на пазара. При това положение е невъзможно да се избегне маркетинговият аспект. Двете основни понятия, с които ще боравим в тази глава, са *софтуер* и *маркетинг*.

При такава постановка единият възможен аспект за изследване е *софтуерът, необходим за нуждите на маркетинга*. Това е тема, сама по себе си важна от практическа гледна точка. Не е трудно да се разбере, че в съвременните условия е немислимо да се осъществява маркетингова политика без съответни софтуерни средства. От най-общи съображения е почти очевидно, че за целта е необходима разнообразна информация, организирана по такъв начин, че да е достъпна за съответните специалисти и лица, вземащи решения. При това този достъп трябва да е достатъчно бърз и ефективен при запитвания, формулирани по разнообразен начин. Освен това са необходими разнообразни изходни документи за нуждите на анализа и отчетността. Непосредственият извод е, че не може да се мине без съответните технически средства и софтуер. Общото понятие, което се използва в тези случаи, е *маркетингова информационна система*. В последните години тя не може да бъде друга освен компютризирана. Тази тема обаче излиза от обсега на настоящите разглеждания и затова повече няма да се спирате на нея.

Остава за разглеждане вторият аспект — има ли специфика в маркетинга на софтуера и каква е тя [1]. Отговорът на този въпрос следва оттук нататък до края на тази глава, като се дават общите понятия и принципи на маркетинга, илюстрират се преди всичко с примери от областта на софтуера (и по-рядко на хардуера) и се очертава спецификата на маркетинговия софтуер, където я има.

11.2. Определения за маркетинг

Търде много определения за *маркетинг* могат да се намерят в литературата. Това се обяснява с няколко фактора — сравнително краткия период на развитие на теорията на тази област, невъзможността (поне в настоящия мо-

мент и обозримото бъдеще) за формализиран подход, решаващото влияние на динамичната практика на маркетинга върху теорията и др. За илюстрация на разнообразието на подходите ще посочим малък брой от известните дефиниции [2].

— Маркетингът се състои от всички дейности, чрез които една компания се приспособява към своята среда — творчески и изгодно. (Рей Кори)

— Маркетингът е практическа дейност, система от управленски функции, чрез която се организира и ръководи комплекс от дейности, свързани с оценката на покупателната способност на потребителите, нейното превръщане в реално търсене на изделия и услуги и придвижване на тези изделия и услуги към купувача за постигане на печалба или никаква друга цел. (Английски институт по маркетинг)

— Маркетингът е социален и управленски процес, в който отделните личности и групи получават това, от което се нуждаят и желаят, като създават, предлагат и разменят с другите хора продукти с определена стойност. (Филип Котлър [3])

За нуждите на нашето разглеждане най-подходящо е да разглеждаме маркетинга като съвкупност от научни, организационни и технически средства за изследване на състоянието и въздействие върху развитието на пазара.

11.3. Основна маркетингова концепция

Отново както и при проблема за качеството, във фокуса е потребителят. Той е основополагащият елемент във философията на маркетинга. Формулирана в едно изречение и във възможно най-кратка форма, тази философия, наречена още *основна маркетингова концепция*, гласи: *задоволяване нуждите на потребителя при печалба*. Разбира се, тази формулировка може да се модифицира, без това да доведе до съществено смислово изменение. В [3] например тя е, че *постигането на фирмениците цели зависи от определянето на потребностите и желанията на целевите пазари и от задоволяването на клиентите по начин, който е по-рентабилен и ефективен от този на конкурентите*.

Не трябва да се забравя, че в областта на производството на софтуер (както и в някои други области, свързани предимно с научни изследвания и въобще с перспективни, но рискови и евентуално нерентабилни начинания) съществуват организации, чиято приоритетна цел не е непременно печалбата, нито предимство над конкурентите по отношение на рентабилността или ефективността. Типичен пример са организации с идеална цел или държавни институции, чиято основна цел е да разработят продукт с определени авангардни качества. Последният, изнесен на пазара, се очаква да повлияе на характеристиките на продуктите, разработвани от фирмите и други подобни организации. Добър пример в това отношение са някои от първите компилатори, разработени в чужбина преди около четири десетилетия (Фортран, Алгол). Те не са били обект на търговска разработка, финансирана са били без оглед на непосредствената печалба, но са оказали огромно влияние върху качеството на разработените по-късно с тяхна помощ софтуерни продукти. Такава практика продължава да съществува и сега, макар че финансиращите институции все повече държат резултатният продукт от тази „непечеливша“ схема на разработване да бъде лесно усвоим от практиката и да окаже върху нея положително въздействие. Вероятно най-добрият пример за такива финансиращи организации в областта на софтуера в България през 90-те години са двата фонда към Министерството на образованието и науката — за научни изследвания и за структурна и технологична политика, респективно комисиите им по информатика.

11.4. Маркетингова стратегия

Всяка организация трябва да има своя маркетингова стратегия. Последната се състои от две стъпки:

- определяне на *целевия пазар*
- разработване на *маркетингов микс*

11.4.1. Целеви пазар

Целевият пазар се определя като съвкупност от купувачи, притежаващи общи потребности или характеристики, които организацията решава да обслужва.

За да определи своя целеви пазар, всяка организация трябва да го изследва с оглед на своите технологични, финансови, кадрови и други възможности. Постигането на тази цел става най-удобно чрез прилагането на техниката на *сегментиране на пазара*, което означава разделяне на пазара на хомогенни под-пазари, т. е. на групи от подобни клиенти. Сегментирането може да се извърши по различни признаки. Когато става въпрос за софтуер, тези признания са следните.

Индивидуален/индустриален. Първоначално пазарът е бил изключително от тип индустриски или корпоративен. Практически не е имало индивидуални потребители. Всички софтуерни продукти са били разработвани за компютри, притежавани и използвани за нуждите на организации (банки, заводи, търговски фирми, армия и пр.). С възникването на персоналните компютри през 80-те години и с изключително интензивното нарастване на броя им, броят на индивидуалните притежатели на компютри, а с това и на софтуер за тях, също се увеличава неимоверно. Поради тази причина в момента се налага сегментиране на пазара по този признак. Разбира се, има софтуерни продукти, които се използват еднакво и в организациите, и от индивидуалните потребители — текстови редактори, електронни таблици, електронни калкулатори, средства за компютърна презентация и др. Има обаче твърде много, които са предназначени точно за единия от тези два типа. Софтуерът за управление на производствени процеси, за резервация на самолетни или железопътни билети, за банково обслужване, за проектиране на големи обекти и т. н. се използва единствено от корпоративен тип потребители. От друга страна пък, програмни продукти за лично счетоводство, за попълване на лични данъчни декларации, компютърни игри и т. н. се ползват изключително от индивидуални потребители. Трябва обаче да се отбележи, че дори за софтуер, използван от двата типа потребители, сегментиране по този признак е възможно, защото организацията производи-тел-разпространител може да реши да се занимава само с единия тип.

Възраст. Производителите на компютърни игри очевидно се насочват към детската и юношеска публика. Определянето на целеви пазар по този начин води по-нататък до съобразяването на цялата маркетингова политика с неговите особености, което впрочем е вярно и за всеки друг определен целеви пазар. С развитието на информационното общество се очаква сред потребителите на софтуер все по-често да се оказват лица от всяка възрастова група и вероятно в много случаи този признак ще се окаже от значение за определянето на целеви пазар.

Професия. Този признак е твърде очевиден. Ясно е, че адвокатите се нуждаят от специфични за работата си информационни системи, архитектите и инженерите — от софтуер за нуждите на проектирането в дадената област. Списъкът може лесно да се продължи с лекари, фармацевти, програмисти, търговци на дребно, търговци на едро и т. н. Определянето на целеви пазар по този признак може да стане твърде отчетливо.

Образование. По принцип доскоро потребителите на софтуер бяха хора с високо и специализирано образование. С масовизирането на компютрите и разширяването на потребителската аудитория равнището на образование започва все повече да губи значението си и следователно сегментирането по този признак започва да става от значение.

География. Определени типове софтуер се влияят от това, дали са за местно или международно ползване. Най-типични в това отношение са програмните продукти, свързани с естествения език. Доскоро текстовите редактори силно зависеха от този признак, но като че ли напоследък най-разпространените от тях започват да стават все по-

независими благодарение на това, че допускат различни азбуки, съдържат средства за проверка на правописа за всеки език, за спазване на специфичните правила на пренасяне на нов ред т. н. Корсuerът (софтуер за нуждите на обучението) обаче продължава силно да зависи от местния език. От една страна, в много случаи той е предназначен за подрастващите и трябва изцяло да бъде на съответния език, от друга — езиково универсализиране както в случая на текстовите редактори не е възможно, тъй като за всеки предмет и за всяка тема се създава отделен обучаващ софтуер. Това впрочем е една от силните възможности на местните фирми за намиране на свой целеви пазар.

Tip организация. Това е сегментиране, което може да достигне до значителна дълбочина. Обикновено не е достатъчно да се определи типът организация потребител на най-високо ниво на общност — военна, научна, банкова, търговска. Интересите в областта на науката например са най-разнообразни и ако за някои типове програмни продукти дори такова генерално сегментиране е достатъчно (почти всяка научна организация се нуждае от софтуер за статистически пресмятания, широка е общността на тези, които използват софтуер за различни видове симулации), то има и специфични нужди на отделните науки, което налага по-дълбоко сегментиране на пазара по тях. Любопитно е, че самите производители на софтуер се нуждаят от твърде разнообразни програмни продукти. Във връзка с това има класически пример за несъобразяване с особеностите на пазара, респективно за това, че производителят е пропуснал да анализира сегментите и да определи своя целеви пазар. Става въпрос за разпространения навремето език за програмиране PL/1. Той е бил замислен и реализиран през 60-те години като език за всякакви нужди на всякакви потребители (програмисти и проектанти). Скоро обаче се оказалось, че политика, основана на такава философия, е невъзможна и само мощта, допълнителните лостове, амбициите и усилията на фирмата производителка успяват да задържат този софтуерен продукт на пазара относително по-дълго. Като изключим обаче един провален опит в средата на 80-те години за създаване на компилатор за PL/1 за персонални компютри, никой вече не говори за този език — единствено някои потребители на по-стари разработки, за които преходът към нещо по-модерно остава голям и скъп проблем. В противовес на това може да се посочи езикът Фортран, чийто пазарен сегмент е много по-ясно определен и това продължава да го държи на пазара. Към последния пример впрочем ще се върнем в друг аспект малко по-късно.

Примерът с PL/1 е поучителен в още едно отношение — щом за една организация от световен мащаб е необходимо съобразяване със сегментирането на пазара, толкова по-необходимо и направо задължително е за малка или нова организация да се насочи към точно определен сегмент от пазара.

Още две са важните понятия, свързани с целевия пазар. Първото е *пазарен прозорец* и означава пренебрегнати от производителите и търговците сегменти от пазара. Второто е *пазарна ниша*. Това е такъв сегмент, на който подхожда в най-голяма степен конкретният продукт или опит. Първият компонент на маркетинговата стратегия е определянето на целевия пазар. За конкретната фирма е толкова по-добре, ако някой от определените за даден програмен продукт сегменти от фиксирания целеви пазар се окаже пазарна ниша.

11.4.2. Маркетингов микс

Маркетингов микс се нарича съвкупност от четири управляеми маркетингови средства — продукт, цена, пласмент и промоция. Оригиналното английско название — *Marketing mix* — понякога се превежда на български и като *маркетингов комплекс*. В английски език понякога се нарича „four p-s“ — от първите букви на английските названия на казаните току-що четири средства: product, price, place, promotion.

За да поясним какво точно се има предвид под тези четири средства (наричани понякога и променливи) и доколко те подлежат на управление, да вземем следния пример. Да допуснем, че фирмата X е решила да атакува пазарния сегмент на средните училища, като се насочва към потребителите на персонални компютри. Този пример впрочем от 1998 година става особено актуален и реалистичен, като се вземат предвид предприетите активни мерки от държавата за въвеждане на информационните и комуникационни технологии в средното училище.

Какъв *продукт* би могла да предложи на пазара фирмата X? Една възможност е това да е интегриран програмен пакет, който да съдържа най-необходимите за ученика средства. Разумно би било в този софтуерен продукт да се включи текстообработваща компонента, значително по-проста от известните и разпространени на пазара, непременно на български език, с включени средства за проверка на правописа (spell-checker) и за пренасяне на нов ред. Друга необходима компонента е калкулаторът. Вероятно подходящо е включването на несложен файлов процесор, който да позволява създаването и използването на

прости информационни системи. Необходимост за ученика би била и компонента справочник за математически формули, а защо не и с възможности за символни пресмятания. Би могла да се включи и компютърна игра — не прекалено интересна, за да не му отнема твърде много от времето, но все пак достатъчно привлекателна, за да му служи за отмора за кратки периоди от време. Едва ли си струва да се включват средства за комуникация (връзка с Интернет), поради безусловната популярност на стандартните. Оформяйки по този начин съдържанието на продукта, фирмата X следва да извърши за него в съответствие с изискванията на софтуерните технологии анализ за осъществимост в различните му аспекти. Този анализ между другото следва да установи и кои от фиксираните компоненти ще се разработват от самата фирма, кои евентуално ще бъдат поръчани за разработка навън и кои следва да се вземат наготово след съответните процедури за придобиване на права за това.

Цената е следващата променлива, чиято стойност следва да се определи. В случая, като се изхожда от това, че се отнася преди всичко за индивидуални потребители с неголеми финансови възможност, цената ще трябва да се фиксира на възможно най-ниско ниво. Това е валидно и в случай, че се хвърлят усилия да се продава продуктът и на училища — общо взето, тези организации не разполагат с особени финансови възможности, разчитат преди всичко на бюджета (и много по-рядко на други източници — дарения, спонсорства, собствени приходи) и вероятно винаги ще изискват значителни отстъпки в цената, може би все пак за сметка на закупуване на повече от един екземпляр от продукта. Несъмнено трябва да се вземе предвид, че продуктът ще се продава в България и следователно следва да е съобразен с мащаба на цените на софтуерните продукти в страната.

По принцип за определянето на цената са известни *две крайни стратегии: пробив (penetration) и обиране на каймака (skimming-the-cream)*.

При стратегията на *пробива* продуктът се лансира с ниска цена, за да се запази конкурентоспособност на пазара и заедно с това да се привлекат максимален брой клиенти.

При стратегията на *обиране на каймака* фирмата започва с висока цена като визира тези от потенциалните клиенти, които имат вкус към новото и са в състояние да платят за него. По-нататък цената постепенно се понижава, за да обхване все по-широки кръгове от потребители.

Трудно е да се каже каква стратегия предпочитат софтуерните фирми. Дълго време пример за стратегията на пробива даваше фирмата „Борланд“. До началото на 90-те години, когато все още беше конкурент на „Майкрософт“, тя продаваше всички свои

програмни продукти на цена 99 щатски долара. За тази цена е имало и едно основание, специфично за софтуера. Фирмата е била преценела, че при такава цена повечето потребители биха предпочели да я платят, вместо да направят известен разход за копиране и след това да носят увеличаващия се риск от използването на незаконен софтуер. Не се наемаме да кажем кои са били по-късно причините за тежките проблеми и метаморфози на тази фирма, но е факт, че от един момент продуктите ѝ започнаха или да бъдат разбивани на твърде дребни, отделно продавани компоненти, или просто да се продават на много по-високи цени.

Друга възможност при определянето на цената е съобразяването със сегментите от пазара, където се лансира продуктът. Има сегменти, които са по-малко чувствителни към цената, и потребителите в тях могат да приемат да платят повече. Ясно е, че в областта на софтуера такава сегментация има място и ние я отбележахме вече (индустриален/индивидуален потребител). Понякога може да се окаже, че е необходима обосновка за подобна диференциация на цените и това обикновено се прави, като се сочат примерно повече гаранции, някои допълнителни „профессионални“ възможности, по-голяма ефективност на продукта и пр. Впрочем софтуерните фирми биха могли да вземат пример в това отношение от авиокомпаниите, които имат голямо множество от тарифи в зависимост от сегментите на пазара, които обслужват.

Пласментът е свързан преди всичко с местата на пласиране на продукта и начините, по които това става. За програмен продукт, предназначен за ученици, едно подходящо място за разпространение са ученическите книжарници. Друг тип подходящи магазини са тези за музикални касети и дискети. Естествено, не трябва да се забравят и стандартните места, където се знае, че такъв тип продукти могат да се купят — за София например това са някои от книжарските щандове на площад „Славейков“ и, разбира се, неголемият брой специализирани магазини за продажба на хардуер и софтуер. Макар и сега да прохожда, разпространението чрез Интернет добива все по-голяма популярност и в никакъв случай не трябва да се пренебрегва, особено за този тип потенциални купувачи — учениците, за които световната мрежа е изключително популярна.

Промоцията се разглежда като специфичен комплекс от реклама, персонални продажби, настърчаване на продажбите и връзки с обществеността, които фирмата използва за достигане на своите маркетингови цели. В дадения примерен случай подходящо място за пускане на реклами за продукта са например предаванията на специализираните музикални телевизии, за които се знае, че се следят от ученическата публика. Възможно е също продуктът да бъде обявен като награда в някои от състезанията по тези телевизии или пък в насочени към ученическата аудитория радиопредавания. Практикувана форма са промоцио-налните продажби в определени магазини, когато по преценка на фирмата може да се обявят промоционални ниски цени.

11.5. Пазарен жизнен цикъл

Въпросът за жизнения цикъл може да се разглежда в два аспекта — жизнен цикъл на разработването [4] и жизнен цикъл на готовия продукт. Макар тези аспекти да са свързани, удобно е с цел съсредоточаване върху определени характеристики те да бъдат разделени.

