Глава 4

МЕТОДИ ЗА РАЗРАБОТКА НА ПРОГРАМНИ СИСТЕМИ

Развитието на изчислителната техника като единство на апаратни и програмни средства е свързано с определени тенденции. Отначало компютърните системи бяха наситени с "много" хардуер и "малко" софтуер и определящ фактор за цената бе стойността на апаратурата. Съвременните изчислителни системи са изградени от евтини интегрални схеми и са обезпечени с разнообразно програмно осигуряване на различни езикови нива. Цитират се данни, според които разходите за изграждане на програмно осигуряване надхвърлят 50% и достигат 60 - 70% от стойността на една изчислителна система. Разликата в тенденциите при развитието на хардуера и софтуера наложи представата за *софтуерна криза* и необходимостта от нейното разрешаване. Развиха се методики за повишаване ефективността на програмисткия труд от типа на модулното и структурното програмиране, обектно ориентираното и компонентното програмиране, тотално, възходящо и низходящо проектиране. Въведе се концепцията за *жизнен цикъл на програмните системи.*

4.1. ГРЕШКИ И НАДЕЖДНОСТ НА ПРОГРАМНОТО ОСИГУРЯВАНЕ

Една от проявите на софтуерната криза се свързва с оплакванията, че програмното осигуряване (ПО) е скъпо и ненадеждно [17]. Експлоатацията му се съпровожда с проява на грешки и това налага допълнителни разходи на средства за тестуване и съпровождане на програмните продукти. Интересно е да се прецизира представата за грешка при работа на ПО. Уточнението се налага, тъй като е възможно възложител и разработчик на ПО да влагат различен смисъл в това понятие, а те се намират от различни страни в процеса на разработка и експлоатация на програмните продукти. Следват някои определения за грешка на ПО според [17].

Определение 1: ПО съдържа грешка, ако поведението му не съответства на спецификациите за реализацията му.

Това определение не е изрядно, тъй като неявно се предполага коректност на спецификацията.

Определение 2: ПО съдържа грешка, ако поведението му не съответства на спецификациите при експлоатацията му за изменение стойностите на определени параметри в зададени граници.

Недостатък на тази дефиниция е, че ПО трябва да има адекватно поведение дори в случай, че стойността на даден параметър при изпълнение излезе извън зададените граници.

Определение 3: ПО съдържа грешка, ако поведението му не съответства на официалната документация, предоставена на потребителя.

Това определение също е некоректно. Възможна поява на грешки както в ПО, така и в съпровождащата документация.

Следната дефиниция [17] за грешка на ПО се представя като коректна и свободна от изброените недостатъци.

Определение 4: ПО съдържа грешка, ако то не изпълнява това, което потребителят разумно очаква от него.

Наличието на грешки в ПО се определя като неспособност на програмния продукт да действа в съответствие с изискванията на потребителя. Отказите при експлоатацията на ПО са проява на грешки при работата на програмите.

Надеждност на ПО съгласно [32] се определя като *вероятността за безотказна работа в течение на определен период от време.*
Надеждността на ПО не е само вътрешно свойство на програмите. Тя е свързана повече

с това как се експлоатира програмата или каква е вероятността потребителят да подаде набор входни данни, които да предизвикват отказ и авариен изход на програмата при изпълнение.

Надеждността на хардуера и софтуера се проявяват по различен начин като функция на времето.

Експлоатацията на хардуера в началния период се характеризира с висока честота на отказите. Следва дълъг период на експлоатация с ниско ниво на отказите, след което тяхната честота отново се повишава. Честотата на отказите в хардуера зависи от времето и не зависи от входните данни, с които се изпълняват програмните продукти.

Честотата на отказите в софтуера непрекъснато намалява. Това е вярно, ако изчистването на една грешка не води до вмъкване на нови грешки. Честотата на отказите в софтуера не зависи от времето в горния смисъл. Проявлението на грешките в една програмна система съществено се определя от входните данни.

Отрицателните последици от наличието на грешки при експлоатация на програмните системи поставят въпроса за отстраняване на грешките и създаване на надеждни програмни продукти. Важен препоръчителен извод

се дава в [17]. Усилията на разработчиците следва преди вичко да се насочват към анализ и отстраняване на причините за грешките и след това към разработка на ефективни методи за локализиране и отстраняване на грешки. В този смисъл най-общата трактовка изтъква, че единствената важна причина за грешките в ПО е *неп равилният превод на данни от едно представяне в друго.*

Преди да продължим, ще поясним разликата между понятията данни и информация в контекста синтез, анализ и проектиране на системи. Според същия източник данните са сурови факти, изолирани и независими. Те носят смисъл и значение, но сами по себе си не са полезни. Информацията представлява обработени и трансформирани данни с цел да бъдат полезни на потребителя. Информацията казва на хората нещо, което те не знаят или потвърждава нещо, което те подозират. Изобщо, информация се получава като резултат или функция от обработката на данни. INF = function(Data, Processing).

Възможно е информацията от един процес да послужи като данни за обработка при втори, следващ процес.

4.2. ЖИЗНЕН ЦИКЪЛ НА ПРОГРАМНА СИСТЕМА

От гледна точка на концепцията за преобразуване на данни (информация) от едно представяне в друго жизненият цикъл на една програмна система от идеята за нейното създаване през реализацията и експлоатацията и изхвърлянето й от употреба се описва със следните етапи [32]:

1. Дефиниране (специфициране) на проблем за решаване;

2. Изготвяне на задание;

3. Проектиране;

4. Програмиране;

5. Тестуване и настройка;

6. Демонстрация и предаване на клиента (краен потребител);

7. Експлоатация и съпровождане.

