Writing a Specification

Whether the software you're about to write replaces paperwork with software, or replaces old software with new, or solves a new problem, there is one way to begin: analyse the problem in detail.

A convention in North America is to call this first design stage *defining the business problem*. That usage is too narrow; the problem may be technologic, scientific, personal, educational, commercial, govermental, or something else. What application software problems have in common is not business problems but *requirements*. We model them by defining exactly what tasks are to be done, who is to do them, how they are structured, how their data is to be stored, and how the data maintenace software is to be structured.

Requirements modelling

In the early days of databases, and still in the early days of microcomputers, database developers were often solving smallish problems one at a time, often with databases that were simple in structure and limited in capability. Design often involved not much more than sketching a flow diagram, inferring the database structure from the diagram nodes, then writing the pseudocode. Development was *code-centric*. The resulting software was 2-tier, and the two tiers were inextricably linked.

Forty-five years ago, a middle-of-the-road computer had 64K of memory and filled a room. The internet was just an idea. 25 years ago, a reasonable microcomputer had four times that much memory and sat on a desk. The internet was still just an idea. Now a middle-of-the-road laptop has 32,000 times as much memory as that long-ago mainframe, is thousands of times faster, and with one click can connect itself to small or huge databases anywhere on the planet, or to any of a billion internet users.

No more. The problems you are paid to solve now are complex and multi-layered: the application will be event-driven, it may interact with other software running anywhere on the planet, it may be distributed across multiple computers in one building, or on several continents; it may access multiple databases in one building or on several continents, and hundreds or thousands or millions of users may be banging on it at one time. Almost certainly it will be too complicated for a single flow diagram.

Sure, you can still work code-centrically. Depending on your pain threshold, you might continue that way for a month or a year. But eventually the ghastly truth settles upon you: the impulse to code has been your worst enemy. Absent a coherent design, unanticipated requirements will require not just new code, but many a rewrite of code that is scarcely out of the debugger. Your wheels spin. *The art of programming*, as opposed to the mere act, *lies in postponement*.

Delay writing code for as long as possible. Draw sketches on napkins, or invest in an expensive equivalent such as Sybase PowerDesigner—whichever your method and budget, it is almost always correct to postpone the actual coding. Write detailed pseudocode.[5] If the slightest ambiguity remains in your specification, it is too soon to code.

To fully model an entire software project, you need more than paper and pencil, more even than a tool like PowerDesigner or Rational Rose. *You need a software lifecycle methodology*, a lifecycle process for requirements and technical issues:

- a full set of consistent concepts and models
- a collection of rules and guidelines
- *a full specification of all deliverables*
- a workable notation with good drawing tools
- a set of tried and tested techniques
- a set of appropriate metrics, standards and test strategies
- define organisational roles e.g. requirements analyst, software architect, programmer
- guidelines for project management and quality assurance

Our concern here is the third bullet in that list—full specification of all deliverables—and how to turn that document into a correct database. You can tackle it head on as we describe *here*:

1. Document all required tasks down to every detail,
2. From [1], list all required attributes and actions,
3. Group attributes from [2] into entities, normalised tables, and relationships,
4. Turn all actions from [2] into queries, stored routines or application pseudocode,
5. Create a physical database from [3] and [4], and seed with test data,
6. Iteratively test and revise the database until no more errors are found.

Or you can model the job more creatively. Two main modelling approaches have evolved, *data flow* and *use case* (Figs 5-1, 2). They combine nicely:

1. *Identify the actors* (persons or other systems) who will interact with the system,
2. *Specify the tasks* that the system must perform, including all required outputs,
3. *Map actors and tasks to use cases* until all requirements are accounted for,
4. *Model use cases as entities, tables, and other database objects and routines*.

Then you work out a user interface (UI) that your client can live with, and *then*, finally, you code the application. When you finally get round to coding, you find that most of the code writes itself straight from your requirements model.


Coping with impatient project managers

The real world has bulimic budgets, deadly deadlines, manic managers and political pressures. Thanks to the perfectionism and optimism that made you a software developer, demanding managers have you over a barrel. How to cope?