11.5.1. Пазарен жизнен цикъл на разработването на продукта

Доколкото общият предмет на разглеждането ни е софтуерът, интерес представлява едно успоредно разглеждане на пазарния жизнен цикъл на разработване и някой от

популярните модели на жизнения цикъл на програмния продукт [5] — например този на Гънтьр. Сравнението ще бъде по-лесно и по-обозримо, защото така избраният модел на Гънтьр може да бъде разглеждан в хронологичната му част, а пазарният жизнен цикъл е от същия тип. Припомняме, че Гънтьр отличава във времето шест фази: *изследване, анализ за осъществимост, про-*

ектиране, програмиране, оценка и използване. В пазарния жизнен цикъл пър-вите две фази са силно разтегнати, разчленени и конкретизирани за сметка на същинското разработване. Всички фази на пазарния жизнен цикъл са следните:

— *Възникване на идеята.* Най-същественото от маркетингова гледна точка е, че всяка идея за нов програмен продукт трябва да е предизвикана от потребителски нужди, а не от интереси на проектанти или програмисти.

— *Анализ за осъществимост.* Може да се приеме, че има пълно съвпадение на пазарния и на производствения модел в тази фаза. Тя се занимава с икономическата и технологичната осъществимост на програмния продукт.

— *Развитие и тестване на идеята.* От производствения модел се знае, че т. нар. външен проект (който донякъде съвпада с приетото у нас задание за разработка) се прави с участието на потребителя. От пазарна гледна точка особено се наблюга на задълбоченото изучаване на отношението на потенциалните потребители към начина и степента на задоволяване на техните нужди от планирания програмен продукт, към цената му, към аналогични, достъпни на пазара програмни продукти. Анализът на така получените данни силно подпомага изграждането на маркетинговата стратегия.

— *Търговски анализ.* Това е съвсем специфична фаза, която по време се разпростира върху няколко други фази. Нейната задача е възможно най-точното определяне на абсолютните стойности на разходите и приходите.

— *Разработване.* В тази фаза са съчетани всички дейности по същинското разработване на програмния продукт, т. е. останалата след фазата по развитие и тестване на идеята част от проектирането, програмирането и оценката, съгласно производствения модел.

— *Фиксиране на маркетинговата стратегия.* То се извършва при готов програмен продукт и по-точно определят се целевият пазар и маркетинговият комплекс.

— *Тестване на пазара.* Това е също специфична фаза. В известен смисъл тя прилича на прототипирането на програмния продукт в производствените модели. Става въпрос за проверка на маркетинговата стратегия чрез ограничено лансиране на новия програмен продукт на подбрани представителни части от пазара. В тази фаза изprobването на варианти все още е много по-евтино, отколкото в следващата фаза.

— *Комерсиализация.* Това е завършващата фаза на пазарния жизнен цикъл. Чрез нея програмният продукт навлиза напълно в целевия пазар при осъществяване на всички компоненти на маркетинговия комплекс.

11.5.2. Пазарен жизнен цикъл на готовия продукт

Отличават се четири фази в жизнения цикъл на готовия продукт (в скоби Даваме оригиналните английски названия).

- въвеждане (introduction)
- растеж (growth)
- зрелост (maturity)

— спад (decline)

Удобен пример за илюстрация е програмният продукт компилаторът от Фортран. Макар и донякъде условно, във времето нещата изглеждат така:

- въвеждане — 1956 година
- растеж — докъм края на 60-те години
- зрелост — 70-те и 80-те години

— спад — може би 90-те години, макар да не се наемаме да твърдим това с абсолютна сигурност, доколкото все още остават немалък брой потребители на Фортран, разработването на нови версии на езика продължава и съответни компилатори се пускат периодично на пазара.

В днешно време жизненият цикъл на софтуерните продукти се съкраща поради бързия технологичен напредък. Стремежът на всяка фирма е да *удължи* по възможност живота на своите продукти поради очевидни финансови причини. За целта съществуват различни възможности. Една от тях е да се обявят нови свойства на съществуващ продукт. Класически пример са езиците за символни преобразувания или за симулиране, създадени като такива. По-късно, когато терминът изкуствен интелект се изпълни със съдържание и най-важното стана модерен и търсен, същите тези езици бяха обявени за езици на изкуствения интелект. Има и други начини, но те изискват значителни усилия при проектирането и програмирането на софтуерния продукт с оглед осигуряването на по-голяма от обичайната гъвкавост. Така например един от езиците в популярната система за управление на бази данни за персонални компютри ACCESS е BASIC ACCESS. В първоначалния си вариант, т. е. във версия 1.0 и 1.1 на ACCESS той се отличаваше малко от добре известния BASIC и само се загатваше за възможности за обектна ориентираност. Постепенно, минавайки през версии 2.0, 7.0, 97, 2000, BASIC ACCESS започна да придобива далеч по развити обектно ориентирани свойства. Това просто се оказа отдалече разработена гъвкава концепция за паралелно развитие на няколко от базовите езици на „Майк-рософт“. Разбира се, тук стои и другият въпрос — доколко част от потребителите биха се съгласили с така обявяваните и въвеждани нови свойства на продукта, с който вече са свикнали да работят.

11.6. Позициониране и марка

Сред проблемите на маркетинга от особено значение са следните два — **позиционирането** на програмния продукт и **марката**.

Позиционирането е създаване на силен и индивидуален образ на програмния продукт и внушаване на този образ на потребителите. Начините за постигането на това са систематизирани.

Единият от тях е **сравняване с познати образци**. Ако отворите кое да е американско специализирано списание, ще видите например, как за всяка по-нова система за управление на бази данни се изтъква, че примерно сортира k пъти по-бързо от FoxPro (или ACCESS) или че допуска повече полета за запис от нея и т. н. Допреди няколко години на мястото на FoxPro (ACCESS) обикновено стоеше dBase III. Докато такъв начин на анонсиране и рекламиране е позволен от закона в САЩ, в много страни не е разрешен или се допуска при значителни ограничения. У нас по тези въпроси няма категорична уредба и затова нещата се решават на морално равнище.

Втори възможен начин е **обявяването** на програмния продукт за **специфична категория потребители**. Не е рядкост някой от известните у нас програмни продукти за счетоводство да се рекламира примерно така: „Вече няколко хиляди счетоводители от

счетоводните колективи в над 1000 фирми в страната са освободени от рутинните трудопогълщащи операции и работят в условията на компютъризирано счетоводство с помощта на..." Една модификация на този начин е подчертаването на *ползата* от експлоатацията на програмния продукт. Например: „Всяка грешка във ведомостта за заплати става невъзможна, ако внедрите...“ И още един начин: обявяване на специфични възможности: „Това е единственият програмен продукт за текстообработка на пазара, който има вградени средства за проверка на правописа и за пренасяне на думите на нов ред...“.

Марката е наименованието, под което програмният продукт навлиза на пазара. Има три вида марки: *индивидуално име, общо име и комбинация*.

- Clipper е типичен пример за първия случай.
- ФИКС-ЕС, ФИКС-VAX, ФИКС-РС илюстрира втория случай, когато продуктът е един и само се отличават вариантите за различни среди (компютри, операционни системи).
- Третият случай се илюстрира най-добре от продуктите на „Майкрософт“, повечето от които доскоро започваха с Microsoft, а сега — с MS: MS Word, MS Access, MS Visual C++ и т. н.

Лесно могат да се видят предимствата и недостатъците на трите подхода. В третия случай названието на фирмата е протекция и гаранция, но един неуспешен продукт може да навреди на всички останали от същата фирма. Във втория случай положението е аналогично за съответната гама. Първият е противоположност на разгледаните два — провалът му по-трудно ще се отрази отрицателно на останалите продукти на същата фирма, но пък по-малко би ползвал евентуално вече създаденото добро име на фирмата.

Тук е невъзможно да навлезем в правните проблеми, до които може да доведе необмисленият избор на търговска марка. Ще подчертаем само, че следва да се вземат предвид доста ограничения. Ще изброим обаче някои най-общи условия, чието спазване е в интерес на производителя и търговеца — марката трябва да е **запомняща се**, да **внушава някакъв образ**, да **напомня за възможностите** на програмния продукт. Например Windows като че ли удовлетворява и трите условия — лесно се запомня, говори за силно развитата прозоречна техника, а на по-абстрактно ниво внушава образа за отвореност и широк поглед към проблемите.

11.7. Моделиране на софтуерния пазар

Специален интерес представлява моделирането на софтуерния пазар. Само по себе си то е интересно от теоретическа гледна точка, но е изненадващо колко полезни практически резултати биха могли да се извлекат от един достатъчно адекватен модел, стига да е основан на достоверни входни данни.

Първите и като че ли най-успешни опити за моделиране на софтуерния пазар са разразени в няколко статии с основен съавтор и идеолог Суон (напр. [6]). У нас този модел е бил усъвършенстван в определени насоки [7].

Моделът на Суон се основава на някои познати икономически модели, но в **Него** са въведени съществени допълнения, важни за отразяване особеностите на софтуерния пазар. Разглеждат се два аспекта на вътрешното качество на програмния продукт — *хоризонтален и вертикален*. Към *хоризонталния* аспект се отнасят тези характеристики на качеството на продукта, за които няма общо съгласие на потребителите, а към *вертикалния* — тези, за които всички потребители са на единно мнение. Освен това в модела се отчита и редица от *външни обстоятелства* (т. нар. *среда*), свързани с

продукта и от значение за предпочтенията на потребителя — брой на (другите) потребители, допълнителни възможности („add-on-s“), възможности за връзка с други продукти, фактори, намаляващи риска от това продуктът бързо да остане.

В модела се приема, че интересът на потребителя i се описва с функцията

$$U(i) = a(i) + b(i) * m(j) * N(j) + c(i) * Q(j), \quad (11.1)$$

където $Q(j)$ представлява вътрешното качество на продукта j , $N(j)$ е броят на инсталираните копия от продукта j , $m(j)$ е коефициент, усилващ значението на $N(j)$, $a(i)$, $b(i)$, $c(i)$ са параметри, отразяващи предпочтенията на потребител i . Ще отбележим, че формулата е дадена вече във вид, в който параметрите за хоризонталното и за верикалното качество са обединени в члена $c(i) * Q(j)$ след добре дефинирана за целта процедура. Възможно е и известно нормализиране на тази формула.

Предполага се, че всеки нов потребител i избира продукт j така, че да мак-симизира $U(i)$. Отделно от това се прави и предположение (изведеното от предшестващи емпирични изследвания), че потребителите се появяват в постоянен поток, дефиниран чрез т. нар. S-крива:

$$(11.2) \quad n(t) = n_{\max} \cdot \frac{1}{1 + \exp(-g(t - t^*))}$$

или след преобразуване:

$$(11.3) \quad n(t) = n(1) \cdot \frac{1 + \exp(-g(t - t^*))}{1 + \exp(-g(t - t^*))}$$

където $n(t)$ е броят на новите потребители в момент t (или сумата на продадените на тях моделирани програмни продукти), $n(1)$ е същото в началния момент 1, g е функцията на нарастващо, t^* е инфлексната точка на кривата S, n_{\max} — нивото на насищане с нови потребители.

Да илюстрираме използването на модела с един пример. Да предположим, че

$$n(1) = 1, t^* = 3 \text{ и } n_{\max} = 5.$$

Броят на продуктите нека е 3 с дати на въвеждане на пазара 1, 3 и 5 и със съответно вътрешно качество $Q(1)=1$, $Q(2)=2$, $Q(3)=3$.

Предполагаме още 15 потребителски типа от $c(1)=0$ до $c(15)=5.7$.

Като изход от модела се получава обемът на продажбите за всеки продукт j в даден период t , както и общият брой на продажбите. Общият обем на продажбите зависи само от n_{\max} , t^* и $n(1)$. Обемът на продажбите за отделния продукт j се определя динамично от неговото вътрешно качество $Q(j)$ и от средата $N(j)$, която е имал в предходния период. Това става по следния начин:

- Пресмята се за всеки потребителски тип с функцията на предпочтание за всички продукти j .
- Намира се за всеки от типовете, кой продукт j има максимално $U(j)$.
- Преброява се за всеки продукт, от колко потребителски типа е бил предпочтен.
- Този брой, отнесен към общия брой потребителски типове (в случая 15), дава каква част от общия обем на продажбите за периода t е получил продуктът j .
- Средата $N(j)$ за всеки продукт се увеличава с броя на продажбите, направени за този период.
- Преминава се към пресмятання за следващия период.

Получените по този начин и при тези входни данни резултати са дадени в таблица 11.1.

период	общ обем продажби	продукт 1	продукт 2	продукт 3
1	1,00	1,00	0,00	0,00
2	1,67	1,67	0,00	0,00
3	2,50	1,17	1,33	0,00
4	3,33	1,56	1,78	0,00
5	4,00	1,60	1,33	1,07
6	4,44	2,07	1,48	0,89
7	4,71	2,51	1,88	0,31
8	4,85	3,23	1,62	0,00
9	4,92	4,59	0,33	0,00
10	4,96	4,96	0,00	0,00

Табл. 11.1. Входни данни за модела

Получената ситуация е твърде интересна. Продукт 1 се появява първи на пазара и изгражда около себе си среда. В началото на период 3 се появява продукт 2, с по-добро вътрешно качество и веднага (поради тази причина) завзема по-голям дял от пазара, като създава очакване, че този процес ще продължи. Така би и станало, но всъщност тенденцията трае до период 5, когато на пазара се появява продукт 3, с още по-добро качество. С тази си характеристика продукт 3 започва да вреди на продукт 2, защото всъщност те двата започват да се конкурират и не си позволяват един на друг да увеличат средата си (т. е. броя на потребителите си). Крайният резултат се вижда в реда за период 10, когато на пазара е останал само продукт 1.

Така описан, моделът е всъщност в най-простия си вариант. Допълнително в него могат да се въведат други фактори, реално влияещи на пазара:

— Предварително обявяване на продукта. Ако потребителите имат доверие на производителя, обявяващ своя продукт, те вече имат поле за избор — могат да купят такъв продукт, който вече е на пазара, но могат и да изчакат известно време до появата на обявения и тогава да направят преценка за покупка. Не е трудно да се въведе в (11.1) параметър, отразяващ тази особеност, например да се раздели $U(I)$ на $(1+g)^t$, проблемът е да се определи адекватна стойност на g , за t е ясно, че е периодът на изчакване от момента на обявата до Момента на появата на продукта на пазара. По-подробен анализ показва, че g следва да се дефинира като функция на няколко фактора — интензивността на рекламната кампания по предварителното обявяване, имиджа на производителя, неговата способност да удържи обещанията си за датата на пускане на про-Дукта и за неговото качество.

— Появата на абсолютно нов продукт следва да се отличава от появата на нова версия на вече известен продукт. В този случай би мото например да се предположи, че продукт — нова версия — автоматично взема част от потребителите на продукта — предшестваща версия. Трудно е, разбира се, да се прецени каква е тази част.

— Преценката на потребителите за качеството на продуктите може да се променя във времето (най-малкото поради промяната на цената, която се разглежда като компонент на качеството и която производителят променя от маркетингови

съображения). Това означава, че $Q(j)$ не е константа спрямо времето в (11.1) и това би могло да се отрази в модела.

Литература

1. Friedman H. H., L.W.Friedman, Marketing Methods for Software. The J. of Systems and Software, 7(1987), p. 207—212.
2. Благоев, В. Маркетингът в определения и примери. Изд. „П. Берон“, София, 1989.
3. Котлър, Ф. Управление на маркетинга. Том I и II. Изд. Графема, София, 1996.
4. Василев Д., Е. Мичева и др. Маркетинг. Теория и практика. София, 1981.
5. Ескенази А. Програмно осигуряване и маркетинг АИТАС, 11/1990, с. 33—36.
6. Swan P., H. Lamaison, Vertical Product Differentiation, Network Externalities and market Defined Standards: Simulation of the PC Spreadsheet Software Market. CRICT, Brunnel University, 1990, CRICT Discussion paper.
7. Eskenazi A., R.Rashev, Simulating the Software Market. In Proc. of the ACMBUL 93 International Conference of Information technologies, Varna, 1993, p.1—1, 1—7.

12. АВТОМАТИЗАЦИЯ НА СОФТУЕРНОТО ПРОИЗВОДСТВО

Същността на софтуерния парадокс е, че разработчиците на софтуер, които се занимават с автоматизиране на работата на другите, все още не са направили достатъчно за автоматизиране на собствената си работа. Статистически данни показват, че разходите за обработка на информацията се удвояват на всеки 5 години, а производителността на програмисткия труд се удвоява на всеки 25 години. Това обуславя актуалността на проблема за автоматизация на софтуерното производство (АСП).

Предназначенето на автоматизиращите средства е да поддържат избраните методи за разработване на софтуер, да улесняват управлението на проектите и на цялостния процес на създаване на ПП, да извършват трансформации от едно представяне на софтуерните продукти в друго и да проверяват правилността им. В зависимост от прилаганите автоматизиращи средства могат да се разграничат два основни подхода — използване на индивидуални средства и използване на интегрирани среди. Ще се спрем на основните характеристики и особености на всеки от тези два подхода.

12.1. Автоматизация чрез индивидуални средства

Този подход е исторически първият и се състои в използването на отделни (stand-alone) „полезни“ програми, улесняващи извършването на някаква дейност при създаването на софтуера. Първоначално средствата са били компилатори, асемблери, свързващи редактори, дебъгери, осигуряващи директна помощ за програмирането. Едва в началото на 70-те години се появяват средства, подпомагащи и други дейности.

Съществуващите индивидуални средства за АСП могат да бъдат класифицирани по различни критерии — по функциите им (т. е. кои дейности подпомагат), по използването им в различни фази от жизнения цикъл, по основните им потребители (менеджъри, програмисти, отговорници по осигуряване на качеството и др.), по степента им на сложност на усвояване и прилагане и др. В зависимост от автоматизираните дейности те се разделят на следните групи:

- а) средства за програмиране — езиковоориентирани редактори, трансла-тори, свързващи редактори, дебъгери, средства за управляемо изпълнение на програми и др.
- б) средства за тестване — анализатори на програми, генератори на тестови данни, средства за управление на тестването
- в) средства, подпомагащи управлението на проекти. Чрез тези средства мениджърът на софтуерната разработка може:
 - да оценява предварително трудоемкостта, стойността и продължителността на проекта и броя на хората, необходими за реализацията му. Оценката се прави чрез въвеждане на индиректна мярка за размера на проекта и анализиране на някои общи характеристики — сложност на проблема, опит на разработчиците в тази приложна област, зрелост на процеса на разработване и др.;
 - да определя основните задачи и да създава графики за изпълнението им;
 - да проследява развитието на проекта и при необходимост да го препла-нира с промяна на ресурсите и сроковете;
 - да оценява производителността на труда и качеството на създавания продукт чрез прилагане на подходящи метрики;

— да проследява удовлетворяването на изискванията.