Създаването на програмно осигуряване се описва като последователност от процеси за преобразуване на данни, започващи от описанието (спецификацията) на задачата и

завършващи с работоспособна програма (инструкции за компютър) за решаването на задачата. Видно е, че програмирането е само етап при решаването на задачата, а програмното осигуряване се явява съвкупност от информационни елементи (данни и команди), описващи решението на проблема.

Постановката с етапи в жизнения цикъл позволява да се посочат източниците (причините) за грешките в ПО. Приема се, че създаването на ПО е съвкупност от процеси на транслация (трансформация) на данни от едно представяне в друго. Става дума за превод на първоначално формулирана задача в различни междинни решения до получаването на програма като набор машинни команди. Тогава причините за грешките, които се локализират в окончателния продукт се дължат на *непълен и/или неточен превод на данни от едно междинно представяне в друго* [17]. Ето защо грешките не винаги са вътрешно присъщи на програмите. Причините за грешките могат да се формулират като:

1. Неточности при преход от описание на проблема към формулиране на задание;

2. Неточности при преход от задание към изготвяне на проект;

3. Неточности при преход от готов проект към реална програмна система.

Жизненият цикъл на програмните системи може да бъде формулиран и в по-кратък вид [46] като последователност от следните етапи:

1. Системен анализ (Systems Analysis);

2. Системно проектиране (Systems Design);

3. Системна реализация (Systems Implementation);

4. Ситемна поддръжка (Systems Support).

Системният анализ е началният етап от жизнения цикъл. Той включва изучаване на текущото състояние на проблема, определяне на специфични потребности и изисквания, начална оценка на алтернативни решения. Този етап завършва с изготвяне на задание за проектиране, което съдържа уточнените изисквания на потребителя към системата.

Системното проектиране завършва със спецификация за реализация. Тази спецификация съдържа общо и детайлно описание на компютърното решение, избрано при анализа. Формулираните спецификации се дават на програмистите за реализация.

Системната реализация включва създаването на работеща програмна система. Програми се пишат, тестуват и настройват. Бъдещите потребители се обучават за работа с новата програмна система.

Ситемната поддръжка включва експлоатация и съпровождане след внедряване на системата. Възможно е чрез обратна връзка с програмистите да се репрограмират отделни модули за подобряване характеристиките на системата. Допуска се този етап да завърши с формулиране на изисквания за разработка на нова система, с което цикълът анализ, проектиране, реализация и поддръжка се затваря.

Друга трактовка за процеса на проектиране, разработка и реализация на един програмен продукт се дава в [48], където жизненият цикъл на програмните системи се формулира като последователност от следните фази:

1. Формулиране на изисквания (Requirements phase) - 3%;

2. Описание на проблема (Specification phase) - 4%;

3. Планиране (Planning phase) - 2%;

4. Проектиране (Design phase)-6%;

5. Реализация (Implementation phase) - 12%;

6. Интегриране (Integration phase) - 6%;

7. Поддържане/съпровождане (Maintenance phase) - 67%.

Относителният дял на всяка една фаза е представен в проценти.

Цел на фаза формулиране на изисквания е в диалог (интервю) с потребителя да се определят възможно най-точно неговите потребности - от каква програмна система се нуждае той. Фазата се нарича още системен анализ.

Целта на фаза описание на проблема е след определяне на потребностите да се създаде документ, който описва проблема и точно отразява какво следва да прави програмният продукт.

Целта на фаза планиране е оценка на необходимото време и ресурси за изпълнение на проекта - каква продължителност, колко проектанти и програмисти, какви разходи ще изисква проектът.

Целта на фаза проектиране е следваща детайлизация. Ако във фаза описание се казва *какво*

трябва да прави продуктът, сега при проектирането се създава документ, който предписва
как
продуктът ще постигне целта си. Продуктът се разбива на съставни компоненти, всеки от които се проектира.

Целта на фаза реализация е програмиране на продукта. Изборът на инструментален програмен език се фиксира в документа за описание на проблема. Всеки модул се кодира и тестува.

Целта на фаза интегриране е свързване на отделните модули в завършен продукт. Тази фаза завършва с успешно провеждане на тест за приемане, с който клиентът потвърждава, че продуктът отговаря на изискванията в документа за описание.

Целта на фаза поддържане/съпровождане е отразяване на промени в продукта, след като е предаден на потребителя. Промените могат да се предизвикват от необходимостта да се отстранят възникнали грешки (bugs, corrective maintenance) или да се добавят нови функционални възможности към досегашните, с които разполага продуктът (enhancement maintenance).

От практиката са известни следните модели на изпълнение на жизнения цикъл на един програмен продукт с неговите седем фази [48]:

1. Създай и фиксирай (Build and Fix model). Продуктът се създава и предава на клиента без да се провеждат фази описание на проблема, планиране и проектиране. Клиентът посочва какви евентуални промени да се направят. След извършване на промените продуктът се предоставя на клиента. Този процес продължава, докато изискванията на клиента се удовлетворят. Моделът е приложим за малки по обем проекти като студентски задачи и учебни демопримери.

2. Водопад (Waterfall model). Този модел се нарича още лавинообразен или каскаден [50]. Последователно се изпълняват фазите от жизнения цикъл. Резултатът от всяка фаза се подлага на проверка за валидност (verify and test) и само след положителен резултат от проверката се пристъпва напред към следваща фаза. Особено внимание се

отделя на диалога (интервюто) с клиента за получаване на необходимата информация за определяне на изискванията и формулиране на документа за описание на проблема. Предимството на модела е наличието на документация, съпътстваща всяка фаза. Недостатъкът на модела е, че по време на отделните фази клиентът не разполага с работеща макар и неокончателна версия на продукта, а разполага само с документация, от която слабо се информира за потенциалните възможности на разработвания продукт. Така клиентът не може да получи реална визуална представа за продукта преди предоставянето му в крайната фаза на жизнения цикъл.