1. A core managerial strategy is to divide time-required estimates by 3 or more. To compensate for this and avoid arguments, multiply your time-required estimates by 3 or more before you submit them (an instance of a famous rule for consultants: begin with the bad news).

2. From client information specifiy all the software's required outputs, and get management to sign off on your list before you estimate the cost. Analysing all required outputs yields lists of all required inputs, and protects you from being blindsided by the inevitable "we need just this one more report" that breaks the specification two days before you're due to go live.

3. Remember that eventually we are all replaced, so when faced with impossible demands, decide whether you prefer to be replaced before failing to meet the impossible demands, or after.
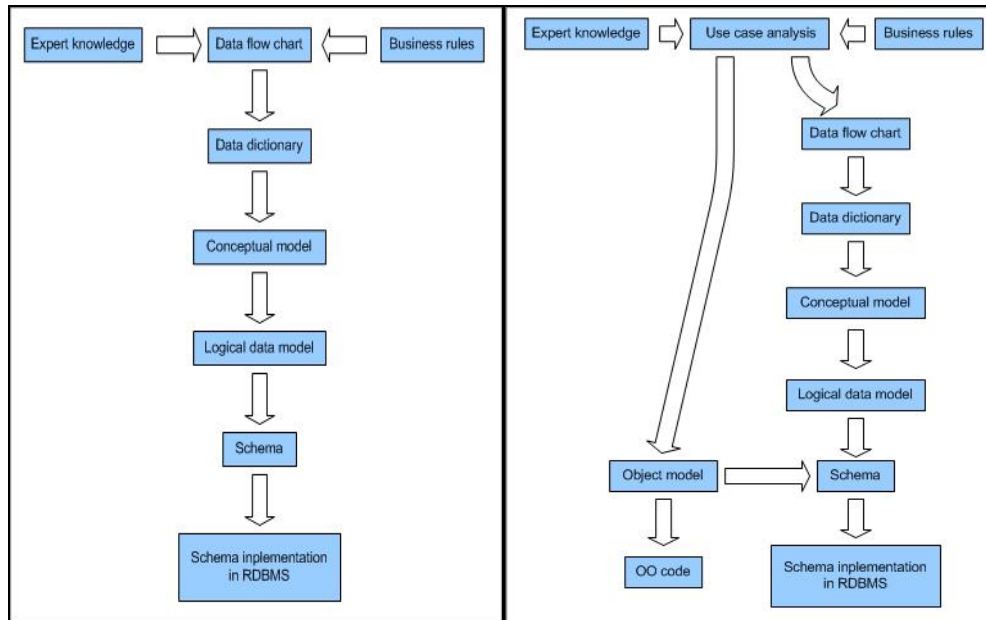
Figs 5-1, 5-2: Data flow chart and use case modelling of database problems

Yes, there is debate about postponing UI design this long. Some think such postponement may well wreck the UI. To find out why we disagree, read on.

Modelling tools

You have many possibilities, ranging from scribbling on a napkin, to writing out the requirements in ordinary language, to bulleted lists, to homemade sequence and structure diagrams, to spreadsheets, to auto-generating the database from a diagram built with a modelling tool. Unified Modelling Language (UML), *released* in 1997 by Rational Software Corporation and the Object Management Group, is the leading modelling language; even if UML is more than you need, we recommend Terry Quatrani's excellent introduction[1] to it. If you have access to a UML modeller listed at *http://www.uml-forum.com/tools.htm*, you will probably want to use it to design your databases.

Commercial database modelling tools are too rich for many budgets, but some are free or affordable. Version 1.4.x of *Clay Database Modeler*, open source and user-friendly, runs as an *Eclipse* plugin, and supports moving databases to and from many RDBMSs. For how to get it and set it up see "Eclipse, an elegant IDE for MYSQL" on *our MySQL Tips page*. Fig 5-3 shows a Clay model of the *Northwind database*. MySQL offers *MySQL Workbench*, also free. It derives from *DBDesigner 4* and is now stable. To produce the equivalent of Fig 5.3 in MySQL WorkBench, click on Database | Manage Connections to create a connection, then click on Database | Reverse Engineer and follow the prompts. For this book we used Datanamic's *Dezign for Databases*. It is neither free nor expensive, it does not do UML or use cases, but it makes GUI data modelling very easy, and instantly generates robust SQL scripts from its models.