г) средства, подпомагащи документирането

Те осъществяват създаването, оформлянето и отпечатването на документи. В тази група са текстообработващите програми с възможности за въвеждане, редактиране, проверка на граматическата правилност и стил на текст. Графичните редактори позволяват илюстрирането на текста с диаграми, схеми и произволни изображения. Издателските системи реализират форматирането (предпечатната подготовка) на оформлените в съответствие с определени стандарти документи.

д) средства, подпомагащи съпровождането:

— за управление на софтуерните конфигурации — идентифициране, контрол на версийте и управление на внасянето на изменения;

— статични и динамични средства за възстановяване на детайлния проект по изходните текстове на софтуерната система с цел повторно разработване или промяна, за да се подобрят някои характеристики на софтуерната система (reverse engineering и re-engineering);

е) средства за анализ и проектиране

Те позволяват създаването и оценяването на модел на софтуерната система, която ще се разработва. Могат да поддържат различни методи на проектиране — структурни или обектно ориентирани.

ж) средства за прототипиране и симулиране

Тези средства представлят някои функции или характеристики на поведението на софтуерни системи, работещи в реално време.

з) средства за проектиране и разработване на потребителския интерфейс; и) езикови процесори.

Тези средства са за използване на различни езици — за специфициране, за описание на проекти, за автоматично генериране на текста на програмите и т. н.

Изследвания за използване на индивидуални средства за АСП показват, че разпределението им по различните фази и функции от жизнения цикъл не е равномерно и че успехът на нови методи в софтуерното производство зависи в голяма степен от това, дали тези методи се подпомагат и от съответни автоматизирани средства [1].

Основно предимство на подхода за АСП с индивидуални средства е, че те не са толкова скъпи и всяка софтуерна фирма може да си ги позволи. Освен това тези средства улесняват една или няколко дейности и повишават производителността на персонала, участващ в създаването на софтуер. Недостатъците на подхода са няколко. Преди всичко индивидуалните средства са обикновено хардуерно зависими и са предназначени за точно определена операционна и програмна среда. Използването на няколко независими средства принуждава разработчиците да разучат няколко различни потребителски интерфейса. Липсата на интеграция намалява производителността, защото не е възможно последователно извикване на някои средства, без да има дублиране на общите дейности.

12.2. Автоматизация чрез интегрирани среди

Полезнотта на индивидуалните средства за АСП е безспорна, но интегрирането им за съвместно използване би улеснило:

— предаването на информация между тях;

— ефективното изпълнение на глобални дейности, като документиране, осигуряване на качеството и управление на софтуерните конфигурации;

— реализацията на потребителски сценарии за решаване на конкретен проблем. Естествено развитие на идеята за автоматизация с индивидуални средства

е използването на групи от средства, които са проектирани да работят съвместно. Един от приносите на операционната система UNIX към разработването на софтуера е предоставянето на т. нар. PWB (Programmer's workbench), при който интеграцията се осъществява чрез общи формати на файловете, възможности за управление на версии (SCCS) и за изграждане на софтуерна система чрез описание на съставящите я части (MAKE). Идеята за разрастващо се изграждане на средства чрез „навързване“ на по-малки средства е изключително полезна. По-нататък развитието е през средите за програмиране на Ада, докато се достигне до сегашната обща теория за изграждане на интегрирани среди.

В [2] са формулирани следните изисквания към интегрираните среди за АСП:

— да осигуряват механизъм за общо използване на информацията от всички средства в средата;

— да допускат директно извикване на всяко средство;

— да поддържат решаването на всяка конкретна задача при реализацията на софтуерния проект чрез подходящо съчетаване на средства;

— да улесняват комуникациите между всички участници в процеса на създаване на софтуер;

— да натрупват статистическа информация, която да се използва за подобряване качеството на процеса и продукта.

В зависимост от начина на свързване на група от автоматизирани средства в единна среда интеграцията може да бъде:

а) еклектична интеграция — съществуващи индивидуални средства се обединяват в система чрез създаване на програма—монитор, извикваща всяко от средствата;

б) интеграция чрез данните — използване на общ модел на данните. Този вид интеграция може да бъде с различно ниво на сложност — обмен на данни между две средства чрез програма — конвертор, използване на обща съвкупност от прости символни файлове или чрез система за управление на обекти;

в) интеграция чрез потребителския интерфейс. В този случай средствата в системата използват общ стил и съвкупност от общи стандарти за връзка с потребителя;

г) интеграция чрез дейностите, които се поддържат.

Този тип интеграция се основава на модел на процеса, който определя основните извършвани дейности, резултатите от тях, потока на данни и потока на управление. Известно е и кои средства в интегрираната среда кои основни дейности поддържат.

12.2.1. Видове интегрирани среди

Всяка интегрирана среда обединява в едно положение няколко софтуерни средства, поддържащи специфични дейности в процеса на създаване на софтуер. Чрез интеграцията се постига:

— хомогенен и логически последователен потребителски интерфейс;

— лесно извикване на всяко средство и на верига от средства;

— достъп до общо множество от данни, поддържани по централизиран начин.

Удобствата за работа, предоставяни от някои среди за АСП, могат да допринесат за въвеждане в софтуерната организация на нови методи и техники. В зависимост от предназначението си интегрираните среди могат да бъдат за:

а) планиране и моделиране на бизнесинформационни системи. Този клас включва продукти, подпомагащи идентифицирането и описанието на сложни бизнесдействия. Те се използват за построяване на обобщени модели на предприятие, за да се оценят общите изисквания и информационни потоци и да се определят приоритетите в разработването на информационните системи. Средствата, интегрирани в такива продукти, включват графични редактори (за създаване на диаграми и структурни схеми), генератори на отчети и генератори на справки за срещането на отделни елементи.

б) анализ и проектиране.

Съвременните средства автоматизират най-често използваните подходи за анализ и проектиране — структурния, обектно ориентирания и подхода на Джак-сън. Те обикновено включват един или повече редактора за създаване и модифициране на спецификации и други средства за анализирането, симулирането и трансформирането им. Например Excelerator има редактори за създаване на диаграми на потока на данните, на блок-схеми и на диаграми същност—връзка. Той включва още редактор и симулатор за създаване и тестване на макети на системните входове и изходи (форми и отчети), както и генератор на код, който създава първичен код на Кобол на основата на блок-схемите.

Възможностите на средствата от този клас зависят от:

— нивото на формализираност на използваната нотация. Ако тя е неформална (структурен английски или друго свободно текстово описание), то се осигурява само редактиране и съставяне на документ. Ако нотацията е полуформална (без точна семантика, но да е възможно да се проверява синтаксисът) или с формално определени синтаксис и семантика (крайни автомати или мрежи на Петри);

— от вида на приложението — дали преобладава обработката на данните, както е в банковите и счетоводни системи, или управленските функции;

в) създаване на потребителски интерфейс

Смята се, че потребителският интерфейс е определящ за пазарната реализация и използване на всяка софтуерна система. Затова са създадени интегрирани среди, които позволяват на разработчика да създава и да тества лесно компонентите на потребителския интерфейс и да ги свърже с приложната програма. Те включват:

— графични редактори за създаване на прозорци, диалогови кутии, икони и

др.;

— симулатори за тестване на създадените компоненти преди интегрирането им с приложението;

— генератори на първичен текст;

— библиотеки за поддържане на генерирането на изпълним код.

г) програмиране

Те включват текстов редактор за създаване и променяне на текста на програмите, компилатор, свързващ редактор и дебъгер. Характеризират се с:

— удобен потребителски интерфейс;

— управление на създаваната по време на сесия информация — файлове с първичен текст, междинни, обектни и изпълними файлове. За ускоряване на съставянето

и тестването на програмата се съчетават компилатор с интерпретатор и свързване с отчитане на направените промени. Примери за такива среди са Turbo C++, Turbo Pascal, Microsoft C++ Developer Studio.

д) верификация и валидация

Такива среди подпомагат модулното и системно тестване и обикновено включват:

- статичен анализатор за създаване на управляващ граф на програма и граф на извикванията;
- средство за инструментиране на програмата и за проследяване (трасиране на изпълнението) при динамичен анализ;
- генератор на тестови данни;
- управляваща програма — средство за реализация на тестването, което създава, съхранява и поддържа тестови данни, сценарии, резултати и документация.

е) съпровождане

Средата за съпровождане управлява внасянето на промени, създаването и контрола на версии и за управление на софтуерните конфигурации.

ж) управление на проекти

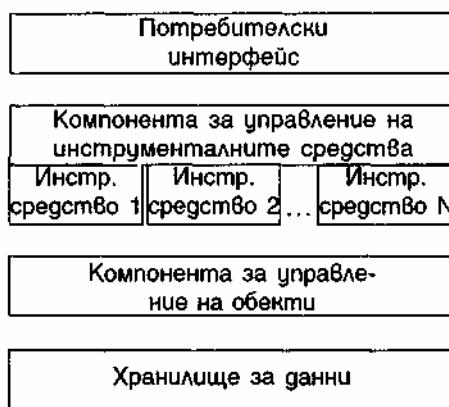
Подпомагат се планирането, съставянето на графици и текущото следене на изпълнението на проекти.

12.2.2. Принципи на изграждане и архитектура на интегрираните среди

Ще представим концепцията за изграждане на АСП-среди, следвайки изложението в [3].

Предназначението на АСП-средата е да подпомага цялостния процес на създаване на ПП, като поддържа *хранилище* (repository) на информацията, необходима за осъществяване на софтуерните разработки.

Поддържането и използването на информацията от хранилището се осъществява чрез *интегрираща архитектура*, представена на фиг. 12.1. Основните компоненти са: база от данни, в която да се съхранява информацията, система за управление на обекти, чрез която да се управлява променянето на информацията, механизъм за управление на инструменталните средства (за координиране на използването им) и потребителски интерфейс. Повечето модели представят тези компоненти като слоеве.



Фиг. 12.1.

Ще опишем накратко основните й компоненти.

Потребителският интерфейс осигурява удобна и ефективна работа със системата. **Протоколът на представяне** е множество от указания, чрез следването на които всички АСП-средства в системата се използват по подобен начин. Така екраните имат едно и също разпределение на отделни области с еднакво предназначение; има правила за имената и организация на менюта, икони и обекти; стандартизирано е използването на клавиатура и мишка; описан е общ механизъм за достъп до средствата. С използването на протокол на представянето се постига унифицираност.

Слоят на индивидуалните средства включва самите средства и програми за управлението им. Препоръчва се управляващите програми да са проектирани и реализирани така, че да се поддържа динамична съвкупност от индивидуални средства. Това би позволило на потребителя да съставя нужната му конфигурация от средства и да я променя, като добавя, изключва или модифицира някои средства. Ако в системата се поддържа многозадачна работа, управляващите програми осигуряват синхронизация при прилагане на средствата, регулиране на потока на данните, следене на правата за достъп и прилагане на метрики за ефективността на използване на всяко от средствата.

Слоят за управление на обекти осъществява интегрирането на средствата с данните от хранилището и управлението на софтуерните конфигурации. Той е съвкупност от програми, които за всяка заявка идентифицират обектите от съответната софтуерна конфигурация и ги представят във вид, подходящ за

съответното средство; поддържат различните версии, управляват планирането, внасянето и документирането на промени, регистрация и поддържане на описание на всеки елемент на софтуерна конфигурация.

Слоят за управление на хранилището включва базата данни на системата и функциите за управлението ѝ. Идеята за централизирано съхраняване на данните присъства във всеки модел на интегрирана среда, макар и под различни имена: АСП-база от данни, база на софтуерните разработки, хранилище и др. Съхраняваната информация може да се раздели на две: обща информация (свързана със софтуерната фирма като цяло и стила на работа в нея) и информация за всеки конкретен софтуерен проект. Общата информация може да включва описание на организационната структура в софтуерната фирма, съвкупността от вътрешни правила и стандарти, които трябва да се спазват, описание на използваната методология на разработване, процедури за извършване на основни дейности и др.

Информацията за конкретен проект може да бъде:

- елементи на софтуерните конфигурации (първичен текст на програми, обектни модули, описания за свързване, изпълними програми, описание на връзките между елементите на дадена софтуерна конфигурация и др.);
- информация за провежданите дейности по осигуряване на качеството (планове, сценарии и резултати от тестване и проверки; резултати от прилагане на метрики, резултати от статистическа обработка на данните);
- информация, свързана с управление на проекта — планове, оценки на трудоемкост и продължителност, графики, отчети от проследяване на проекта в определени контролни точки);
- документация — съпровождаща и потребителска.

Съдържанието и начинът на организация на данните в хранилището се определят при конкретната реализация на концептуалния модел.

Освен обичайните функции на СУБД към компонентата за управление на хранилището са формулирани някои допълнителни изисквания:

- да осигурява интегрираност на данните — да проверява всички елементи така, че да не се допуска дублиране (редундантност) на данните; да осигурява съвместимост между свързаните обекти, автоматично да извършва последователните модификации, когато промяната на един обект изисква промяна на всички свързани с него обекти;
- да осигурява механизъм за използване на информацията от множество разработчици и различни инструментални средства; да управлява многопотребителския достъп до данните и чрез защитни механизми да предотвратява наслагване на промените;
- да осигурява интеграция на данните и индивидуалните средства в системата чрез поддържане на механизми за трансформиране, управляем достъп и защита.

Така се постига съхраняването и обработването на сложни структури от Данни, които се използват ефективно от съответните инструментални средства в системата, и то така, че разработването на ПП става в съответствие с избраната методология.

12.2.3. Жизнен цикъл на софтуерните среди

Практическото използване на софтуерните среди преминава през следните шест фази:

a) избор на софтуерна среда

През тази фаза трябва да се направи проучване на софтуерния пазар и да се избере софтуерна среда, която е най-подходяща за дадена организация. Основните критерии за избора са:

- методологията на разработване на софтуер в конкретната организация, т. е. съвкупността от прилагани модели, методи, техники и стандарти. В някои случаи чрез използването на определена среда може да се премине от стихийно разработване към систематично и организирано разработване с придръжане към поддържаната от софтуерната среда методология;

- наличия хардуер в софтуерната организация. Обикновено използването на софтуерните среди изисква значителни изчислителни ресурси, които да не могат да бъдат осигурени от компютрите, с които разполага организацията;

- приложната област, в която ще бъде създаваният софтуер;

- цената на средата. По данни от [1] въпреки обещаваното увеличение на производителността на труда от 40—100% цената за закупуване, инсталација, настройване и усвояване на работата със средата може да бъде неприемливо висока.

б) настройване на средата към специфични за организацията изисквания През тази фаза трябва да се създаде версия на средата, която да съответства на използваната в организацията платформа и модел на процеса на разработване. Това изисква определяне на стойностите на параметри, определяне на елементите на системата за управление на обекти, избор на съвкупност от средства и цялостно документиране на получения вариант на средата.

в) инсталација и експериментално използване на средата Организира се обучение и пробно използване, за да се оцени полезното

на средата. Обикновено трябва да се преодолява съпротивата на разработчиците заради променения стил на работа и съпротивата на мениджърите заради големите инвестиции с неизвестна възвращаемост.

г) използване и еволюция на средата

Препоръчва се натрупване на статистически данни, въз основа на които да се оцени ефектът от използване на средата. Поддържането на обратна връзка с доставчиците би осигурило подобряване на функционирането ѝ и своевременно получаване на нови версии.

д) преустановяване на използването на средата

Възможни са две ситуации — отказ от използване на автоматизиращи среди или замяна на използваната среда с друга. Във втория случай заради съпровождането на разработени вече системи може да се наложи паралелно функциониране на двете среди.

12.2.4. Преимущества и недостатъци на автоматизацията чрез интегрирани среди

Потенциалните ползи от прилагането на средства за автоматизация са:

- повишаване на систематичността и управляемостта на софтуерните проекти;
- намаляване на стойността на разработването и особено на стойността на съпровождането;
- подобряване на качеството на софтуера;
- ускоряване на процеса на разработване, т. е. намаляване продължителността на софтуерните проекти;
- повишаване на производителността на труда;
- препоръчвани техники стават реално използваеми поради подпомагането им от софтуерни средства.

Независимо от тези преимущества АСП-средите имат все още ограничено използване. Основни причини за това са:

- няма публикувани изследвания за резултатите от практическото използване на средите;
- няма натрупани данни и подходящи метрики за измерване, как средите влияят на производителността на софтуерните разработчици;
- средите са големи и сложни софтуерни системи. Те изискват значителни инвестиции за закупуване на самите среди, за подготовка и осъществяване на внедряването им. Функционирането им е свързано с големи изчислителни ресурси. Всички тези изисквания не могат да бъдат удовлетворени в неголемите софтуерни организации.

Сложността на съвременните софтуерни системи изиска автоматизация през различните фази на разработването. Независимо от трудностите при приспособяване на автоматизиращите средства към стила на работа в конкретната организация бъдещето на софтуерната индустрия е немислимо без създаването и усъвършенстването на АСП-средствата.

Подробно описание на АСП-средства може да бъде намерено чрез [4].

Литература

1. Sommerville I. Software Engineering, Addison-Wesley Publ. Company, Fourth Edition, 1992.
2. Forte, G. In Search of the Integrated Environment. CASE Outlook, March/April 1989, pp. 5—12.

3. Pressman, R. Software Engineering — A Practitioner's Approach. R.S. Pressman & Associates, Inc. 2000.
4. <http://www.rspa.com/spi/CASE.html>

13. СЪВРЕМЕННИ ТЕХНИКИ ЗА ПРОГРАМИРАНЕ И РАЗРАБОТВАНЕ НА СОФТУЕР

В зависимост от приложната област и конкретните изисквания към създавания програмен продукт могат да се избират различни програмистки техники или подходи на разработване. Ще разгледаме някои от тях.