3. Бърз прототип (Rapid Prototyping model). При този модел първата фаза за определяне на изискванията освен с данни за съставяне на документ за описание на проблема, завършва и с предварителна груба и примитивна реализация на продукта, наречена бърз прототип (rapid prototype). Клиентът получава възможност за визулана представа относно бъдещия продукт и реална оценка на функционалните му възможности. Така той преценява дали се удовлетворяват или не съответните изисквания. Клиентът експериментира с прототипа и прави забележки, които се отразяват непосредствено в прототипа. Този процес трае, докато клиентът бъде удовлетворен. Едва сега се създава описанието на проблема, който става спецификация за реализация на продукта.

4. Спирала (Spiral model). Основно внимание се отделя за оценка на рисковете, свързани с реализацията на продукта и извършване на действия, които разрешават проблемите и водят до минимизиране на рисковете. Възможни са различни рискове. Например опасението, че крайният продукт няма да удовлетвори изискванията на клиента. Удачно решение за разрешаване на този риск е да се създаде и предостави своевремнно на клиента работещ прототип на бъдещия продукт. Друг възможен риск се крие в опасението, че разходите за реализация на продукта няма да се възстановят.Удачно решение за разрешаване на този риск е да се провежда преди началото на всяка нова фаза прецизен икономически анализ и по-нататъшната реализация на продукта да продължава само след положителна икономическа прогноза. Изобщо анализ на възможните рискове се провежда преди всяка фаза от реализацията на проекта. Две условия обуславят прилагане на спирален модел съгласно [49]. Наличие на голям по обем проект, както и изискването клиент и разработчик на проекта да са от една организация.

Три категории специалисти са ангажирани при създаването на една програмна система. Те могат да бъдат групирани така:

1. Възложител, краен потребител (client, end user);

2. Проектант (systems analyst, systems designer, systems consultant, management consultant, operations analyst, information analyst, data analyst, business analyst, solution designer);

1. Програмист (developer, applications programmer, systems programmer). Примерно разпределение на участниците в жизнения цикъл на програмната система по етапи *с*
показано в табл. 4.1.

Програмистите и проектантите са специалисти с различна квалификация.

Какви са изискванията към програмиста? Той трябва да владее програмен език или програмни езици, конкретна операционна система и набор ситемни помощни програми.

Каква квалификация е нужна на проектаната? Наред с уменията на квалифициран програмист той трябва да бъде с много богата професионална култура. Необходими са познания по принципи и системи за работа с бази данни, комуникации на данни, микрокомпютри, софтуер за компютри -програмни езици и операционни системи, машинна графика, сигурност на данните, изкуствен интелект, експертни системи и др. Проектантът трябва да бъде комуникативен. Много важно е да бъдат разбрани и правилно формулирани изискванията на потребителя, а това зависи от умението на потребителя да общува.

В [17,48] са описани различни схеми за обединяване на специалистите, работещи върху реализацията на програмен проект в колективи:

1. Бригада на водещия програмист (chief programmer team). Това е екип с главен програмист и редови програмисти.

2. Екип с двама ръководители. Той включва главен програмист (team leader) и управител (team manager);

3. Екип с йерархична подчиненост на три нива. Тези нива включват ръководител на проект (project leader), един или няколко ръководители на екипи (team leader) и редови програмисти (programmer).

4.3. ЕТАП ПРОЕКТИРАНЕ - МЕТОДИ И ТЕХНИКИ

Три етапа предшестват програмирането в жизнения цикъл на програмните системи. Тук се включват дефиниране на проблем за решаване, изготвяне на задание за проект и проектиране. Тези три етапа приключват със завършен проект за програмиране. Ще представим основните процеси и дейности, които се изпълняват при проектирането на голяма програмна система [17].

1. Изисквания. Формулирането на изискванията описва очакванията на крайния потребител от готовия програмен продукт.

2. Цели. Целите определят какви задачи за решаване трябва да се поставят, за да се удовлетворят изискванията на потребителя от т. I.

3. Предварителен външен проект. Формата на взаимодействие (диалог) с потребителя се изгражда в най-общ вид без подробности.

4. Детайлен външен проект. Уточняват се всички подробности на взаимодействието с потребителя.

5. Архитектура (фасада) на програмната система. Фиксира се множеството програми, изграждащи програмната система.

6. Всички структури от данни за работа на програмната система се проектират и описват.

7. Структура на отделна програма. Проектира се структурата на всяка една програма от общото множество по т. 5. Специално внимание се отделя на дейността за избор на алгоритъм.

8. Логика на модули. Проектира се логиката на отделните модули, ' изграждащи отделна програма по т. 7.

Изброената поредица от действия е идейна и не следва да се третира като задължителен канон. Възможно е някое действие да липсва: например ако програмната система не работи с външни данни или структурата й е от единствен програмен модул. Обратно, възможно е определено действие да се натовари с повече от описаните дейности: например изборът на инструментален програмен език е дейност, която може и следва да се включи на различни места в горната схема - т. З, 7 или 8.

Изборът на най-добър алгоритъм е важен фактор за получаване на ефективна и коректна програмна система. Известно е, че за решаване на определен проблем като правило са възможни множество алгоритмични решения. Недобра практика е при проектирането да се избере първият дошъл на ум алгоритъм, който може би не винаги е най-доброто възможно решение. Поуката и за проектанти, и за програмисти е, че един час повече, използван при проектиране за избор на удачен алгоритъм, може да спести много часове за програмиране, тестуване и настройка. Естествено е да се прави анализ на временната и пространствената сложност на възможните алтернативи при избора на алгоритъм.