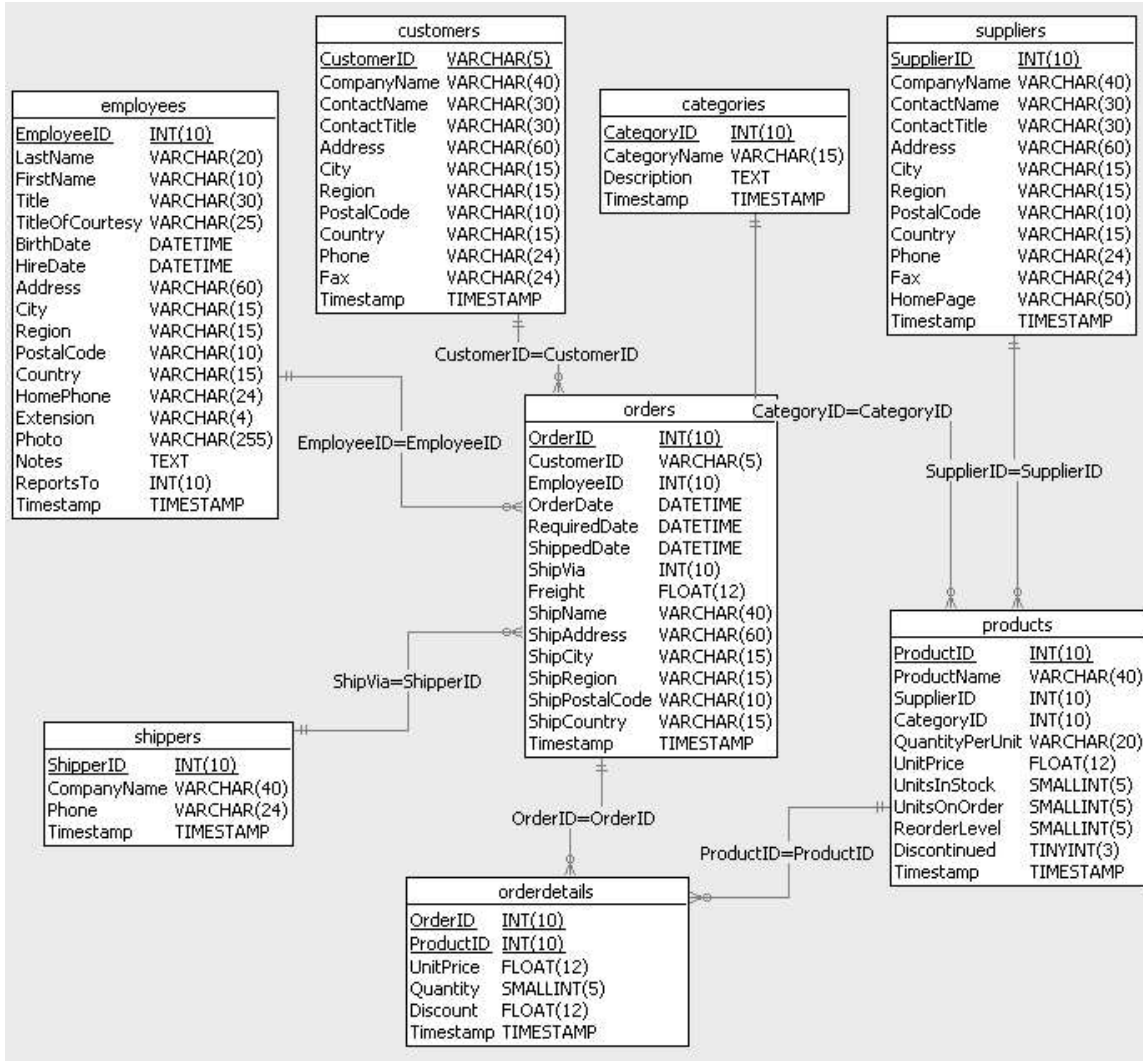You may be muttering, who has time to write formal models?