13.1. Програмиране, осигуряващо надеждно функциониране на програмните системи

Надеждността на функциониране е динамична характеристика, която е свързана с честотата на сривове в системата.

Под **срив на системата** се разбира ситуация по време на работа на софтуера, при която той започва да се държи по необичаен и неочекван начин.

Дефект е наличие в програмата на една или няколко грешки, водещи до срив при изпълнение на програмата с определени входни данни и попадане в определен програмен сегмент.

13.1.1. Разработване на софтуер с минимизиране на дефектите в него

Създаването на програмни системи без грешки (*fault-free software*) е свързано с големи разходи и може да се приложи само за системи, функционирането на които е свързано с животоопасни или изключително скъпо струващи дейности (управление на вредни производства и атомни електроцентрали, космически изследвания и др.). Затова обичайната практика е след тестване до определено ниво системата да се разпространява и да се отстраняват грешките, откривани по време на използването ѝ.

Избягване на дефекти (*fault avoidance*) е подход на разработване, целта на който е да се намали възможността за внасяне на грешки в програмите.

Избягването на дефекти и разработването на софтуер с минимален брой грешки се основава на следните **принципи**:

- а) създаване на прецизни, по възможност формални спецификации;
- б) използване на проектиране, което позволява капсуляция (скриване) на информацията;
- в) използване на механизмите за осигуряване на качеството със систематична верификация и валидация на системата (вж. глава 8.) в определени контролни точки;
- г) планиране и осъществяване на тестването на системата така, че да се откриват и грешките, останали след верифицирането и валидирането.

Обобщавайки натрупания практически опит, в [1] са формулирани следните **препоръки**:

— да се избягват, доколкото е възможно, конструкции на езиците за програмиране, които са потенциални източници на грешки. Например: използване на числа с плаваща точка (има проблеми при сравняването им); използване на указатели — директното общение към паметта крие опасности; паралелели-зъм (трудно се прогнозира ефектът от

динамичното взаимодействие между паралелни процеси, трудно се тества, доколкото взаимодействията зависят от конкретните обстоятелства); рекурсия (изисква висока програмистка квалификация); използване на системни прекъсвания (предава се управлението независимо от изпълняния код).

— да се използва принципът на скриване на информацията, който е заимствай от военните организации (*need to know* — необходимост да знае — достъп до тази информация има само онзи, който се нуждае от нея, за да изпълнява задълженията си). Прилагането на този принцип при разработването на софтуер означава, че всяка програмна компонента има достъп само до данните, необходими за реализираните от нея функции. Тази препоръка се улеснява от въведената в съвременните езици за програмиране система от типове данни.

13.1.2. Разработване на софтуер с приемливо ниво на грешни

Софтуер с приемливо ниво на дефекти (fault tolerance) е софтуер, който е проектиран и програмиран така, че продължава работата си и при наличие на дефекти.

Софтуерните системи, следвайки този подход на разработване, трябва да реализират следните основни функции:

- а) прогнозиране или откриване на дефекти, по възможност преди да са довели до срив на системата;
- б) оценка на пораженията след срив на системата (т. е. определяне кои нейни части са засегнати);
- в) възстановяване след срив. Това може да се постигне чрез отстраняване на повредите или чрез връщане към някое предишно устойчиво състояние на системата.
- г) локализиране и отстраняване на дефектите, предизвикали срива на системата. Осигуряването на надежно функциониране може да се осъществи чрез **принципа на повторящите се елементи**. Той се състои в това, че за извършване на една и съща дейност се разработват различен брой (обикновено нечетен) компоненти. Всяка компонента изпълнява определените функции, след което се сравняват получените резултати. По-нататъшната работа на системата продължава с най-често повтарящия се резултат.

При софтуерните системи този принцип се реализира чрез *N-версийно програмиране*. Избира се критична за функционирането на системата компонента. Тя се възлага за проектиране и програмиране на различни групи и всички версии се включват в системата. Когато е необходимо, те се извикват (паралелно или последователно) и изпълнението продължава с най-често срещания резултат.

Понякога в програмните системи се вграждат т. нар. *възстановяващи блокове* — програмни единици, проверяващи за срив на системата и включващи алтернативен код, позволяващ повторение на процедурата, ако има съмнение за наличие на дефект. Използването им може да се илюстрира със следния пример. За решаването на един и същ проблем са създадени три компоненти, реализиращи три различни алгоритъма. Разработена е и тестваща компонента, която проверява правилността на получаваните резултати. За определени входни данни се изпълнява компонентата, реализираща алгоритъм 1, и резултатите от изпълнението се анализират от тестващата компонента. Ако те са правилни, изпълнението продължава. Ако има съмнения за грешки, същите входни данни се подават на компонентата, реализираща алгоритъм 2, и резултатите отново се насочват към тестващия модул. При неуспех аналогична процедура се повтаря

с третата компонента. При неуспех и след нейното изпълнение се включва специална програмна част.

В някои софтуерни системи се създават отделни *компоненти за обработване на изключенията* (*exception handlers*). Изключение е всяко необичайно събитие, което е откриваемо хардуерно или софтуерно и което може да изиска специална обработка. Така могат да се управляват реакциите на системата при настъпване на аварийни ситуации. Най-простата реакция е извеждане на съобщение и преустановяване на изпълнението. Но са възможни и по-сложни последователности от действия. Например при системите за управление в реално време, като се установи, че времето за отговор на системата ще бъде по-голямо от допустимото, компонентата за обработка на изключения трябва да включва автоматично съответните защитни и аварийни механизми.

В [2] са разгледани някои вградени в езиците за програмиране възможности за реализация на обработване на изключенията.

13.1.3. Защитно програмиране

Този подход на разработване се основава на предположението, че в програмите има неоткрити грешки. Затова още при програмирането се вграждат механизми за откриване на грешката, оценка на засегнатите програмни части и отстраняване на последствията. Една възможна техника е определяне на критични за състоянието на системата променливи и текущо проверяване на стойностите им чрез твърдения, описващи условия или пресмятане на контролни суми.

Например в СУБД този принцип се реализира, като промените в базите от данни се съхраняват междинно в някакъв буфер и се внасят в съответната база само след успешно завършване на поредната транзакция.

Друг пример на защитно програмиране е използването на контролни точки. Този принцип се прилага най-често при сложни изчислителни задачи. Изчисленията се извършват поетапно. Всеки етап завършва с контролна точка, в

която се запомнят междинните резултати. Във всяка контролна точка изпълнението може да бъде прекъснато и да продължи след време от достигнатото в момента състояние. При срив на системата се осъществява връщане към последното запомнено състояние.

13.2. Разработване с прототипиране

Смисълът на понятието прототип е „първи от даден тип“. Създаването на прототипи е обичайна практика в материалното производство, когато се разработва опитен образец и след успешни приемни изпитания започва серийното му производство.

Прототипният модел на жизнения цикъл е представен в частта 2.3.7. на учебника и показва последователността на извършваните дейности. Тук ще разгледаме начина на осъществяване на прототипирането с отчитане на особеностите на софтуерното производство, като се придържаме към приетата в [3] терминология.

В началото на софтуерен проект потребителят понякога не може да формулира какво точно иска и от какво се нуждае. Разработчиците невинаги могат да преценят дали изискванията са изчерпателни, реалистични и дали някои проблеми имат приемливо решение.

Създаването на прототип зависи от много фактори, например от приложната област, от сложността на приложението, от характеристиките на потребителите и на проекта.

Прототипирането е съвкупност от дейности, осигуряващи ранно демонстриране на някои свойства на софтуера, който трябва да бъде създаден.

Основна цел на прототипирането е да се определят изискванията към разработваната софтуерна система.

Прототипирането се осъществява в четири последователни стъпки:

а) проектиране на прототипа, т. е. избор на функциите и свойствата, които прототипът ще изяви

Прототипирането може да бъде *вертикално*, като се реализират само някои функции, но в техния предполагаем окончателен вид. При *хоризонталното* прототипиране всички функции присъстват, но не са реализирани с пълните си възможности.

б) създаване на прототипа

Обикновено усилията за създаване на прототип са по-малки от тези за създаване на целевата система. Това се постига с избора на функции и на подходящи техники и средства за разработване. Някои характеристики на качеството могат да се игнорират, като в прототипа се изберат само свойства, важни за конкретното приложение — например време на отговор в системи за управление на процеси в реално време, надеждно функциониране и защита на данните в банкови системи и др.

в) оценяване на прототипа

Тази стъпка е важна, защото осигурява обратна връзка. Оценяването се извършва на основата на документ, описващ явно критериите за оценяване и определящ кой, кога, къде и как го извършва. В зависимост от функциите на прототипа оценката се прави от експерти, от отделен потребител, който ще работи със системата, или от групи потребители с регламентирани комуникации между тях.

г) използване

Прототипът или се отхвърля, или се използва като база за разработването на целевата система. Той трябва да се създава достатъчно рано, като се реализира цикълът представяне на прототипа — оценяване — модифициране. Затова прототипът трябва да може лесно да се променя, като се модифицират съществуващи или се добавят нови функции.

Одобреният прототип може да се използва като учебна среда за евентуални потребители на реалната система. Демонстрираното от прототипа и одобрено при оценяването трябва задължително да се възпроизведе в крайния продукт.

Обикновено прототипът не се вгражда директно в целевата система поради следните причини:

— за да се разработи бързо, може да е избрана неподходяща за целевата система операционна или хардуерна среда;

— някои съществени за системата характеристики могат да бъдат игнорирани в прототипа, но да са задължителни за крайния продукт;

— поради многократното модифициране на прототипа той може да е с ниска съпровождаемост.

В зависимост от поставените цели разграничаваме три вида прототипиране.

Целта на **изследователското** прототипиране е да се изяснят изискванията към целевата система чрез представяне на няколко алтернативни решения. То е подходящо за ранните фази от жизнения цикъл — изследване и анализ на осъществимостта. Препоръчва се само ако има техники и инструментални средства, позволяващи бързо създаване на прототипа, и то с минимални усилия.

Целта на **експерименталното** прототипиране е да провери дали избрано решение на даден проблем е подходящо. Експерименталният прототип се разглежда като междинна стъпка между специфицирането и реализацията.

Еволюционното прототипиране е с най-големи възможности, но е най-отдалечено от оригиналното разбиране за прототипиране. Основната му идея е, че разработването се осъществява в динамична среда и непрекъснато променящи се изисквания. Затова крайният продукт се създава като последователност от инкременти, всеки от които е прототип за следващия. Инкременталното (постепенно разрастващо се) разработване се осъществява чрез многократно повтаряне на цикъла проектиране — реализация — оценяване.

За да бъде прототипирането ефективно, прототипът трябва да се разработва бързо, за да може потребителят да оцени резултатите и да препоръча изменения. Най-често използваните методи и средства могат да се разделят в три групи:

— **езици от четвърто поколение**

Езиците от четвърто поколение са например езици на заявките и генериране на отчети в базите от данни, генератори на програми и приложения и други непроцедурни езици от високо ниво. Те дават възможност на разработчика бързо да генерира изпълним код и затова са подходящи за създаване на прототипи.

— **използване на готови софтуерни компоненти**

В този случай някои от частите на прототипа могат да се изберат от съществуващи библиотеки от елементи за повторно използване.

— **среди за формално специфициране и прототипиране**

Съществуват среди [4], които дават възможност след определяне на изискванията, да се създават формални спецификации за системата и чрез инструментално средство те да се превръщат в изпълним код. Потребителят може да използва така създадения прототип за подобряване на формалните изисквания и чрез повтаряне на гореописаната последователност прототипът да еволюира до исканата целева система.

Основно предимство на прототипирането е, че позволява изясняване и уточняване на функционалните и нефункционални изисквания към целевата система достатъчно рано. Недостатък е, че прототипирането изисква допълнително време и средства. В някои случаи може да се окаже, че икономически е по-изгодно да се променя завършената система, вместо да се създават прототипи за нея.

13.3. Повторно използване в софтуерното производство ("Re-use")

Подходът за **повторно използване** (ПИ) е съвкупност от планирани и систематични дейности, насочени към максимално използване на съществуващи софтуерни елементи в процеса на създаване на нов софтуер.

Този подход е заимстван от материалното производство и се оказва приемливо решение за намаляване на сроковете и стойността на софтуерните разработки.

Софтуерните елементи, които се използват повторно, могат да се класифицират по различни критерии.

а) класификация по *същността им*:

— *идеи или концепции*. В тази група са алгоритми, методи, техники или формални модели.

— *софтуерни компоненти*. Това са най-често използваният тип елементи и могат да бъдат:

— *приложни системи*. Например реализация на една и съща програмна система на различни платформи (т. е. в различни съчетания на операционна и хардуерна среда).

— *подсистеми*. Някои подсистеми реализират съвкупности от функции, които са с универсално предназначение. Например подсистема за обработка на грешките, за реализиране на вградена помощна функция, за прилагане на съвкупност от софтуерни метрики, за натрупване на статистическа информация и др.

— *програмни модули, функции или други обособени програмни части*;

— процедури или умения, свързани с процесите на създаване на софтуер. В тази група е know-how информация, която може да бъде под формата на наблюдения, препоръки, експертно знание и др.

б) класификация *по обхват*

В зависимост от формата и докъде се простира повторното използване, то може да бъде вертикално или хоризонтално.

При *вертикално* повторно използване в избрана приложна област се създават основни модели, които се прилагат във всички разработвани за тази област софтуерни системи.

При *хоризонталното* повторно използване се създават универсални компоненти, които могат да се вграждат в програмни системи за различни приложни области — например средства за управление на бази от данни, за разпределени среди, за изграждане на графичен потребителски интерфейс и др.

в) класификация *по начина на осъществяване*

— *планирано и систематично* повторно използване. В съответната софтуерна организация са регламентирани основните принципи и процедури за осъществяване и оценка на подхода.

— *инцидентно* (ad-hoc) повторно използване. За конкретен софтуерен проект се търсят съществуващи софтуерни елементи, удовлетворяващи формулирани изисквания.

г) класификация *по използвана техника*:

— *сглобяващо* ПИ. Съществуващи софтуерни компоненти се вграждат като основни блокове в софтуерните системи.

— *генериращо* ПИ. Използват се идеи или концепции само на спецификационно ниво, след което чрез специално разработени генератори на програми автоматично се създават съответните програмни части.

д) класификация *по начина на използване*:

— *без промяна*. Съществуващи софтуерни елементи се прилагат без изменения. Този вид повторно използване трябва да се предпочита, защото се основава на проверено качество и минимизира усилията за съпровождане.

— с **модифициране**. В този случай се изисква допълнително специфициране и реализация на измененията и тестване, за да се провери коректността на получения вариант.

е) класификация по вида на използвания **софтуерен продукт**. Могат да се използват спецификации, проекти, фрагменти от програмен текст, документация, тестови примери и др.

Ще се спрем накратко на проблемите, свързани с осъществяване на подхода за повторно използване в конкретна софтуерна организация.

Създаването на софтуер в една и съща приложна област или в една и съща среда дава възможност за натрупване на знания и опит, които могат да се използват и в следващи проекти. Преди всичко необходимостта от въвеждане на подхода трябва да бъде осъзната от ръководството на фирмата, което да планира, финансира и управлява конкретна програма. Мотивите за ПИ могат да бъдат подобряване на качеството на създавания софтуер, доколкото се използват елементи с проверени вече свойства, и по-бързо излизане на пазара с нови продукти, тъй като се намалява времето за разработка. Препоръчва се следната постъпкова процедура:

— идентифициране на компонентите за ПИ. Обикновено се анализира приложната област, като се откриват и обособяват общи модели и свързаните с тях структури. Могат да се изследват и създадените във фирмата програми, които да се класифицират по реализираните функции;

- документиране на всяка компонента и включване в общодостъпна библиотека;
- обучение на всички участници в разработката, как да използват библиотеката.

Например след решение да се приложи повторно използване в Nee Software Engineering Lab е била съставена библиотека от 32 групи програми, реализиращи 130 алгоритъма с универсално предназначение.

Друга софтуерна организация анализира 5 000 програми на Кобол, разработени в нея, и ги класифицира в три групи: редактиращи, обработващи и съставящи справки. След целенасочено повторно използване на съществуващи програмни фрагменти са постигнати 50% увеличение на производителността на програмисткия труд, като повторно използваните елементи са съставлявали 60% от новоразработвания софтуер [5].

Съществени за успеха на подхода са техниките за съхраняване и извличане на компонентите. За да се опише какво се търси, се използват някои класификационни схеми, познати от библиотечните системи. Механизмите за търсене могат да се основават на ключови думи, на текстови описания на естествен език, фасетна техника чрез съставяне на описания от различни гледни точки и т. н.

Препоръчва се сформиране на специална група от високо квалифицирани специалисти, която да осъществява подхода за ПИ в софтуерната организация. Основна дейност на групата е разработване и съпровождане на компоненти за ПИ. Проектирането и създаването на такива компоненти изисква от 5 до 10 пъти повече усилия, отколкото за обикновените компоненти. От програмистка гледна точка трябва да се вземат предвид следните особености:

- а) програмните компоненти трябва да се проектират и програмират така, че да са лесно настройваеми и да решават клас сходни задачи;
- б) да се обмисли и реализира систематично процесът на „клониране”, т. е. копиране на избран програмен фрагмент на съответно място в новата програмна система;
- в) да се регламентира механизъм за съставяне на имена на променливи за предотвратяването на програмни грешки от дублиране на използваните имена;

г) програмните компоненти да са преносими, т. е. да могат да работят в различни операционни и хардуерни среди. Формулирани са някои принципи, следването на които би осигурило висока преносимост на разработвания софтуер. Основен принцип е всички зависещи от средата програмни части да се оформят като самостоятелни модули. При това трябва да се отчитат различната дължина на машинната дума, различното представяне на числовата и символна информация, различните съвкупности от системни прекъсвания и др.

Същата група от специалисти поддържа и библиотеката за ПИ — контролира съдържанието и качеството на съхраняваната информация, създава описание за нови елементи, изтрива неизползвани елементи, консулира потребителите, натрупва статистика за използването ѝ и др.