Недостатъците, които се проявяват на етап изготвяне на задание, и необходимостта от систематичност при провеждането на етап проектиране са наложили развитието на различни методи и техники за описание на заданието и формулиране на резултатите от процеса на проектиране:

Словесен подход. Средството за проектиране се нарича обикновен естествен език. Касае се за прилагане на ограничен, структуриран естествен език, наричан още псевдоезик, който се комбинира с традиционни синтактични изразни средства на програмните езици от високо ниво като оператори за цикъл и разклонение. Така се получава хибрид, който след това се уточнява и прецизира във формата на конкретна програма. Приложение на този подход може да се види и в книгата на Kernighan/Ritchie „The C Programming Language" [37]. Проектирането и реализацията на програма, филтрираща най-дългия текстов ред от входния поток, въведен от клавиатурата, се описва в т. 4.1. Разновидност на словесния метод с ориентация към обектно ориентираното проектиране и програмиране е дадена от R. Abbott в статията „Program Design by Informal English Descriptions" (Comm of the ACM, vol 26, no 11). Предлага се описанието на проблема за решаване да се направи в изречение на естествен език. След това съществителните имена се маркират като кандидати за означаване на обекти и/или шаблони на класове, а глаголите се маркират като кандидати, с които се именуват методите на избраните класове. Например от изречението „Стекътсъхранява и извежда данни." не е трудно да се състави описание на клас стек с два метода за съхранение на данни в стек и извеждане на данни от стек.

```
//

class Stack

{

void push(int ..)  // метод за съхранение (запис) на данни в стек

{ - }
 int pop()                          // метод за извеждане (четене) на данни от стек

{ ...  }

};
```

//

Схеми на потока данни (data flow diagrams DFD). За разлика от процедурните блоксхеми, схемите на потока данни не показват как се предава управлението при изпълнение на една програма. Те не задават проверки на условия и циклични обработки. Тези схеми показват само потока на данните (data flow), които се обработват - откъде идват, къде се съхраняват, къде се извеждат резултати, кои са процесите, обработващи данните. Работи се с физически схеми на потока данни (physical DFD), които се отнасят до физическата обработка на данните и логически схеми на потока данни (logical DFD), които третират логическата обработка на данните.

Йерархични диаграми. Съставя се йерархична структурна и функционална диаграма (visual table of contents) за проектираната програмна система. Значими компоненти (функции) от системата се маркират. За всяка от означените функции се съставя диаграма на обработките по класическия модел на изчислителен процес (Input, Process, Output). По тази причина диаграмите са известни като IPO-схеми. С тях се описват функционалните изисквания на потребителя и те му дават нагледна представа за бъдещата програмна система без да съдържат описания на конкретни структури от данни или първични текстове на конкретен програмен език.

Таблици на решение (Decision Tables). Те са средство за описание на даден проблем, което позволява систематична формулировка на всички възможни варианти на входни условия във формата на правила. За всяко от записаните правила се задава адекватна потребителска реакция, наречена действие. Таблиците на решение са средство за общуване между проектант, краен потребител и програмист. Съставят се от проектанта. Те са информативни за потребителя и служат като отправна точка за програмиста. Подробно таблици на решение са описани в [28].

Блоксхеми (flowcharts). Ще поясним блоксхемите като метод с приложение в проектирането. Трябва да се знае, че те са средство, приложимо както в проектирането (процесът завършва *с* продукт блоксхема), така и в програмирането (процесът започва с продукт готова блоксхема). За разлика от схемите на потока данни, блоксхемите показват потока на управлението. Те описват какви действия се извършват в зависимост от едни или други условия при изпълнението на една програма с конкретен набор входни данни. В този смисъл като друго означение на блоксхема може да се ползва и терминът control flow diagram (CFD).

От гледна точка на практиката всяка блоксхема се състои от следните конструктивни елементи:

1. Блокдействие с един вход и един изход;

2. Блокразклонение с един вход и два или повече изхода;

3. Елемент начало с един изход;

4. Елемент край с един вход;

5. Елемент сливане с два или повече входа и един изход.

От гледна точка на теорията блоксхемата е абстрактна структура -насочен (ориентиран) ацикличен граф, който указва реда на изпълнение на операторите в една програма.

Всеки оператор от програмата се представя като възел в графа. Всяко възможно предаване на управлението се представя като дъга (линия, свързваща два възела) в графа. Ако един възел има една входяща и една изходяща дъга, той описва оператор за действие от програмата и се нарича *функционален възел.* Ако един възел има една входяща и две или повече изходящи дъги, той описва управляващ оператор от програмата и се нарича *пред икатен възел.*
Третият възможен тип възли има две или повече входящи дъги и една изходяща дъга и се нарича
възел за сливане.
Възлите за сливане не влияят по никакъв начин на обработваните данни.
Управляващата структура на всяка една блоксхема се конституира чрез формиране на комбинации от функционални възли, предикатни възли и възли за сливане.

Изпълнение на една блоксхема се нарича последователността от действията, написани във функционалните блокове, през които минава пътят (управлението). Дървото на изпълненията представя множеството от възможните последователности от действия през всички възможни клонове на блоксхемата [15].

CRC-карти. Това е техника с приложение в обектно ориентираното проектиране. Съкращението означава Class (Клас), Responsibilities (Отговорности, Действия), Collaborators (Сътрудници). За всеки клас от проблемната област се строи структура по подобие на фиг. 4.1.

Отляво в колона се изброяват отговорностите на класа. Това са действията (един или няколко метода), които проектираният клас ще изпълнява. Отдясно срещу всяка отговорност се изброяват класове (обекти), които ще се активират при изпълнение на съответното действие. Това е начин за изясняване на връзките между класовете в общия модел. Ниска степен на обвързване (low coupling) между класовете е характерна за добрите обектно ориентирани модели [53]. Обратно, признак за некачествено създаден обектно ориентиран модел в съответната предметна област е случаят на CRC-карта с отговорност, на която съответстват много сътрудници.