В началото на всеки нов проект се изследва възможността да се използват някои компоненти от библиотеката със или без допълнително модифициране.

Прилагането на подхода за повторно използване е все още ограничено поради следните причини:

— много от разработчиците имат високо професионално самочувствие и предпочитат създаването на собствени софтуерни компоненти пред търсенето, разучаването и евентуално модифициране на съществуващи. Пренаписването на дадена компонента обикновено подобрява качеството ѝ, защото може да се отчете опитът от използването ѝ.

— ефективно повторно използване може да се осъществи само при наличие на софтуерни средства, които го подпомагат на методологическо и техническо ниво.

В [5] са анализирани резултатите от прилагане на ПИ в редица индустриални софтуерни организации.

Основни тенденции в изследванията са:

- проектиране на библиотеки от елементи за ПИ и вграждането им в среди за разработване на софтуер;
- създаване на стандарти за интегрирано използване на библиотеките от елементи за ПИ;
- създаване на модели за процеса на ПИ и на съответни инструментални средства, поддържащи разработването за и с повторно използване.

13.4. Разработване на софтуер с участието на потребителеля

Този подход започва да се използва в началото на 90-те години и е включен в списъка на единадесетте най-перспективни решения на софтуерната криза, съставен от Йордън. Въпреки сравнително краткия период на реалното му използване вече има публикации за положителни резултати от прилагането му [6].

Ще представим накратко същността и особеностите на реализацията му.

В зависимост от **инициализацията** си софтуерните проекти могат да се разделят на две групи.

Автономни са проектите без конкретен възложител. Обикновено до идеята за създаването им се достига след маркетингово проучване на състоянието и тенденциите на софтуерния пазар. Тези проекти се самофинансират от фирмите разработчици, като функционалните им възможности и изискванията, които трябва да удовлетворяват, се определят чрез изследване на съществуващи аналоги (действащи или описани в литературата) и обобщаване мнението на потенциалните потребители.

Поръчани (възложени) са проектите с конкретен възложител, който финансира разработката и след завършването ѝ става нейн собственик. Класическите подходи на разработване в този случай предвиждат участие на възложителя в началните фази (изследване, анализ на съществяване) до дефиниране на изискванията и отново чак след окончателното му завършване — за оценка и приемане на готовия програмен продукт.

Основна идея на подхода на разработване с участието на потребителя е да се повиши активността и съответно отговорността на потребителя за качеството на програмния продукт, като той се включи активно и в някои междинни фази.

Ще дадем следната дефиниция:

Разработване с участието на потребителя е съвкупност от методи, техники и целенасочени действия, осигуряващи активното участие на потребителя в проектирането и създаването на софтуерната система, която той ще използва.

Под потребител се разбира лице или лица, които ще използват софтуерната система за решаване на проблеми в ежедневната си работа. Идеята за разработване може да бъде на мениджъра или на друг представител в управленската йерархия, но партньор в обсъжданията и съвместната работа трябва да бъде непосредствен потребител.

Подходът се избира в началото на възложен проект и се следва систематич- I но и планирано. Предполага се активно участие на потребителя. Осъществява се не само регистриране на реакциите му при работа с междинни версии на програмния продукт, а включването му в процеса на вземане на решения в критични за проекта ситуации.

Реализацията на подхода изиска специални техники на разработване. Една възможна и проверена в практиката техника е еволюционното прототипиране, при което разработването на ПП се разглежда като създаване на последователност от междинни ПП, всеки от които е разширение на предишния. След завършване на един прототип се извършва едновременно тестване на готовия прототип в реални условия (от потребителя) и създаване на следващия прототип (от разработчика). Освен ранно откриване на грешки използването на прототип в реална потребителска среда позволява формулирането на допълнителни изисквания.

Съществени за успеха на подхода са:

а) правилното декомпозиране на програмния продукт на прототипи, всеки от които да може да се използва самостоятелно и да е с максимална гъвкавост — лесно да се добавят нови или да се модифицират съществуващи програмни части. От значение е и реализацията на „защитно програмиране“, позволяващо възстановяване на системата след аварийни ситуации, предизвикани от неоткрити програмни грешки или неправилни потребителски действия. Потребителят трябва да има свободата да експериментира. Чувството, че нищо непоправимо не може да се случи, осигурява безстресова работа с прототипа и поощрява инициативността му.

б) регламентиране на комуникациите с потребител — честота на обсъжданията, вземане на решения с консенсус на основата на разумни компромиси и от двете страни, поощряване на активността на потребител и поддържане на самочувствието му на пълноценен партньор, така че практическият му опит и познаване на приложната област

да доведат до избор на решения, непостижими по умозрителен път или чрез проучване на литература.

Основното преимущество на подхода за разработване с участието на потребителя е, че се създава ПП, който не се нуждае от поправяне на грешки или доработване поради пропуснати или погрешно разбрани потребителски изисквания. Недостатък на подхода е необходимостта от овладяване на специални техники за разработване.

Литература

1. Sommerville I. Software Engineering, Addison-Wesley Publ. Company, Fourth Edition, 1992.
2. Sebesta R. Concepts of Programming Languages. Benjamin/Cummings Publishing Company, Inc., 1993.
3. Boar, B. Application Prototyping. Wiley-Interscience, 1984.
4. Rzepka, W., and Y.Ohno (eds.) Requirements Engineering Environments. IEEE Computer (special issue), vol.22, No 5, May 1989.
5. Proceeding of the Symposium on Software Reusability, Seattle, Washington, April 28—30, 1995.
6. Communication of the ACM. Special Issue on the Participatory Design, vol. 36, No 4, June 1993.

14. ЧОВЕШКИЯТ ФАКТОР В РАЗРАБОТВАНЕТО И ИЗПОЛЗВАНЕТО НА СОФТУЕРА

Софтуерът се създава от индивиди с определени физически и умствени възможности и различна психологическа нагласа. Изследването и съобразяването с тези личностни характеристики може да подобри организацията на работата и да повиши производителността на труда на разработчиците на софтуер [1]. От друга страна, софтуерът се използва от хора и отчитането на психологическите им особености в началните фази на анализ и проектиране може да доведе до създаване на надеждни, ефективни и удобни за използване системи.

Ще се спрем накратко на някои проблеми, изследвани от софтуерната психология.

Ефективното управление на софтуерните проекти се основава на управлението на хората, продукта, процеса и проекта (т. нар. четири P's — people, product, process and project). Наредбата не е случайна. Мениджърите в софтуерното производство са осъзнали, че подценяването на човешкия фактор може да застраши успешната реализация на всеки проект. Това е обяснението за съживения интерес към софтуерната психология. Известният Software Engineering Institute разработи CMM-модела [2], осигуряващ мярка за глобалната ефективност на дейностите в дадена софтуерна организация чрез определяне на различни нива на зрелост (вж. глава 8.). Беше разработено и разширение на модела — PM-CMM (People Management Capability Maturity Model), предназначението на което е „да подобри готовността на софтуерните организации да осъществяват приложения с нарастваща сложност чрез привличане, обучаване, мотивиране и задържане на талантите, необходими за подобряване на техните възможности за разработване на софтуер“ [3]. Ще разгледаме някои от ключовите дейности, определени от този PM-CMM модел.

14.1. Как да се наемат най-добрите софтуерни специалисти

Всяка софтуерна организация трябва да има кадрова политика, регламентираща принципите и процедурите по назначаване и освобождаване на служители, системата от поощрения и наказания, правилата за издигане в служебната

йерархия и др. Някои страни на тази политика зависят от това, че създаването на софтуер не е изцяло производствена дейност, а има елементи и на творческа, интелектуална дейност.

Набирането на кадри (recruitment) обхваща всички дейности, насочени към намиране и назначаване на най-подходящия кандидат за определена длъжност. Разглежданите кандидатури могат да бъдат от самата организация (internal recruitment) или външни (external recruitment) [4].

При **вътрешното набиране** след определяне на длъжностните характеристики за вакантните позиции се разглеждат възможностите за преназначаване. Използваните техники могат да бъдат:

поддържане на информация за всички служители (worker pool repository) професионални знания и умения, квалификация, опит от предишни софтуерни проекти и т. н. Тази информация може да се използва или за допълване на постоянно действащи екипи за разработване (за да се удовлетворят специфичните изисквания на новия проект), или за динамично сформирани екипи за всеки нов проект в организацията.

вътрешен конкурс. Свободните места се обявяват и всички служители, които се интересуват, могат да подават съответните документи, като в някои случаи се изиска и съгласието на непосредствения ръководител на кандидата;

назначаване на основа на препоръки (employee referrals).

При **външното набиране** целта е да се потърсят и назначат нови за софтуерната фирма специалисти, които са необходими поради разрастване на дейността, започване на проекти в неизследвана приложна област или проекти, изискващи прилагането на нови методи на разработване. Основни техники са:

- използване на връзки със съответните университети и привличане на най-способните завършващи студенти;
- използване на услугите на специализирани фирми за подбор на кадри (обикновено за ръководни или особено важни длъжности);
- примамване на висококвалифицирани специалисти (head-hunting), които не само не са безработни в момента, а обикновено имат успешна кариера в друга конкурентна организация. В този случай специално наети външни експерти проучват мотивацията, условията на живот и работа на избранника, за да се формулира такъв пакет от финансови и други предложения, който би го изкушил да приеме новата работа в друга организация.

За избор на подходящи служители се предлага използването на т. нар. психологически профили. Психологическият профил е съвкупност от индивидуални личностни качества, за които се смята, че са важни за съответната професия. Създадени са модели на профил за програмисти. При назначаване чрез тестове се изследва психологическият профил на всеки кандидат и се сравнява с този на най-добрите професионалисти във фирмата. Реалното използване на този подход изиска разработване на адекватни тестове, съобразени с националните, културни и професионални различия.

Не се препоръчва изборът на софтуерни специалисти да се основава изцяло на психологическия профил поради следните причини:

- личностният психопрофил се променя във времето в зависимост от възрастта, условията на работа, междуличностните отношения и др.;
- различни качества са ценни за различните позиции в софтуерното производство. Например общоприетите за отрицателни качества „заядливост“ и „придирчивост“ са полезни за функциите на отговорника по тестване;
- тестовете за психопрофил могат да бъдат разучени предварително и интелигентните кандидати да отговарят заблуждаващо така, че да се получи исканият профил.

Друг подход е предложен от Уейнбърг. Той предлага изследването само на две характеристики, които смята за задължителни за работещите в областта на софтуерните технологии: адаптивност и устойчивост към стрес.

Под *адаптивност* се разбира способността за бързо приспособяване към изменящата се среда. За програмистите от началото на 80-те години беше достатъчно да изучат една операционна система и един език за програмиране и да ги ползват до пенсионирането си. Сегашните разработчици на софтуер са принудени да усвояват непрекъснато нови подходи и съответните им техники. За успешна професионална изява е необходимо личността да има психологическата нагласа, че за реализацията на всеки проект ще е необходимо допълнително обучение.

Статистическите изследвания показват, че разработването на софтуер е измежду десетте най-застрашени от стрес занимания в съвременното общество. Със съдействието на професионални организации в САЩ и Япония са проведени изследвания и са идентифицирани следните стресови фактори [5]:

- напрежение от сроковете;
- претовареност;
- недостатъчна почивка;
- твърде много извънредна (включително нощна) работа;
- недостатъчно време за обучение;
- технически проблеми;
- твърде много рутинна работа;
- проблеми в комуникацията с членовете от групата;
- проблеми в отношенията с потребителите;
- неприятности след инсталациране на нов програмен продукт;
- нееднозначност на спецификациите;
- малък шанс за вземане на собствени решения;
- ограничени възможности за индивидуална изява;
- неравномерно разпределение на работата;
- недостиг на квалифицирани членове на екипа;
- лоша среда за разработване на софтуер;
- лоши условия за работа;
- ограничения при напредване в кариерата;
- неудовлетвореност от системата за материално поощряване и връзка свършена работа — заплащане;
- бърза промяна в хардуерните и софтуерните технологии. Индивидуалната устойчивост към стрес, съчетана с целенасочените мениджърски действия за контролиране на стресовите фактори, би осигурила висока производителност на създателите на софтуер.

14.2. Определяне на структурата и състава на работните групи

Ще разгледаме някои основни принципи за сформирането на работните групи.

Идентифицирани са четири типа комуникативно поведение при изпълнение на групови дейности: обособяващ се, водим, лидер и сътрудничаш.

Основното за хората с *обособяващ се* тип комуникативно поведение е устойчивата нагласа към индивидуализъм. При разпределението на функциите на такъв член на групата трябва да се възлагат задачи, които са относително самостоятелни и усамотяването няма да повлияе отрицателно върху изпълнявания проект.

Хората с *водим* тип комуникативно поведение имат подчертана нагласа към доброволно подчинение и силно изразено предпочтение към изпълнение на възложена работа под непосредственото ръководство на лидера на групата. Предполага се, че установката към водимо сътрудничество се дължи на конформистката природа на личността или на видимо преобладаваща внушаемост.

Анализът на *лидерския* тип комуникативно поведение е изключително важно за планиране на дейността на групата. Психологическият профил на лидерите е с нагласа към доминиране и предполага доброволно подчинение на останалите.

Хората с предпочтения към *сътрудничество* показват стабилен стремеж към колективно решаване на общата задача, бързо ориентиране към рационалните решения благодарение на повишената ориентационна способност на личността. В този случай е характерна готовността да се приемат и следват разумните решения и да се организира работата в групата за тяхното осъществяване.

На основата на типовете поведение са формулирани и някои препоръки за груповата дейност на разработчиците на софтуер. Ефективна работа може да се постигне, като на всеки член се възлагат задачи, които са съобразени със стила на неговото поведение.

Смята се, че поради склонността си към самоизолиране на обособяващия се тип трябва да се възлагат задачи, които не изискват взаимодействие и чести контакти с други хора. Например на разработчик от този тип не бива да се възлагат преговорите с потребителите.

За хората от *водим* тип са подходящи изпълнителски дейности. Пълната удовлетвореност от работата при тях се свързва с успеха от изпълнението на възложената задача.

На хората с ярко изразен лидерски тип ориентация трябва да се възлагат по-сложни задачи и ръководни функции поради нагласата им към поемане на отговорност, добра ориентация и организаторски способности. Например те са подходящи за осъществяване на изследователско прототипиране, за експертна оценка на междинен софтуерен продукт, за определяне каква CASE-среда да се закупи и др.

Много от задачите при създаването на софтуер са на едно ниво на сложност и най-успешно се поемат от хора със сътрудничаш тип комуникативно поведение.

Числеността и разпределението на задълженията в групата зависят от размера и сложността на конкретния софтуерен проект, който ще се осъществява. Препоръчва се да се определи съставът на групата чрез „дължностни характеристики“ за дейностите, които ще се извършват, и след това да се изберат членовете на групата с отчитане на:

- възможността и желанието да се извършва дадена дейност. Някои предпочитат изпълняването на познати, добре овладени процедури, а други — експериментирането с нови подходи и техники и проверка на нови идеи;
- опит от реализацията на подобни приложения или прилагане на определени техники и средства;
- стил на работа. Той се определя от комбинирането на две личностни характеристики — начин на комуникиране (екстраверти и интроверти) и начин на вземане на решения (интуитивно или рационално). Екстравертите предпочитат да изразяват своите схващания, а интровертите — да се вслушват в мнението на другите. Интуитивните хора реагират емоционално на проблемите и вземат решения спонтанно, а рационалните преценяват всички известни факти и възможни решения и избират едно от тях след задълбочен анализ.

14.3. Управление на взаимоотношенията в работната група

Разработването на софтуер се извършва в групи, чиито членове са обединени от обща социална дейност и се намират в непосредствено лично общуване, което е основа за възникване на емоционални отношения, групови норми и групови процеси.

От психологическа гледна точка са определени две нива на взаимоотношенията — делови функционални контакти и междуличностни взаимоотношения. Последните се основават на предпочтанията и личната привързаност между хората, субективните установки, чувството за симпатия и антипатия, доверие и подозителност. Психическата съвместимост е необходимо условие за успешна работа.

Някои задължения и права на членовете на групата са:

- да знаят какво се очаква от тях;
- да имат възможност да обсъждат възлаганите им задачи;
- да разполагат с необходимите им ресурси;
- да бъдат информирани за състоянието на проекта;
- да бъдат третирани като личности;
- да организират работата си чрез поставяне на приоритети, определяне на време за съсредоточена работа без прекъсване, установяване на собствена система за документиране и планиране на ежедневната заетост и др.

За цялостното функциониране на групата отговаря избиран или назначаван ръководител (мениджър). При определянето му се вземат предвид две групи условия — обективни (устойчиво централно положение в групата, лидерски тип комуникативна ориентация) и субективни — добре изразени и стабилни личностни качества, като висока степен на владене на знания в областта и опит в прилагането им, авторитет сред членовете на групата, организаторски способности, приемлива ценностна ориентация.

Някои от задълженията на мениджъра са:

- да определя задачите и функциите ясно и точно;
- да дискутира проблемите и възможните решения с всички засегнати;
- да разработва ясни процедури и да ги описва с подходящо ниво на детализация;

— да оценява периодично работата на всеки участник;
— да поддържа добри взаимоотношения с външни за групата хора — снабдители, администратори, потребители, подизпълнители и др. Препоръчва се предварително регламентиране на отношенията още в началото на проекта, което може да бъде оформено и като формален документ (договор или споразумение).

Някои от осъществяваните контакти могат да бъдат делегирани и на друг член на групата, който има персонални връзки или чията работа съществено зависи от външните контрагенти. Най-важни за проекта са връзките с потребителите и те трябва да бъдат определени.

Могат да се разграничават четири вида управленски профил:

- *насочващ*, когато мениджърът дава инструкции и следи за тяхното изпълнение;
- *съветващ*, когато освен даване на инструкции и проследяване на изпълнението им мениджърът обяснява решенията си и поощрява обсъжданията и даването на предложениета;
- *подкрепящ*, когато мениджърът улеснява и поддържа усилията на подчинените си и се чувства отговорен за вземаните от тях решения;
- *делегиращ*, когато мениджърът прехвърля на подчинените си отговорността за вземане на решения при възникващите проблеми.