Изпълнението на етапите проектиране, а в последствие и програмиране се реализира чрез спазването на определена техника (методика). Възможни са следните подходи:

Тотално проектиране/програмиране. Тук няма никакава специална идея или замисъл. Целият проект се замисля като един модул, който се свежда до работеща програма. Намира ограничено приложение за малки по обем учебни и/или демонстрационни програми.

Модулно проектиране/програмиране. Проектите/програмите се разделят на логически части и последователно се проектират/програмират. Предимството се състои в разделянето на голям проект или програма на по-малки логически обособени части, решаването на всяка от които е по-лека задача от решаването на проекта като цяло. Типични изисквания, които се поставят при обособяване на модулите, имат за цел да осигурят:

1. Независимост. Всеки модул е самостоятелна единица и не зависи от останалите. Може да се замени с друг без това да влияе на останалите.

2. Приемлив размер. В [14] се цитират данни от компютърни компании и софтуерни фирми, които налагат различни практически нормативи. Модулът е част от програма, който се пише, тестува и настройва за 1 месец. Модулът съдържа не повече от 100 оператора на програмен език от високо ниво.

Възходящо (bottom-up) проектиране/програмиране. Отделните модули на дадена програмна система се разработват (проектират/програмират) отначало отделно и след това се обединяват в единно цяло. Основният недостатък на този подход се крие във факта, че грешки в проектирането често се откриват едва при настройката (обединяването) на модулите в едно цяло, след като много модули са написани и се окажат безполезни.

Низходящо (top-down) проектиране/програмиране Основната идея е в противовес на възходящото проектиране. Управляващите програми се съставят отначало (проектират, програмират, тестуват и настройват), а функционалните изпълнителни модули се добавят в процеса на разработката на цялата система. Обичайно низходящото проектиране започва с логическо проектиране. Така се обезпечава съгласуваност на работата между отделните програмни модули.

При възходящото проектиране програмните модули не се проверяват като съставна част на цялата система до края на разработаката им.

При низходящото проектиране програмните модули се проверяват веднага за съвместимост към системата. Ако те не са готови, тяхното място се заема от празни програмни единици (stubs).

Низходящата методика допуска ранното откриване и отстраняване на допуснати грешки при проектирането/програмирането. Това става в периода, когато програмите са все още при разработчика. Обратно, възходящата методика е предпоставка за откриването

на грешките на по-късен етап при комплексната настройка на програмната система. Това е недостатък, защото разработчикът на конкретните модули е приключил работа и е възможно да работи по друг проект.

Описаните по-горе методи и техники за проектиране (словесен, схеми на потока данни, йерархични диаграми, таблици на решение, блоксхеми), както и методиките за тяхното провеждане (тотално, модулно, възходящо, низходящо проектиране/програмиране), произтичат от принципите на структурния анализ и проектиране на системи (SSAD - Structured Systems Analysis and Design) и са свързани със структурното програмиране. Еволюцията на структурното програмиране доведе до развитие на обектно ориентираното програмиране, а това наложи и нови тенденции в анализа и проектирането на програмни системи. Водещият принцип от структурния анализ и синтез на системи *мислене чрез функции* (Thinking in Functions) остава класически, но се счита остарял и неактуален. Той се измества от наложилия се съвременен *притцип мислене чрез обекти*
(Thinking in Objects), който е валиден за обектно ориентирания анализ и синтез на системи. Развиха се множество методи за обектно ориентиран анализ, синтез, проектиране и програмиране на програмни системи. Най-известните от тях според [53] са:

1. Метод за обектно ориентирано проектиране OOD (Object Oriented Design) с автор Гр. Буч (Grady Booch);

2. Метод за обектно моделиране OMT (Object Modeling Technique) с автор Дж. Ръмбау (James Rumbaugh);

3. Метод за обектно ориентирано софтуерно инженерство OOSE (Object Oriented Software Engineering) с автор А. Якобсон (Ivar Jacobson).

Тримата цитирани автори работят за фирмата Rational Software (http:// www.rational.com) от 1994-95 г. Там те обединяват усилия за разработка на унифициран метод за обектно ориентирано проектиране (Unified Method). В резултат се ражда езикът за описание и моделиране на обекно ориентирани системи, известен с абревиатурата UML (Unified Modeling Language). Най-характерното за този език е, че на проектанта се предоставя възможност в унифициран диаграмен вид (фиг. 4.2.) да изобрази класовете/обектите, с които се описва задачата за решаване.

Диаграмата за всеки клас/обект съдържа раздел наименование на класа, раздел елементи данни и раздел методи на класа. Всеки елемент или метод може да бъде предшестван от представка +, - или #. Така се описват квалификаторите за видимост public, private или protected. Атрибут за статично (static) обявяване на компонент се задава допълнително с $. Ако практическият проблем е достатъчно елементарен, той се описва само с диаграмата на един обект. Реалните проблеми се описват с множество взаимосвързани класове/ обекти. Проектирането продължава с изграждане на връзки между диаграмите на класовете/обектите. Обобщено връзките между класовете се наричат асоциация (association) и се означават с линия, която свързва диаграмите на класовете. Връзка от тип асоциация между клас А и клас Б е показана на фиг. 4.3.

Наследяването между базов и породен клас е връзка, която се нарича още връзка тип „е" („is a"). Като пример за наследяване са показани връзките между класовете Мраз, Хладилник и Домакински уред. Те са представени на фиг. 4.4. и се изговарят по следния начин. Мраз е марка (вид) хладилник. Хладилник е вид домакински уред.

Мраз е клас, който има свои характеристики, но наследява и характеристиките на класа Хладилник.

Хладилник е клас, който има свои характеристики, но наследява и характеристиките на класа Домакински уред.

Агрегацията е връзка от тип част - цяло, която се нарича още връзка тип „има" („has") или връзка тип „е част от" („is a part of). Този тип асоциация е илюстриран на фиг. 4.5 с пример за връзка между класовете Ястие и Хранителен продукт. Дадено ястие има (съдържа) хранителни продукти. Даден хранителен продукт е част от едно или повече ястия.

Този тип връзки имат и характеристика кардиналност (cardinality). Връзката се надписва в двата края с числа (стойности или диапазон от стойности), които показват броя обекти от съответния клас, които участват в асоциацията. Възможни стойности за означаване са:

1 - един обект;

* - неопределен брой обекти;

0..1 - нула или един обект;

0..* - нула или повече обекти;

1 ..* - един или повече обекти.

Обогатената връзка от тип агрегация с въведена кардиналност от двете страни е показана на фиг. 4.6. Тя съответства на следната схема. Едно или повече ястия съдържат три или повече хранителни продукта. Три или повече хранителни продукта са част от едно или повече ястия.

Следващата връзка се нарича композиция и е разновидност на агрегацията. При композицията цялото напълно притежава (strongly owns) своите компоненти [53]. Всички действия с обектите от целия клас (копиране, преместване, разрушаване) влекат същите действия, които се извършват над обектите, които го съставят. Обектите компоненти са притежание на точно един определен обект от класа цяло. Затова стойността, с която се означава връзката от страната на цялото, е 1. Примерът, който е показан на фиг. 4.7, описва връзка композиция между клас Жилищен блок и клас Апартамент.

Жилищен блок има (съдържа) апартаменти. Апартаментът е част от жилищен блок.

Не е възможно един конкретен апартамент да е част от няколко жилищни блока. Той принадлежи на точно един жилищен блок. По тази причина с отчитане на кардиналността описанта връзка композиция означава: Един жилищен блок (конкретен обект) има 17 апартамента.

4.4. ЕТАП ПРОГРАМИРАНЕ

Идеята на възложителя, трансформирана в проект, намира своята практическа реализация във вид на програмна система на етап програмиране. Погрешна е представата сред част от професионалната колегия, че програмирането е най-значимият етап в жизнения цикъл на програмния продукт. Напротив, статистиката [17] сочи следното примерно разпределение на разходите за отделните етапи: проектиране - 20%, програмиране - 5%, тестуване и настройка - 25%, съпровождане - 50%. Усилията на програмистите не бива да се омаловажават, но данните показват реалния дял на етапа програмиране. Ето защо не е правилно да се обръща внимание единствено на програмистката квалификация за сметка на другите етапи в контекста на цялостния жизнен цикъл на програмките системи. Независимо от това, свидетели сме на еволюция и усъвършенстване на стиловете, методите, техниките, концепциите в програмирането. Всички те имат за цел повишаване ефективността на този труд. Тези въпроси са разгледани подробно в *гл. 5. Стилове в програмирането.* Друга насока за повишаване ефективността на програмисткия труд е автоматизираното генериране на програмни текстове, сведения за което се дават в
т. 1.2. Автоматизирано генериране на Windows -приложения
и
т. 4.6. Средства за автоматизирана разработка на програмни системи.

4.5. ЕТАП ТЕСТУВАНЕ И НАСТРОЙКА

Дейностите тестуване и настройка са свързани с откриването, локализирането и отстраняването на грешки при реализацията и изпълнението на програмните системи. В началото на тази глава бяха дадени определния за грешка в програмното осигуряване и надеждност на програмното осигуряване.

Когато се говори за грешки, надеждност, тестуване и настройка, тематиката не се възприема еднозначно. Програмните системи силно се различават от гледна точка на сложността - обем първични оператори, модулна структура, период за разработка, брой програмисти, участващи в проекта. Естествено е при наличие на голямо разнообразие от програмни системи, представите за надеждност, отсъствие или наличие на грешки да се различават. Тук ще изложим общи принципи, валидни изобщо и независимо от

разнородността на програмните системи, които са обект на разглеждане. На първо място какво означава тестуване?

Ще приведем и анализираме две определения за тестуване:

Определение 1: Тестването е процес на изпълнение на програма, потвърждаващ правилността на изпълнение на програмата и потвърждаващ, че в програмата няма грешки.

Определение 2: Тестването е процес на изпълнение на програма, с който се цели откриване на грешки в програмата.

Първото цитирано определение се оценява като напълно неправилно, тъй като то по същество е антоним на думата тестуване. Всеки програмист с опит знае, че не е възможно да се демонстрира отсъствие на грешки при изпълнение на програма. Коректно следва да се счита второто определение.

В потвърждение ще приведем и известната мисъл на Дийкстра: *Тестването на програми може да служи като доказателство за наличие на грешки в програмите, но никога не може да докаже тяхното отсъствие.*

Следва класификация на видовете тестуване в зависимост от средата, в която се провежда Тестването на програмната система [17]:

Контролно тестуване (verification testing). Опит да се открие грешка или грешки, като програмата се изпълнява в симулирана нереална среда в лабораторни условия с фиктивни недействителни данни. Този вид тестуване е известен и като алфа (a - alpha) тестуване.

Изпитателно тестуване (validation testing). Опит да се открие грешка или грешки, като

програмата се изпълнява в реална среда с реални данни.Този вид тестуване е известен и като бета (6 - beta) тестуване. Навярно читателят се е сблъсквал с бета версии на операционната система MS-DOS, например версия 6.22 или у читателя са попадали компактдискове с бетаверсии на операционната среда Windows, които фирма Microsoft официално разпространява през три месеца преди официалното представяне на своите версии като Windows 95 или Windows 98. По този начин фирмата цели обратна връзка от потребителите за поведението на програмния продукт, който е в етап на тестуване и настройка.

Атестационно тестуване (certification testing). Това е авторитетна проверка за правилността на работа на една програмна система, като резултатите от изпълнението се сравняват с предварително известно множество от очаквани стойности.