Добрият мениджър може да използва различни стилове в отношенията си с различни хора и в зависимост от конкретната ситуация.

Психологическият климат е важен за производителността на труда и мениджърът трябва да се опитва да го осигури чрез подходящи управленски техники Една от тях е разрешаване на конфликти.

Различия в мненията, целите, приоритетите, подходите и др. могат да доведат до спорове, съперничество и конфликти. Конфликтите могат да се разрешават чрез арбитраж (изслушване на страните и вземане на решение), постигане на съгласие на основата на правила за работа, представяне на различията и опит за преодоляването им чрез взаимни компромиси или реорганизиране на работата така, че да се ограничат контактите между каращите се страни.

Основна функция на мениджъра е да организира изпълнението на определени задачи, насочвайки подчинените си и отчитайки мотивацията им. Фактори, определящи мотивацията, могат да бъдат:

- работата, която е възложена, и начинът, по който е възложена;
- стилът на контролиране и управление;
- хората, с които се работи;
- начинът на общуване с мениджъра;
- работната среда (работно място, организация, регламентиране на процедурите);
- организиране на конкретни проекти;
- използвани в ежедневната работа средства;
- възнаграждение на труда и други ползи.

За да поддържа високо ниво на мотивираност, мениджърът трябва да се съобразява с тези фактори и да се опитва да ги контролира.

Мениджърът носи отговорността за проекта като цяло, но той може да делегира част от правата и задълженията си на други членове на работната група. Обикновено

задълженията са за свършване на определена работа, а правата — за вземане на самостоятелни решения, свързани с изпълнението ѝ. Преимущество на делегирането е, че мениджърът може да се съсредоточи върху същинските управленски функции, а членовете на групата са с повишена мотивация, защото се чувстват оценени, самостоятелни и отговорни.

Мотивацията на участниците зависи и от стила на планиране и контролиране на работата. Продължителността на планирания период и честотата на проверките трябва да съответстват на квалификацията и опита на всеки участник на важността на изпълняваната от него задача.

Разработването на софтуер в група изисква съобразяването с всички споменати по-горе управленски аспекти. Продължителната работа на група в един и същ състав може да има и някои недостатъци. Нивото на „сработеност“ може да бъде толкова високо, че да не може да се смени ръководителят на групата; да се допуска вземане на решения без обсъждане на всички възможности или пък да не се проверява регулярно и систематично количеството и качеството на работата на членовете на групата. За преодоляване на т. нар. групово мислене, което ограничава инициативността и критичността към приеманите решения, се препоръчват специални управленски техники — организиране на мозъчни атаки (brainstorm обсъждания) с поощряване на нестандартните идеи; използване на външни експерти за паралелни проверки и сравняване на резултатите и др.

В [6] е предложено така нареченото egoless programming. Това е стил на разработване, при който всички софтуерни продукти (спецификации, проекти програми, документация и др.) се разглеждат като собственост на групата, независимо от кой конкретен неин член са създавани. Те могат да се изследват обективно и да се променят, без да се засяга нещие самочувствие. Този подход преодолява „когнитивния дисонанс“, който се изразява в склонността на всеки човек да приписва заслугите за успехите на себе си, а причините за провалите да търси у другите. Колективната отговорност за разработването и резултатите от него ги подобряват значително.

14.4. Организиране на комуникациите в групата

Според едно изследване на IBM разпределението на работното време на софтуерните специалисти е 50% за комуникации, 30% за самостоятелна работа и 20% — за други дейности. Затова и комуникациите между членовете на групата трябва да се регламентират така, че да се осигури ефективността им.

Формите и интензивността на контактите зависят от:

- числеността на групата. За група от N членове възможните контакти са $N(N-1)/2$;
- структурата на групата. Поради психологически или организационни причини комуникациите между членовете от различни нива на йерархията могат да бъдат затруднени;
- състава на групата, доколкото тя включва личности с различна степен на комуникативност.

Регламентираните формални контакти могат да бъдат централизирани (чрез координатор) или свободни (на всеки със всеки).

При първия вид организация се определя координатор, който приема всички съобщения и решава към кои членове на групата да ги пренасочи. Тази форма се предпочита за неголеми групи и съществени за успеха ѝ са личностните качества на координатора, като информираност, бързина на реакцията, акъратност и др.

Координиращото лице може да поеме и допълнителни функции, като присвояване на приоритет на съобщенията, документиране и архивиране на разменяната информация.

При втория вид организация всеки член на групата може да общува директно с всички останали. Недостатък на тази форма е големият поток неструктурирана информация, който изиска допълнително време за класифициране на съобщенията и може да забави обработката и очакваните като реакция действия.

14.5. Организиране на сбирки и заседания на групата

Сбирките са форма на комуникация, която поддържа колективния стил. Целта им може да бъде анализиране на текущото състояние на проекта, информиране на участниците за настъпили изменения, вземане на решения в случаите, когато е необходимо обсъждане и съобразяване с различни гледни точки и т. н.

Няма правила за броя, типа и продължителността на провежданите сбирки, но са формулирани някои препоръки [7], следването на които би повишило ефективността на работа.

а) планиране на сбирката

Преди всичко трябва да се определи необходимото ли е провеждането на сбирката и тя да се свиква само ако проблемите не могат да се решат чрез индивидуални срещи, разговори по телефона или размяна на съобщения. Всеки участник трябва да бъде информиран своевременно за деня и часа, очакваната продължителност, мястото на провеждане и дневния ред, както и за задълженията му (проучване на материали, подготовка на отчет, проект или предложения).

б) провеждане на сбирката

За ефективното протичане се определя водещ, който:

- открива сбирката, представяйки целите и начина на протичането ѝ;
- ръководи сбирката, като се опитва да поддържа изказванията и обсъжданията в конструктивен, коректен и положителен тон в съответствие с дневния ред и ограниченията във времето.

в) закриване на сбирката

Сбирката се прекратява, когато свърши заплануваното време, когато се изчерпи дневният ред или когато групата няма ресурси да продължи. Задължително е резюмиране на постигнатите резултати и взетите решения и насочване на следваща сбирка, ако е необходимо.

14.6. Ергономика

Основната цел на ергономиката е да изследва и да предложи подходяща Работна среда, така че човешките ресурси да се използват по най-ефективен начин.

Ще се спрем накратко на някои ергономични фактори, свързани с процеса на разработване на софтуера.

Организацията на работното място влияе в най-голяма степен на производителността. От значение са:

- индивидуалното достъпно пространство,

- мебелировката,
- вътрешното оформление на помещението,
- контролираното ниво на шум и температура,
- чистотата и подредеността,
- наличието на помещение за почивка с достъп до кафе машини и Изследване, финансирано от фирмата IBM. е показало, че минималните изисквания са 9 кв. м обща площ и 2.7 кв. м работна площ за всяко лице. Нарушаването на тези изисквания би трябвало да се компенсира с подобряване на някои от останалите.

Работните места могат да бъдат в общо помещение или в индивидуални стаи за всеки разработчик. Двата вида организация са с различни възможности за контрол на работата и за комуникации между участниците в работните групи. Имайки предвид творческите елементи в процеса на създаване на софтуер, възможността за самостоятелна и съсредоточена работа в индивидуални стаи или в обособени пространства се предпочита за ефективната реализация на всеки проект.

Освен харacterистиките на физическата среда от значение са и наличните комуникационни, хардуерни и софтуерни средства, предоставени на всеки участник в софтуерната разработка. Според статистическите данни подобряването им може да увеличи производителността до 3 пъти.

Като ергономични фактори могат да се разглеждат и установените в софтуерната фирма процедури и правила за осъществяване на спомагателните дейности — административни, финансово-счетоводни, снабдителски и др.

14.7. Професионализъм и етично поведение

В специализираната литература са описани случаи, в които софтуерни специалисти съзнателно са заблудили свои партньори, предоставяйки им неверна информация за извършена работа, за предполагаема продължителност или трудоемкост на предстоящ софтуерен проект, за степента на завършеност или за качествата на ПП, за приложимостта на даден софтуерен продукт за решаване на конкретен проблем и др. В резултат са били взети решения, облагодетелстващи неправомерно лицата или организацията, която те представят.

Проблемът за професионалната етика е дискутиран със софтуерни специалисти от цял свят [8]. Много от тях не само са потвърдили, че такива случаи са често явление, но и смятат, че описаното поведение е оправдано в повечето случаи. Това не са просто примери за лоши управленски подходи, а посочване на по-серioзен проблем — **липса на правила и стандарти за етично поведение**. Във всяка организация се натрупват прецеденти и повтарящи се реакции и те се предават от по-опитните на по-младите софтуерни специалисти чрез действия и недокументирани практики. Например по-младите програмисти се запознават със становището популярно правило за оценяване на обема или на стойността на извършена работа: прави се оценката, след това се удвоява и накрая се добавят още 30%. Понякога се прилага и принципът на Мърфи за определяне на необходимото време за определена работа: прави се оценката, удвоява се и се сменя мерната единица със следващата (например от час ден. от ден — на седмица и т. н.).

Такова поведение и стил на работа обикновено се улесняват от липсата стандартизирано управление на софтуерните проекти, на процедури за оценка на качеството, оценка на разходите и др., които да са широко разпространени в екипите. По време на реализация на софтуерен проект вместо решения, основаващи се на това, което би било по-добро за клиента, много софтуерни специалисти избират решения, които биха

улешили самите разработчици или софтуерната разработка. Така ситуацията на некоректно отношение се повтарят, без да предизвикат обсъждане, преразглеждане или загриженост.

Липсата на правилно етично поведение има огромно влияние върху развитието на организацията. Повтарящите се случаи на неволно или умишлено некоректно поведение, в резултат на което са се осъклили или провалили проекти, създават лош имидж на софтуерната фирма и застрашават нормалното й функциониране и дори съществуване.

Друго следствие от липсата на утвърден етичен правилник е деформирането на компютърния специалист като личност. Сблъсквайки се с конкретна ситуация, той взема решения на основата на собствената си ценностна система и морал, понеже неговата организация не е осигурила насоки за приемливо етично поведение. След няколко подобни случая той може и да не разглежда поведението си като неетично или непрофессионално.

Едно възможно решение е в софтуерната организация да се обсъждат проблемите от морално-стично естество и да се приемат правила за етично поведение. Те не могат да бъдат задължителни, но поне биха определили приемливо поведение. Отклонението от него би било въпрос на личен избор и отговорност на индивида. Осъзнавайки значението на нормите за етично поведение, в [8] е предложен следният кодекс на работещите в областта на софтуерните технологии: Кодекс за етично поведение

1. Ще се въздърjam от нерегламентирано обсъждане на проекти и работа, свързана с други — моите мениджъри, клиенти, членове на екипа и др.
2. Няма да поемам задължения към мои клиенти, други професионалисти и или мениджъри, които не мога да изпълня,
3. Няма да поемам задължения от името на други професионалисти без предварителна и пълна консултация с тях.
4. Няма да се съгласявam с изисквания, крайни срокове и др. без обяснения за риска, без допълнителна информация и без документиране на всички въпроси, засягащи преговарящите страни — клиента, мениджъра и други професионалисти.
5. Уважавам правото на другите да преговарят за промяна на изисквания и ограничения (цени, условия, срокове и др.).
6. Моите партньори трябва да признават правото ми да променям цени, срокове, ресурси и качество, след като те променят изискванията.
7. Имам професионалното право да казвам „не“, давайки подробни обяснения и без да пораждам взаимни обвинения.
8. Няма да поставям мои лични или професионални интереси над тези на мой клиент или организация.
9. Ще отчитам свършената от мен работа честно и добросъвестно.
10. Ще се старая да се отнасям с останалите като с професионалисти, дори и те да не се отнасят така с мен.
11. Няма да разкривам каквато и да е информация за дейността на мой клиент, която съм получил при работата си с него, без изричното му разрешение.
12. Ще се старая да бъда пример за навлизашите в професията и да дискутирам с тях проблемите за етично поведение.
13. Ще се противопоставям на всички нарушения на този кодекс от други професионалисти.

Разбира се, такъв кодекс не може да бъде задължителен, но се препоръчва спазването му от всички софтуерни специалисти, които го приемат и смятат, че е важно спазването на определен!: морално-етични норми и при изпълнение на софтуерните дейности. За важността на проблема говори и фактът, че най-големите професионални организации, свързани със софтуера — ACM и IFIP — от няколко години имат етичен кодекс за своите членове.

Литература:

1. De Marco, T., T.Lister, Peopleware. New York: Dorset House, 1987.
2. Paulk, M. et al, Capability Maturity Model, SEI, Carnegie Mellon University, Pittsburg PA, 1994.
3. Curtis, B. et al, People Management Capability Maturity Model for Software, SEI, Carnegie Mellon University, Pittsburg, PA, 1993.
4. Parish, P., Recruitment Sources. IN W.F.Cascio, Human Resource Planning, employment and Placement. Washington DC, 1989.
5. Fujigaki Y., Stress Analysis, in Special Issue on Peopleware, American Programmer, vol. 6, No 7, 1993.
6. Weinberg G.M., The Psychology of Computer Programming. Van Nostrand Reinhold, 1971.
7. Adler R., Communicating at work. McGraw-Hill, Inc., 1989.
8. Thomseff R., Crossing the line Professionalism and Software Ethics, in Special Issue on Peopleware, American Programmer, vol.6, No 7, 1993.

15. МАЛКАТА СОФТУЕРНА фирмА

Всяка от главите дотук се смята за съществена част от всеки уведен курс по софтуерни технологии. Биха могли да се добавят (според предпочтанията на различните автори) и малък брой други важни теми. Решихме обаче да завършим тази книга с една глава, която да представлява интерес от гледна точка на реалната обстановка днес в България и да съответства на определени аспекти на най-често срещания сценарий, когато някой реши да започне самостоятелен бизнес в областта на софтуера. Знае се, че в софтуера навсякъде по света преобладават малките предприятия. В огромното мнозинство от случаите по света и още повече у нас софтуерният бизнес започва със създаването на малка софтуерна фирма. Поради тази причина интерес представлява един аналитичен поглед в генезиса и еволюцията ѝ

15.1. Еволюция на малката софтуерна фирмА

15.1.1. Основа на изследването

В световната литература са известни многообразни изследвания, посветени на малкото предприятие. Специфичните за софтуерната индустрия обаче са все още изключително малко. В това отношение [1] е пионерско изследване. Макар да се основава на конкретни данни от Италия, то има голямото достойнство да обхваща твърде представителен период от около 15 години, естествено, при всички възможни отклонения в течение на годините — закрити или изчезнали фирми, ръководители, загубили желанието си да отговарят на поставените въпроси, сменени имена или предмет на дейност. За българския софтуерист е от значение и фактът, че изследваните фирми са от Южна Италия, част от страната която по степен на развитие (а защо не и донякъле по менталитет) е по-близка до България. Следователно резултатите и изводите от това изследване биха били твърде полезни и поучителни за пас.

15.1.2. Типове ноу-хау на стартиращата софтуерна фирмА

От особено значение за бъдещото развитие на стартиращата софтуерна фирмА е ноу-хауто, с което тя започва. Един малко по- внимателен поглед показва, че това ноу-хау се свежда до ноу-хау от съседна област, а след още малко анализ се вижда, че то е от 4 основни типа:

— **базово** ноу-хау (група 1), което включва научни и технически знания и умения, безусловно необходими за владеенето на технологиите на сектора; това ноу-хау се овладява в университетите, научноизследователските центрове, големите електронни или информационни фирми, фирми потребители на мащабни електронни обработки на данни;

— *свързано* ноу-хау (група 2), което включва дълбоко познаване на нуждите от хардуер на потребителите в сектора или на софтуерните им проблеми: такова ноу-хау се придобива най-добре в консултантски фирми по въпросите на бизнеса и на електронната обработка на данни, във фирмии доставчици на хардуер и информационни услуги;

— *пазарно* ноу-хау (група 3), което включва знание за тенденциите в търсениято и на характеристиките на разпространителските канали; това ноу-хау човек може да придобие в резултат от работа като продавач на софтуер и хардуер;

— ноу-хау за *потребителя* (група 4), което включва знанието, необходимо за да се доведе софтуерният продукт до организацията потребител; такова ноу-хау започва с детайлното познаване на нуждите на крайния потребител (фирма, огдел, тип бизнес) и на умението да се установи съответствие между тези нужди и най-подходящия за целта софтуерен продукт.

Какво ноу-хау имат стартерищите софтуерни фирми според изследванията? Условно могат да се разгледат три периода — инкубационен, начален и консолидационен. За Италия те са били от 1978 до 1984 година с нарастваща продължителност (1, 2, 4 години). Трудно е да се каже как във времето тази схема следва да се пренесе към България, но със сигурност тя е изместена най-малко с 10 години и е повече разтеглена. Разпределението е дадено в таблица 15.1.

Тип ноу-хау	Инкубация	Начало	Консолидация	Общо
Базово (група 1)	25.71%	40.00%	36.36%	34.40%
Съвързано (група 2)	25.71%	22.86%	16.36%	20.80%
Пазарно (група 2)	20.00%	22.86%	30.92%	25.60%
За потребителя (група 4)	28.58%	14.28%	16.36%	19.20%

Табл. 15.1. Разпределение на типовете ноу-хау

От таблицата веднага могат да се направят някои интересни изводи.

1. Сравнително по-малка част от фирмите са се развили на пазара с ноу-хау за потребителя. При това с времето относителният им дял намалява. Едно правдоподобно обяснение е, че владеенето само на това ноу-хау се оказва твърде недостатъчно.

2. Забелязва се нарастване на значението на ноу-хауто за пазара. Това отразява общата тенденция познаването на пазарните механизми и умението им те да се използват ефективно да е не по-малко важно от създаването на технологично качествени продукти.

3. Независимо от горното обаче все пак най-голям е делът на фирмите, които са старили с базово ноу-хау. Още по-показателно е, че техният дял във времето нараства. Това отразява факта, че при изключителната динамика на областта, изменяща се техника и софтуерни инструменти, най-адаптивни си остават тези, които имат фундаментални знания и умения.