Ще приведем и класификация на видовете тестове в зависимост от структурните елементи на програмната система, които се подлагат на тестуване:

Автономно тестуване (module testing, unit testing). Провежда се тестуване на отделен програмен модул в изолирана среда и независимо от всички останали модули. Автономните тестове имат отношение към вътрешната логика на модула и външната му спецификация за връзка с други модули.

Интегрално тестуване (integration testing). Провежда се с цел контролиране на връзките между компонентите на програмната система. Този вид тестуване има отношение към структурата на отделната програма и архитектурата на цялата програмна система.

Комплексно тестуване (system testing). Провежда се с цел контрол на програмната система по отношение на окончателните резултати, които се получават при изпълнение на програмната система.

По време на етапа тестуване се провежда още една дейност - *настройка (debugging)* на програмните продукти. Погрешна е представата, че настройката е разновидност на тестуването. Двете дейности не бива да се смесват. Между тях има връзка и разлики, които ще поясним съгласно [25].

Ако една програма очевидно не работи правилно, тя се настройва.

Ако една програма видимо работи правилно, тя се тестува.

Тестуването е дейност, с която се открива наличие на грешки.

Настройката е дейност, с която се установява точната природа на грешката. Следва локализиране на грешката и нейното отстраняване.

Казаното по-горе потвърждава, че става дума за два взаимосвързани и припокриващи се процеса. Обикновено резултатите от тестуването служат като начални данни, с които се старира настройката.

В практиката са се наложили различни подходи, методи и средства за настройка (локализация на грешки) при изпълнение на една програмна система. При [32] се прави следната класификация:

1. Метод на грубата сила;

2. Интелигентни методи.

Ще изброим някои типични дейности в областта на грубата сила: а/ Добавяне (вмъкване) в програмата на контролен печат. Тази идея за вмъкване на оператори за печат на междинни резултати не е лишена от смисъл, но е пример за безсистемен подход на програмиста, който просто е изчерпал всички други възможности. Ако този принцип се възприеме, ето някои препоръки: 1. Извеждане на ехопечат за всички входни данни; 2. Извеждане на информация за развитието на числовите пресмятания; 3. Извеждане на информация за логическата последователност при изпълнението на програмата; 4.

Установяване на програмни флагове за включване на контролния печат само в случай, че се работи с тестовите версии на програмните системи.

б/ Извеждане съдържанието на определени области от паметта.

в/ Стъпково проследяване изпълнението оператор по оператор.

г/ Установяване на условни и/или безусловни точки на прекъсване (break points) и автоматично изпълнение с прекъсване при достигане на съответна точка на прекъсване.

Изброените дейности могат да се извършват ръчно и автоматизирано. В първия случай се изискват умствени усилия от програмиста или тестуващия инженер. Във втория случай се налага познаване на средствата за настройка, които предлагат програмните среди и продукти. Известни са програмите Debug.exe (DEBUGger), Symdeb.exe (SYMbolic DEBugger), Afd.exe (Advanced Full screen Debugger). Те се използват за тестуване и настройка на асемблерни първични текстове в среда MS-DOS. На разположение са средства за тестуване и настройка на C/C++ първични текстове - вграден дебъгер в интегрираните програмни среди и автономна програма Td.exe (Turbo Debugger) и др.

Интелигентните методи се основават на систематично формулиране на ситуациите с изграждане на множество възможни хипотези и тяхната последваща проверка.

Практическата работа по тестуване и настройка на една програмна система най-често включва елементи и подходи както от грубата сила, така и от интелигентните методи.

Подобно на работата при етапи проектиране/програмиране, и тук при тестуването и настройката са възможни различни техники (стратегии) [17]:

Тотално тестуване. Програмната система се компилира и тестува, като се проверява

изпълнението поведнъж за всяка от възможните комбинации входни данни. Този подход е възможен практически само в случаи, когато комбинациите входни данни са ограничен краен брой и времето за изпълнение е приемливо кратко.

Възходящо (bottom-up) тестуване. Програмната система се компилира и тестува отдолу нагоре. Единствено модулите на най-ниско ниво се тестуват автономно и изолирано. След като тяхното тестуване завърши, извикването им трябва да е толкова надеждно, колкото е извикването на функциите от системните библиотеки или методите от библиотеките с класове на комерсиалните програмни продукти. Следва тестуване на модули, които непосредствено викат вече проверените модули. Този път тестването не е изолирано, а съвместно с модулите от по-ниските нива. Процесът продължава, докато се достигне до върха в йерархията на програмната система.

Низходящо (top-down) тестуване. Програмната система се компилира и тестува отгоре надолу. Изолирано и автономно се тестува главният модул. След това към него се съединяват един след друг съставните модули, които главният непосредствено извиква. Получената комбинация се тестува. И тук процесът се повтаря, докато се комплектоват и проверят всички съставни модули. При низходящото тестуване е важно да се изяснят два въпроса:

а/ Как се процедира в случай, че тестуваният в момента модул извиква модул, който още не е тестуван? Липсващият модул се замества с фиктивен, който има само входна точка, евентуално контролен печат за индикация и изход в извикващата среда. Това са празни програмни единици (stubs), които в най-простия случай имат следната структура:

заглавие - входна точка

return - логически край

end - физически край

б/ В каква форма се подготвят тестовите данни и как се подават на програмата? Тестовите данни се подготвят така, както се готвят входни данни за всяко друго изпълнение. При добре проектираните програми входните операции са извън главната програма и са концентрирани в самостоятелни модули. Това води донякъде до отклонение от строгия низходящ принцип.