15.1.3. Опростена схема на развитие на малката софтуерна фирма

По принцип малките фирми преминават през два етапа [2] — *технологично ориентиран* и *пазарно ориентиран*. Технологично ориентирианият е фокусиран навътре във фирмата — изграждат се и се развиват основните технологии, оценяват се силните и слабите им страни. В следващия — пазарно ориентирианият етап

— фирмата се съсредоточава върху пазара и върху методите за управление и планиране. Преходът от първия към втория етап протича по различен начин в големите и в малките фирми. В големите се осъществява смяна на организационната схема (създават се нови функции и отговорности, нова йерархия, нови комуникационни връзки). В малките фирми такъв процес е невъзможен най-вече поради липса на ресурси за това и поради по-слабата организационна структурираност. Там този преход е много повече функционален и персонален, отколкото цялостно структурен. Възможно е да се предприеме диференциране на функциите, преразпределение на отговорностите, известно пренараждане на дейностите на фирмата и техните приоритети. Общо взето, този процес не следва строги процедурни правила както при големите фирми, а е по-скоро евристичен, с проби и грешки. Специално за софтуерните фирми нещата се усложняват от високата степен на несигурност и трудна предсказуемост поради вече споменатата динамика. При това положение важна роля започват да играят фактори от рода на творческата способност и интуицията на водещите лица във фирмата.

Оказва се обаче, че моделът с тези два етапа е твърде опростен и че в реалността нещата често са доста по-комплицирани. Едно по-сложно моделиране води до по-адекватни резултати и показва една много поучителна и полезна картина.

15.1.4. Един по-диференциран модел на развитието

Начална точка на този модел е опитът да се идентифицират и групират ресурсите, които са определящи за еволюцията на малката софтуерна фирма. Определени са 16 такива и те са класифицирани в следните 3 групи:

— *ресурси, свързани с предприемача* — индивидуално ноу-хау, опит, лични връзки, ангажираност с процесите на софтуерното производство; това са все Ресурси, относящи се до основния опит на предприемаческата група и до нейното включване в технологическите и управленските аспекти на фирмата;

— *ресурси, свързани с професионалистите* — технически умения, професионална квалификация, разнообразие от компетенции: тези ресурси са свързани с характеристиките на специалистите в софтуерната фирма — колкото те са по-разнообразни и задълбочени, толкова по-благоприятно се отразяват на стабилността и конкурентоспособността ѝ;

— *ресурси, свързани с организацията и технологията* — размер на фирмата, обвързаност с други фирми и лица, прилагани методологии; тези ресурси засягат технологическия и маркетинговия опит на фирмата, наличието на установени организационни схеми и технологии, структурирани връзки с външния свят.

На основата на така идентифицираните и групирани ресурси става възможно дефинирането на 7 организационни конфигурации. Те са следните:

1. *C1 — конфигурация, основана на технологическата насоченост на предприемаческата група.* Предприемачът е преди всичко специалист с тех-нологическо ноу-хау, придобито в други фирми, научни центрове или университети. Основната част от времето и вниманието си посвещава на разработването на софтуерните продукти. Останалите му знания и умения са незначителни. Организационната структура на фирмата е неформална. Работата по същество се извършва от една или няколко софтуерни групи.

2. C2 — конфигурация, основана на професионалисти. Тя се характеризира с технологична и маркетингова компетентност на специалистите и със сътрудничество с външни консултанти, които работят за фирмата на непълно работно време.

3. C3 — конфигурация, основана на връзки (сътрудничество с по-големи фирми за софтуер, хардуер и услуги, с университети и научни центрове). Конкурентоспособността на такива фирми разчита на взаимоотношения с външни организации. Такива връзки позволяват на малката фирма да се усъвършенства в организационно и управленско отношение.

4. C4 — конфигурация, основана на разгърнатата организация и методологии за разработка. Тази конфигурация се отличава с ясно изградена организационна структура, превъзхождаща организационната форма, фокусирана единствено около предприемача. Както организацията, така и използваните технологии се основават на стандарти и на формализирани процедури.

5. C5 — конфигурация, основана на пазарни взаимоотношения. За такава фирма конкурентоспособността е резултат на приложението на маркетинговото ноу-хау на предприемача, благодарение на което тя съумява да реагира бързо на нуждите на пазара.

6. C6 — конфигурация, основана на интегрирани и специализирани продукти или услуги. Конкурентоспособността на такава фирма се базира на възможностите ѝ да отговаря на нуждите на пазара по отношение на допълнителни услуги към даден основен продукт. Обикновено се покрива местният пазар.

7. C7 — конфигурация, основана на комерсиализацията на системи. Такава конфигурация се характеризира с няколко дейности: комерсиализация на хардуер и софтуер, обучение на клиентите, консултации. Обикновено разработването на софтуер е само допълнителна поддържаща дейност към вече изброените.

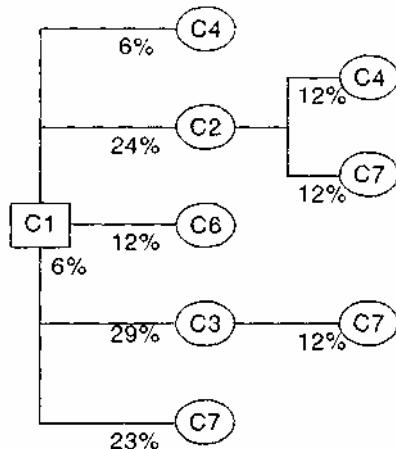
Изследванията на реално създадените и развили се малки софтуерни фирми показва, че те започват развитието си от две от горните конфигурации:

- C1 — конфигурация, основана на технологическата насоченост на предприемаческата група;
- C5 — конфигурация, основана на пазарни взаимоотношения. След това те еволюират на една или две стъпки до други от седемте посочени конфигурации.

15.1.5. Еволюция на конфигурация C1

Оказва се, че фирма, започната от конфигурация C1, може да се развие през всички останали конфигурации, с изключение на C5. Възможно е дори да остане в началното си състояние. Интересно е обаче да се знае, че в края на еволюцията се достига само до 2 конфигурации — C4 и C7. Поучително е също да се види какъв е пътят до тези 2 крайни състояния и какво е съотношението между тех, т.е. кое е най-вероятното развитие на малката софтуерна фирма, която се организира от лице или лица с технологична компетентност.

На фиг. 15.1. са отбелязани възможните пътища, както и разпределението им в проценти. Следвада отбележим, че 6% остават в началното състояние C1.



Фиг.15. 1. Еволюция на конфигурацията C1

Макар че от фиг. 15.1. достатъчно ясно може да се проследят възможните пътища на развитие на фирма, започнала от конфигурация C1, ще отбележим още следното.

Преминаването от C1 към C4 може да е директно (6%) и тогава то е свързано обикновено с увеличаване на броя на служителите и с въвеждане на стандарти.

По-често обаче се преминава към конфигурация C2 (24%). Този преход се характеризира със стратегия за разнообразяване на продуктите и съответно разширение на професионалната компетентност посредством включване на нови специалисти с разнообразна квалификация.

От C2 като втора стъпка има два пътя. Първият води отново до C4 (12%) обикновено чрез въвеждане на CASE средства, които сами по себе си до голяма степен стандартизират технологическите процедури. Вторият път пък довежда до конфигурация C7 (12%) и се постига с прилагането на стратегия на комерсиализация. При нея над 50% от приходите на фирмата идват от продажба на готов хардуер и софтуер, а по-голяма част от останалото — от консултации.

Следващата възможна първа стъпка не е много срещана и е преминаване към C6 (12%). Характеризира се с разширяване на мрежата от клиенти и задълбочаване на взаимоотношенията с тях, най-вече чрез оказване на поддържащи услуги. Този преход изисква задълбочаване на маркетинговите познания за конкретни сравнително тесни пазари.

Рядко срещана алтернатива е запазване на статуквото (6%) – фирмата остава от тип C1 за по-дълго време. Това може да означава все пак, че след такъв по-продължителен период се преминава към някоя от другите конфигурации по общата схема за първа стъпка.

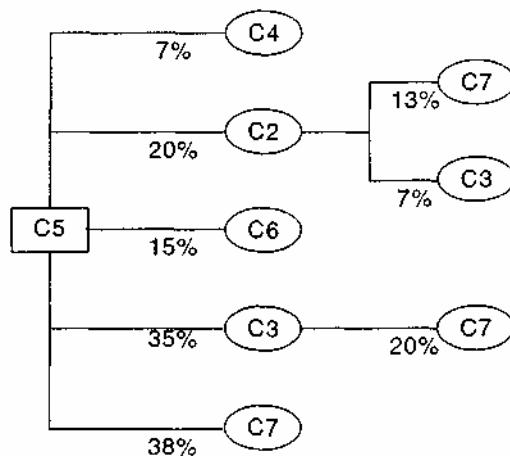
Най-много фирми (29%) предпочитат да преминат към конфигурация C3. Те се свързват с големи хардуерни фирми и стават техни представители в една или друга форма. Друга стратегия е осигуряване на специализирани ресурси за такива фирми, респективно оказване на услуги в определени тесни области. Оттук форсирano, т. е. за всичките тези 29%, се извършва преход към конфигурация C7 по начина, описан вече с прехода от C2 към C7.

Последният възможен директен преход е към C7 (23%). Схемата за преминаване вече описахме по-горе като втора стъпка (от C2 към C7). Възможно е обаче такава стъпка да е принудителна — ако напусне предприемачът, който е бил софтуерно ориентиран. Тогава производството на софтуер по необходимост минава на втора линия като

поддържаща дейност, а на преден план излизат продажбите на готов хардуер и софтуер и консултациите.

15.1.6. Еволюция на конфигурация C5

На фиг. 15.2. са отбелечани възможните пътища с начало конфигурацията C5, както и разпределението им в проценти.



Фиг. 15.2. Еволюция на конфигурацията C5

Тук също накратко ще отбележим някои особености.

Преминаването от C5 към C4 (7%) не е твърде характерно. Стандартизацията се олицетворява обикновено от въвеждането на CASE среда и други технологически стандарти, разнообразяване на маркетинговата стратегия и естествено налагашо се при такива мерки увеличение на персонала.

От C5 може да се премине и към C2 като първа стъпка (20%). Това означава придобиване на допълнителна квалификация на специалистите (чрез обучение или назначаване на нови хора) и като следствие от това — възможност за разнообразяване на продуктите и инвестиции в разработването на нови продукти. C2 обаче е междинен стадий. Оттук пътищата са два — към C7 (13%) чрез мерките, описани вече или към C3 (7%) чрез обвързване с големи хардуерни фирми и сътрудничество с тях в различни форми — представителство и/или работа за тях.

Много често срещан преход е към C3 (35%). Току-що видяхме как става това. От C3 в повечето случаи сравнително бързо се достига до крайната конфигурация C7 (20%), но доста често (15%) фирмата се задържа по-продължително в това състояние.

Най-често от C5 се преминава директно към конфигурация C7 (35%).

15.1.7. Извод

От последните разглеждания се вижда, че независимо от коя начална конфигурация се стартира — технологично ориентирана (C1) или пазарно ориентирана (C5) — на една или на две стъпки, малката софтуерна фирма достига до две възможни конфигурации. Първата е C4, което означава превръщането ѝ в добре структурирана и управлявана производителка на софтуер на основата на стандартизиран и формализиран технолошки, с ясен поглед върху пазара, мястото на фирмата в него и съответна

маркетингова стратегия. Втората е С7, характеризираща се с относителното изоставяне на производството на софтуер за сметка на продажби на хардуер и софтуер и консултационна дейност. Без да се абсолютизира този модел и произлизашите от него движения на малките софтуерни фирми, той може да е добра ориентация за тези, които се впускат в този тип бизнес.

15.2. Екстремно програмиране

15.2.1. Необходимост

На пръв поглед това, което следва, е в пълно противоречие със системните подходи, основани на сериозни теоретични модели, които бяха изложени в предходните глави. От друга страна, не можем да си затваряме очите за реалността, особено за тази в малките фирми. Когато реализацията на даден софтуерен проект лежи върху плещите на 10, а понякога и по-малко специалисти, едва ли е възможно педантично да се приложат организационните и технологичните схеми, предлагани от теорията. Не е трудно да се повярва например на следните данни: ако една софтуерна фирма има 1500 служители, а друга 150, усилията Да се реализира подобряване на софтуерните процеси на основата на СММ (вж. глава 8.) в никакъв случай няма да са 10 пъти по-малки във втората фирма спрямо първата [3]. Изненадата е може би в това, че като се изключи обучението (което е силно зависимо от конкретния брой), за останалите дейности **усилията в двете фирми не се отличават особено**. Въпреки това е добре известно, че много малки фирми създават отлични по качество софтуерни продукти, спазвайки договорените срокове и постигайки добра рентабилност. Разумният подход е тези факти да не се игнорират, а да се направи опит да се обяснят. Един от тези опити е парадигмата на т. нар. **екстремно програмиране**, понякога обозначавано с акронима XP (or Extreme Programming).

15.2.2. Корени

Този модел има за свой прототип **каскадния** модел на жизнения цикъл на програмния продукт (вж. 2.3.6.). В най-опростения му вариант може някои фази да се слеят и да се смята, че има 4 фази — анализ, проектиране, програмиране и тестване. Тези фази в някаква степен могат да се припокриват (например програмирането да започне преди окончателно да е завършено проектирането или пък, съвсем естествено, да има програмиране, след като е вършено тестване). Твърдо се приема още, че потребителят дефинира своите изисквания в началото и после повече не ги променя. И в това е първата неадекватност на този модел. Практиката отдавна е показвала, че почти не се срещат сериозни възложители, които да не променят малко или повече изискванията си по време на процеса на разработване на програмния продукт. За да отразят тази особеност, теоретиците предложиха **прототипния** модел (вж. 2.3.7.). При него до известна степен се допуска повтаряне на някои от фазите (итерации), но не толкова заради променящите се потребителски изисквания, колкото с цел по-ранното откриване и преодоляване на неуспешни технологични решения (например непостигане на желаното време за отговор, неудовлетворителни от ергономична гледна точка интерфейси и т. н.). Последните две стъпки, приближаващи ни към XP, са **еволюционният** (вж. 2.3.8.) и **спиралният** (вж. 2.3.9.) модел. При първия особено се набляга на непрестанно променящите се потребителски изисквания, при втория — на непрекъснатото оценяване на риска. И двата обаче са предназначени да отразяват разработването на големи или относително големи

проекти и по необходимост се налага да се предвиждат съответните на машабите организационни и рискови аспекти.

15.2.3. Същност

Постепенно някои от разработчиците решават, че при относително малки проекти могат да доведат тези итерации до крайност, т. е. да повтарят четирите горни фази много пъти, като очевидно при това ги скъсяват до разумния минимум [4]. Какъв е този разумен минимум, естествено, не може да се каже с абсолютна точност, но един достатъчно конструктивен отговор гласи: дни.

Основополагащият принцип е при всяка такава итерация да се подберат едно или малък брой от най-необходимите за потребителя *средства* (наричани в XP stories — не превеждаме този термин буквално, защото не го намираме за подходящо) сред всички поискани. Този избор се прави на основата на предполагаемата цена и бързина на реализация и на ценността от гледна точка на потребителя. По-просто казано, избират се за реализация най-напред тези средсва (функции), които ще свършат най-много работа на потребителя, а от друга страна, ще бъдат направени качествено за най-кратко време.

Доколкото говорим за *итерации*, ясно е, че веднага щом дадена итерация приключи с предаване на разработените средства на потребителя, започва следващата с избор на най-подходящите средства от останалите. Много е важно да се знае, че при този избор решаваща дума има клиентът (възложител, потребител), но на основата на информация, предоставена от разработчика. За програмистите остава да направят необходимите технологични стъпки — да декомпо-зират поисканото средство (услуга, функция) до необходимия брой задачи, да ги реализират, тестват и предадат за ползване на клиента.

Лесно се вижда, че при този подход четирите фази остават, естествено, в много кратки отрезъци от време.

Анализът се състои точно в този подбор на поредните средства за реализиране. Съществено е, че всяко нещо, което този анализ определи като такова средство, трябва да бъде ясно ориентирано към нуждите на потребителя, да подлежи на тестване и усилията за реализацията му да могат да бъдат отделени и оценени.

Проектирането и програмирането се движат практически едновременно. В момента на определянето на средствата от поредната итерация се извършва декомпозиция и планиране. Това означава, че точно определен брой програмисти получават точно определени задачи със също така точно определени срокове. Тъй като става въпрос за малки и ясни задачи, определянето на тези срокове не би следвало да е трудно, особено след първите итерации. Докато програмистите работят — всеки по своята задача, потребителят, който продължава да е активен участник, създава функционални *тестове*. Тук има още една характерна и малко странна особеност — програмистите работят по *двойки* върху дадена задача на един компютър. Това се свързва с необходимостта да се осигури валидност на създадения програмен модул при липсата на стандартните процедури. Последната е породена от съкратеното време и намалените други ресурси. Предвидена е възможността за контакт на двойката с потребителя по въпроси на функционалността или пък с най-опитните програмисти по технологични въпроси, но винаги в рамките на не повече от] 5 минути.

В този процес на реализация почти веднага навлиза и *тестването*. При това има още една особеност: по логически причини тестването следва програмирането, но самите *тестови примери* се пишат предварително. Както е обично, при почти всеки тест се изявяват различни грешки, които следва да бъдат отстранени. При това винаги приоритет се дава на конструктивността, по-точно:

- ако двойката програмисти вижда „чист“ начин за отстраняване на грешката, тя го прилага незабавно;
- ако вижда „груб“ начин и заедно с това се досеща и за ново проектантско решение, което би довело до „чисто“ решение, то се прилага „чистото“ решение с препроектирането;
- ако вижда само „груб“ начин без друга алтернатива — прилага се този „груб“ начин.

На всеки програмист е ясно какво се разбира тук под „чисто“ решение — то е съобразено със запазване на всички добри качества на програмата — прегледност, ясна структура, способност за лесно модифициране, запазена простота и т. н. При втората алтернатива се предпочита все пак „чистото“ решение, въпреки че то изисква повече време и усилия. С това се осигурява обаче по-лесното съпровождане на продукта в бъдеще.

Следва да се отбележи изрично — макар потребителят да задава свои функционални тестови примери, тестването се извършва от самите програмисти. Смята се, че макар това да елиминира независимото тестване, при възприетия модел и при относително малкия обем на решаваните задачи това не води до отрицателни последствия по отношение на валидността на разработвания софтуер.

15.2.4. Съвсем практичесии аспекти

Дори при най-добре планираните разработки и добре комплектовани колективи могат понякога да възникнат проблеми.

Един не толкова рядко срещан случай е **надценяването** на собствените (на колектива) възможности. Идва момент, в който става ясно, че сроковете няма да бъдат спазени. Ако този момент е все още достатъчно отдалечен от крайния срок, възможна е оценка на прилаганите процедури и евентуални промени в тях.

Може обаче да се окаже, че е късно за дадения проект да се правят резултатни промени. Няма друга възможност, освен изпълнителя да се обърне към възложителя и да поиска някаква форма на облекчение — удължаване на срока, временен отказ от определени планирани за реализация функции, отлагане на дадено средство за следващата итерация. Във всички случаи отрицателните страни на такъв подход са по-добро решение, отколкото сляпата надежда за благоприятен изход в последния момент.

Друг доста често срещан случай са клиентите (възложители), които не са склонни твърде да общуват с разработчика. Те не искат да определят средствата за реализация в поредната итерация, нито пък да създават тестови примери. На пръв поглед това не е никаклошо, но опитният софтуерист знае много добре, че в един момент спестеното време и неприятни моменти от **липсата на комуникация с възложителя** ще му се стоварят в двоен и троен размер, когато клиентът започне да експлоатира продукта и реши, че не е получил това, което си е представял. Ясно е, че всякакви промени на този стадий са много по-трудни (понякога невъзможни) и по-скъпи. Обикновено опитният изпълнител се е погрижил чрез подписания договор да няма никакви задължения за промени в такъв късен момент. Но дори само усилията да убеди клиента в правотата си са достатъчно неприятни и отнемат време. Да не говорим, че ефектът от един неудовлетворен потребител (макар и изключително по негова собствена вина) е във всяко отношение отрицателен за разработчика.

Следователно разработчикът просто е длъжен във възможно най-ранен момент да убеди възложителя да му сътрудничи. Във всички случаи, но особено при XP е абсолютно недопустимо по време на разработката програмистът да се упова на

собствените си догадки вместо на ясно изразените изисквания на възложителя. Вероятно най-доброто решение е да се обяснят максимално нагледно на възложителя тежките последици от нежеланието му да участва в кръга на своята компетентност в разработката. Ако дори тогава няма резултат, изводът е, че сигурно тази разработка не си струва усилията.

Друг важен практически проблем е *текучеството*. За България този проблем съдържа и известен момент на безвъзвратност, доколкото движението на програмистите към чужбина е особено интензивно и не толкова често веднъж заминалите се връщат. А ако се върнат, това е знак, че може би програмистът не е чак толкова добър специалист. Впрочем известен „оборот“ на членовете на колектива действа освежаващо. Ръководителят на екипа обаче винаги следва да се интересува от мотивите на напускация. Ако те са свързани с по-добри материални условия, каквито една малка софтуерна фирма по-трудно може да осигури, това не е толкова тревожно, най-малкото едва ли може да му се противостои. Ако обаче мотивите на отиващия си се дължат на нисък професионализъм във фирмата, на недобра атмосфера, на лоша организация, на неудовлетвореност от равнището на разработваните проекти, това би следвало да бъде сигнал за сериозния ръководител. Естествено, тук стои въпросът и доколко един напускащ програмист може да навреди на фирмата, отнасяйки със себе си определена информация, натрупана по време на работата му там. От една страна това, че се работи в двойка, запазва почти напълно възможностите на колектива безболезнено да продължи работата си. От друга страна, са общите проблеми от опасностите от изнасянето на информация навън от фирмата, но това е толкова общ проблем, че разискването му тук не е смислено.

15.3. Правни проблеми

15.3.1. Необходимост от специализирана правна култура

След като видяхме какво е най-вероятното развитие на една новосъздадена малка софтуерна фирма и разгледахме екстремното програмиране (ХР) като една подходяща възможност за организиране на разработването на определени типове програмни продукти, не може да не отбележим още едно нещо, без което ръководителят (собственикът) на такава малка фирма не може да има сериозни резултати. Става въпрос за една минимална правна култура, свързана със софтуера. Дори придобиването на минимални знания обаче в тази област според учебните планове по софтуерни технологии във водещите университети в САЩ [7] изискват от един до три специални курса. Поради това тук само ще набележим основните направления, по които въпросният ръководител следва да придобие ориентация и знания.

Необходимо е да се знае, че въпросите за разработването, разпространението, съпровождането и разширяването на програмните продукти в правото са предмет на разглеждане от теорията, съответно законите, свързани с интелектуалната собственост. При това положение разработчиците на софтуер трябва да са наясно с правата и отговорностите, произтичащи от тях. Като следствие идват договорите, които се сключват във всички изброени току-що случаи. Несъмнено софтуеристите трябва да познават стандартните лицензионни практики, а така също в какви случаи какви отговорности носят за евентуални вреди, които разработеният и/или разпространяван от тях софтуер е причинил. Както вече се каза, нито в следващите няколко страници, нито дори в рамките на един или няколко курса е възможно достатъчно дълбоко и всестранно усвояване на тази материя. Тя просто изиска юридическо образование. Това, което се

опитваме да постигнем обаче, е да убедим софтуеристите, че има немалко случаи, които те трябва да са в състояние да идентифицират и в които няма друг начин, освен да се обърнат за намиране на решение към юрист — тесен специалист в областта на софтуера.

15.3.2. Закони за интелектуалната собственост

Основната цел на законите за интелектуална собственост е да насърчават разработването и разпространението на инновации, така че те да се ползват от обществото. Ясно е, че създаването им (както и сътворяването на произведения на изкуството и литературата) обикновено изисква инвестирането на време, енергия и други ресурси от страна на знаещи, умеещи и надарени хора. За да окуражат подобни дейности, законите за интелектуална собственост предвиждат възможността да се придобиват изключителни права за контрол върху търговската експлоатация на инновационната (или артистична) творба за определен период от време. По този начин, от една страна, се защитават интересите на творците. От друга страна, и обществото има полза, защото получава възможност за достъп до съответното творение при определени условия, а по-късно след изтичането на определения законов срок — и напълно свободно.

Има случаи, когато дадено интелектуално произведение *не попада* в рамките на действие на закона и остава за напълно свободно ползване от всекиго. Това става при определени условия, например:

- творението не отговаря на всички изисквания на закона;
- този, който може да претендира за съответните права, решава, че по някаква причина не желае да го направи (в областта на софтуера такива случаи са добре известни и никак не са редки);
- не са спазени сроковете или други процедури за предявяване на искане;
- изтекъл е срокът на защита, предвиден от закона.

По принцип даден закон за интелектуална собственост съдържа няколко елемента:

1. Определение за обектите, за които се отнася.
2. Множество от изисквания, които следва да са изпълнени, за да се получи съответната защита — например какви характеристики следва да има произведението, за да се смята за творческо, или пък кой е този, които може да има претенции, или пък какви процедури трябва да се изпълнят и др.
3. Множество от права, които изключват другите субекти от определени дейности.
4. Процедури, които позволяват да се установи има ли нарушения на тези права.
5. Процедури, които се прилагат при установено нарушение.

15.3.3. форми на правна защита на софтуера като интелектуално произведение

От няколко десетилетия юристите усилено се опитват да намерят най-подходящата правна форма за защита на софтуера. Това се прави и в България от около 20 години насам [5]. Първоначално тенденциите са били да се прилага *патентното право* въпреки значителните проблеми, свързани с идентифицирането на софтуера като съответстващ на съществуващите изисквания за патентоспособност. Особени трудности предизвиква връзката на изобретенията (основния па-тентоспособен обект) с техниката. В българския закон от 1993 година [8] например основната характеристика „новост“ се определя чрез понятието техника. Независимо от това има прецеденти от 70-те и 80-те години в САЩ, когато в специфични случаи програми (например за управление на производството) са

получавали патент. През 1977 г. авторът на тези редове заедно със съавтори също успя да получи 5 авторски свидетелства за изобретения (по току-що цитирания български закон те могат да бъдат трансформирани в патенти) за специализирана програма, след като обаче алгоритмите й бяха преобразувани до вид на схеми на техническо устройство, т. е. необходими бяха ред неестествени действия.

Междувременно с появата и развитието на персоналните компютри и изменението на формите и мащабите на разпространение на софтуера юристите започнаха да намират за все по-подходящо *авторското право* и да се оттеглят от патентното право. В българското право това е доведено до изричен текст в Закона за патентите:

„Чл.6. (2) Не се считат за изобретения:

5. програми за електронноизчислителни машини
6. представяне на информация."

За пълнота следва да отбележим, че е имало един период, в който са полагани значителни усилия за създаване на специално право (*sui generis*) за софтуерните продукти. Оказалось се е обаче, че такъв опит ще се сблъска с непреодолими теоретически и практически трудности и затова постепенно е бил изоставен.

Преди да се спрем малко по-подробно на авторскоправната закрила, ще отбележим и още една правна форма на защита — тази на *марките*. Законът за марките и географските означения [9] от 1999 година дава възможност да се ползва специфична защита, отнасяща се преди всичко до търговската дейност, свързана с разпространението на софтуера, както и със съпровождащите го услуги.

В края на 15.2.4. се повдигна въпросът за проблемите, които може да предизвика напускането на фирмата от страна на даден специалист, запознат с малко или повече фирмени тайни. Сериозни правни средства за защита в това отношение може да предостави *Законът за защита на конкуренцията*.

15.3.4. Защита на софтуера чрез авторското право

Вече казахме, че в момента най-пълна и най-адекватна защита на софтуера предоставя Законът за авторското право и сродните му права [6].

Сред *закриляните обекти* в чл. 3 (1), т. 1 са посочени изрично и *компютърните програми*. Те попадат под общата дефиниция за обект на авторското право, която включва „всяко произведение на литературата, изкуството и науката, което е резултат на творческа дейност и е изразено по какъвто и да е начин и в каквато и да е обективна форма“.

Друго важно понятие е *автор*, който чл. 5 определя като „физическо лице, в резултат на чиято творческа дейност е създадено произведение“. Не трябва обаче да влизаме в заблудата, че този автор е непременно и винаги носител на авторските права. В същия член веднага се пояснява, че „други физически или юридически лица могат да бъдат носители на авторското право“.

Последното пояснение е изключително важно за софтуера. То е развито в чл. 14, посветен специално на програми и бази данни и гласи: „Ако не е уговорено друго, авторското право върху компютърни програми и бази данни, създадени в рамките на *трудово правоотношение*, принадлежи на работодателя.“ Както ръководителите (собственици) на софтуерни фирми, така и разработчиците трябва да са наясно с правата си, произтичащи от този текст. Те следва също така да знаят, че разработването на софтуер може да се възлага и под други форми — не само в рамките на трудов договор. Такава възможност е *поръчката* и тогава според чл. 42 авторското право принадлежи на

автора (ако договорът не гласи друго), но пък поръчващият има право да използва произведението без разрешение от автора за целта, за която е било поръчано. Важно е също да се знае, че ако е постигнато споразумение в рамките на трудовото правоотношение за друго разпределение на авторските права, то това трябва да бъде оформено документално във валидна правна форма. Има и още един текст, чиято цел е да възстанови справедливостта при определени случаи. Това е чл. 41 (3), според който: „Когато трудовото възнаграждение на автора по времето, през което е създадъл произведението, се окаже явно несъразмерно на приходите, реализирани от използването на произведението, авторът може да поиска допълнително възнаграждение. Ако не се стигне до съгласие между страните, спорът се решава от съда по справедливост.“

Доколкото в последния текст се споменаха бази данни, следва да се отбележи, че в чл. 11 (1) се определя и техният статут — „авторското право върху сборници, антологии, библиографии, бази данни и други подобни принадлежи на лицето, което е извършило подбора или подреждането на включените произведения и/или материали, освен ако в договор не е предвидено друго“.

Един от най-съществените въпроси е този за ползването на произведението. В чл. 23 се третира въпросът за свободното ползване. Изрично е посочен специалният режим, отнасящ се до програмните продукти: „*Без съгласие на автора и без заплащане на възнаграждение се допуска.... 2. Използването на части от публикувани произведения или на неголям брой малки произведения, с изключение на компютърни програми и бази данни...*“ Този текст всъщност се отнася до ползването в други произведения и показва, че не е възможно, когато пишем собствени програми, да ползваме пасажи от чужди програми. От практическа гледна точка още по-важен е друг текст, чл. 25, също третиращ софтуера като изключение: „*Изготвянето на копия от вече публикувани произведения се допуска без съгласие на автора и без заплащане на възнаграждение само за лично ползване. Това не се отнася за компютърните програми и архитектурните произведения.*“

Смисълът на този текст е пределно ясен на всеки човек, който знае що е програма за компютър. Темата за пиратството е толкова дискутирана, предприеманите мерки срещу него така добре известни, че по-нататъшното ѝ обсъждане е безсмислено.

Колкото и да е странно за професионалистите, които знаят добре колко е животът на една програма, съгласно чл. 28а: „Авторското право върху компютърните програми или бази данни... продължава 70 години след разгласяването на произведението.“ Ако авторът е известен, прилага се общата разпоредба на чл. 27, която включва периода до смъртта на автора и 70 години след това. Съществени за практиката са текстовете от закона (чл. 36), които определят начините на използване на произведението. Според ал. 1: „*С договора за използване на произведението авторът отстъпва на ползвател изключителното или неизключителното право да използва създаденото от него произведение при определени условия и срещу възнаграждение.*“

По-нататък в същия член, в ал. 2, се изяснява смисълът на *изключителното* право — лишава автора от възможността да ползва сам произведенето или да отстъпва правата си върху него на трети лица — разбира се, само на уговорената територия и за уговорения срок и други условия. Тези ограничения не са в сила, когато предоставеното право е *неизключително* (ал. 3). Чрез ал. 4 се защитават правата на автора, в случай че той поради незнание или друга подобна причина не си дава сметка за разликата между предоставяне на изключително и неизключително право — предвидено е *изключителното право да се отстъпва само изрично и писмено.*

В закона има специален раздел (VII), посветен на използването на компютърните програми. Ако по законен път е придобито право за ползване на програма, с нея могат да

се извършват всички **действия, необходими за постигане на целите ѝ** (иначе казано, за това, за което е платено). Тези действия са изброени твърде подробно в чл. 70.

Казано е изрично (чл. 71) и какви **допълнителни действия** са позволени на законния притежател на правото на ползване без изричното съгласие па автора — създаване на резервно копие, изследване на свойствата на програмата и дори по-специализирани действия от рода на трансформиране на кода. Последното обаче е допустимо при определени ограничения, изброени в същия член и целящи да се предотврати използването на програмата или части от нея в друга програма.

Както беше упоменато в 15.3.2., законът предвижда мерки при нарушаване правата на авторите. От една страна всеки автор може да претендира за **обезщетение** от този, който му е причинил вреди, нарушивайки правата му по този закон (чл. 94).

От друга страна, нарушителят носи и **наказателна отговорност**. Специално за софтуера се отнася чл. 97 (1), т. 8 и 9, които гласят, че: „Който в нарушение разпоредбите на този закон:

8. ... притежава компютърна програма, като знае или има основание да предполага, че това е незаконно;

9. възпроизвежда, разпространява или използва по друг начин компютърна програма; ... се наказва с глоба *в размер от 200 до 2000 лева*, ако не подлежи на по-тежко наказание, и *предметът на нарушението*, независимо чия собственост е, *се отнема в полза на държавата* и се предава за унищожаване...“

От така разгледаните най-съществени за софтуера текстове в закона става ясно на каква защита могат да разчитат разработчиците на софтуер — ръководители и преки изпълнители, какви са правата им, какво трябва да спазват, когато ползват чужд софтуер, и какво рискуват, когато нарушият правилата.

Литература

1. Raffa M., Zollo G., Caponi R., The short but interesting life of small software firms. Achieving Quality in Software, Proceedings of the third international conference on achieving quality in software, 1996, Chapman & Hall, London, 1996, p.103—117.
2. Brandt S., Strategic planning in Emerging Companies, Addison Wesley, Reading (MA), 1981.
3. Kelly D.P., Culleton B., Process Improvement for Small Organizations, Computer, October 1999, Vol.32, No 10, p. 41—47.
4. Beck K., Embracing Change with Extreme Programming, Computer October 1999 Vol.32, No 10, p. 70—77
5. Ескенази, И. Правен режим на програмите за ЕИМ. Кандидатска дисертация. Софийски университет, София, 1980.
6. Закон за авторското право и сродните му права. Нова звезда, 2000.
7. Samuelson P., Deasy K. Intellectual Property Protection for Software, SEI Curriculum Module SEI-CM-14-2.1, Carnegie Mellon University, Software Engineering Institute, 1989.
8. Закон за патентите. Нова звезда, 2000.
9. Закон за марките и географските означения. Нова звезда, 2000.

ст. н. с. II ст. д-р Нели Милчева Манева ст. н. с. II ст. д-р Аврам Моис Ескенази

СОФТУЕРНИ ТЕХНОЛОГИИ

Корица *Михаил Руев*

Технически редактор *Симеон Айтров*

Компютърна обработка *Георги Георгиев*

Коректор *Даниела Славчева*

Българска. Издание първо. Излязла от печат 2001 г. Формат 70x100/16. Печатници 12,5

Издателска къща *Анубис*, 1124 София,

ул. *Младен Павлов* № 1,

тел. 443 503,944 16 43.

Търговски отдел 1124 София

ул. „*Никола Ракитин*“ № 2,

тел. 946 16 76, тел./факс 946 14 32

Печатница „Симолини“ — София

ISBN 954-426-311-X