Тестването и настройката на програмните системи са свързани с подбирането на подходящи комбинации входни данни, с които да се тестува работоспособността на разработвания програмен продукт. Тази дейност е известна като *проектиране на тестове* и не е редно да се извършва от програмиста на програмата. В софтуерните фирми се формират специални групи за тестуване. В техния състав влизат високо квалифицирани специалисти (quality assurance engineers). В предговора към книгата на Стив Магуайър [16] „Как да пишем надеждни програми" Дейвид Мур дава данни за Microsoft, от които става ясно значението, което тази фирма отдава на дейността по тестуване на програмни продукти. Тестващата група през 1984 е наброявала 5 човека, а през 1993 е нараствала на повече от 500. Изобщо висококачественото проектиране на тестове е сложен и отговорен процес. При него се изисква както творчество, така и известна доза разрушителна сила на духа [17].

Основен принцип при проектирането на тестове е да се подготвят тестови данни, с които се проверява всеки клон на алгоритъма, всяка ситуация, всяка възможност, всяка комбинация допустими стойности от входни данни. Тестовете следва да обхващат всички възможности при изпълнение на програмата. Например при участък с разклонен алгоритъм е необходим тест, който ще гарантира изпълнение на всички условни преходи във всички разклонения. При участък с цикъл е необходим тест, който ще изпълни цикъла О пъти, тест, който ще изпълни цикъла 1 път, тест, който ще изпълни цикъла максимален брой пъти. Въвежда се правилото на *минимален критерий* - най-мал ко по едно изпълнение за всички клонове на алгоритъма.

Подборът на данните за тестовите поредици трябва да обхваща следните различни случаи:

1. Работа с тестови данни - нормални случаи;

2. Работа с тестови данни - гранични случаи;

3. Работа с тестови данни - изключения;

4. Работа с тестови данни, известни като *нулеви случаи.* За аритметичните данни това е стойност нула. За символните данни това е низ от празни позиции - интервали, или низ с нулева дължина. За указателите това е специалната стойност nil в езика Pascal и NULL в езиците C/C++. Набор от данни, при които програмата не изпълнява никакви действия, се наречени *нулев вариант.*

Тестването и настройката имат своя алтернатива и тя се нарича *анализ и верификация на програмни продукти.* Невъзможността да се провежда тотално тестуване на реални програмни системи и трудностите с подбирането на подходящи тестови поредици за всички възможни случаи и режими на изпълнение са довели изследователите до следната алтернативна идея. Вместо тестуване на програмните системи провежда се *доказателство, че програмите правилно изпълняват своите функции и своето действие.* Процесът на доказателство на правилността на програмното изпълнение се нарича още аналитична верификация. Основната идея на този подход е следната. На всеки оператор от програмата се съпоставят два булеви израза - предикати. Единият изразява знанието за състоянието на програмата преди изпълнението на оператора. Другият изразява знанието за състоянието на програмата след изпълнението на оператора. Формално тези предикати се записват като коментари в програмните текстове:

Pascal	C/C++
{ предусловие }	/* предусловие */
' { постусловие }	/* постусловие */

Поставя се задачата да се докаже истинността на постусловието при положение, че предусловието е вярно за всеки отделен оператор. Този процес се развива върху всички оператори на цялата програма. Разглеждат се оператор по оператор. Започва се от първия и се търси доказателство, че от предусловието на първия оператор следва постусловието на последния оператор на програмата Отношението между първо предусловие и последно постусловие характеризира спецификацията на програмата (какво тя прави) Тогава доказателството, че постусловието следва от предусловието, е равностойно на твърдението, че програмата ще се изпълнява в съответствие със своята спецификация. Изложената идея за аналитична верификация на програмните

системи през годините не доби популярност и следва да се отнесе към теоретичните възможности и области на изследователски интерес. Подробности за този подход могат да се прочетат в [10].

4.6. СРЕДСТВА ЗА АВТОМАТИЗИРАНА РАЗРАБОТКА НА ПРОГРАМНИ СИСТЕМИ

Класическият път за реализация на една програмна система изисква след проектиране да се премине към етап програмиране. В миналото процесът програмиране се извършваше изцяло ръчно. Това е бавна, трудоемка и отнемаща много време дейност. За улеснение на проектантите и програмистите в практиката съществуват специализирани програмни средства (CASE tools - computer aided software engineering tools), с които процесът на генериране на първичния текст, представящ скелета на една програмна система, се провежда автоматизирано по данни, които са резултат от етапа проектиране. В миналото това бяха програмни средства за структурен анализ, проектиране и реализация на програмни системи (structured analysis and design methods). Съвременните CASE tools са ориентирани към обектен анализ, проектиране и реализация на програмни системи (object oriented analysis and design methods). Основни изисквания към съвремнните средства за автоматизирана разработка на програмни системи са:

а/ Наличие на графичен редактор (diagram editor) на диаграми, с който на етап проектиране се задават класовете/обектите и връзките между тях, описващи конкретния проблем, например с диаграми на езика UML;

б/ Средства за проверка на синтактична и ограничена семантична коректност на зададените диаграми;

в/ Средства за генериране на първичен текст (code generator) от описаните диаграми с възможност за избор на инструментален програмен език, например C/C++, Java, Visual Basic;

г/ Средства за генериране на съпътстваща документация (document generator), например файл с разширение .rtf;

д/ Средства за обратно преобразуване (reverse engineering facilities), които позволяват промени, направени ръчно в автоматично генерирания първичен текст, да бъдат отразени в графичните диаграми, с които започва генерацията на първичнен текст;

е/ Запомняща среда (repository), в която диаграмите и техните атрибути/ свойства се съхраняват за по-нататъшна употреба.

Популярни софтуерни продукти (CASE tools), които служат за автоматизарана разработка на програмни системи, са:

Rational Rose (http://www.rational.com);

Visual Case (http://www.stingray.com);

Class Designer (http://www.cayennesoft.com)

Pragmatica (http://www.pragsoft.com).