Fig 5-3: Northwind schema as seen in the Azzurri Clay modeller in Eclipse

Well, as the ad for auto engine oil says, you can pay now, or you can pay later. Pay now, and you have a reasonable chance of delivering the application on time and within the budget. Postpone the payment, and you can count on repeated code and database revisions, and an increasingly unhappy client.

A sample project

To illustrate how this four-step modelling works, we apply it to a small-scale sample requirement, for a web application that tracks contracts, projects, and their revenues and expenses, for freelance technical professionals.

A prospective user might be a software developer. Or the user could be a photographer, or a graphic artist, or a carpenter, or an investment advisor, or a florist or interior decorator—in short, anyone who professionally undertakes jobs for other parties, and who needs to track those jobs on a computer. Or any group of such professionals, for example a co-op. Contractors will be able to budget and track tasks and costs, and invoice and track payments, on a per-project basis.

An early bonus is that since the software will serve as bookkeeper for contracts, it will be able to use its bookkeeping smarts to manage its own user accounts. It will track and bookkeep projects for many kinds of contractor-client networks—cooperatives, neighbourhoods, business networks, skills exchanges and so on.

An architecture professor gives his students a blueprint of a house, saying that the client wants to move this wall two meters to the left, to make room for a grand piano in the adjoining room. The professor asks the students what it will cost to move the wall. An enterprising lad suggests that it depends whether the wall is load-bearing: if it isn't, then the cost is say $500; if it is, multiply by 10.

The professor replies, "Good, but wrong. The cost is $1.25. At $125 an hour, the architect cuts the wall and pastes it into its new location, consuming 30 seconds at most."

The longer you postpone coding, the better off your client will be. Rushing into implementation is the signature of the amateur.

Step 1: Identify the users

A user is any *person* or *system* that interacts with the software--a definition that for some projects will be wide and complex, but for this sample project means just human users, who will come in three main flavours: *contractors*, *clients*, and the *owner* of the app.

Step 2: Define the software tasks

What must the software do? A good answer to this question frames the whole analysis. The method is to start with a general list of application tasks. That list yields a preliminary list of tables that the project will need. Then we drill down *until we have discovered all the details in each task needed to assemble any required output of the software.*

In the sample project, these application jobs are: to process and store information on *clients*, *contractors* and their *projects* including details of those projects such as *tasks*, *charges*, *invoices*, and *payments*. Right away we can see that the task list is not a higgledy-piggledy job collection, but a neat matrix: for each of the entities we just listed, the software has to do at least five things:

- **add** or **create** the entity, eg a *user* or *invoice,* under specified validation constraints
- determine **user access** to the entity
- list instances of the entity -- ie **browse** with the capability of selection
- **maintain** the entity under specified validation constraints -- edit, copy, or even delete it
- **report** the entity

This first job list gives a preliminary list of the tables we need. There are eleven—users, contractors, clients, addresses, professions, skills, projects, tasks, charges, invoices and payments. So we have defined 55 software tasks (Table 5-1)—add, access, browse, maintain and report each table. (Are we going to build this entire project in this book? Relax. No. We use it mainly to illustrate basic database and query design principles.)

Table 5-1: Matrix of application entities and basic tasks

Entity	Add	Access	Browse	Maintain	Report
parties	add party	registration path, or party adding client or contractor	browse, lookup contractor or client	party edit, update	admin only
users	add user profile	user login, or party	browse	user account	global for admin, or of party, contractors, or clients
addresses	add address	party or delegate	browse	edit, update	probably admin only
professions	add profession	ditto	ditto	ditto	global for admin, or of parties, contractors, or clients
skills	add skill	ditto	ditto	ditto	ditto
projects	add project	ditto	browse, lookup, selection	ditto	one-off, for contractor or client
tasks	add task	scoped to project	scoped to project	ditto	of project
charges	add charge	ditto	ditto	ditto	one-off, for party, contractor, or client
invoices	create invoice	ditto	ditto	(careful here!)	ditto
payments	add payment	ditto	ditto	(careful here!)	ditto
roles	add role	admin only	browse	edit, update	list

As we drill down, the matrix may grow or shrink, and may get ragged on its right edge. We can tinker with the matrix's rows. Are clients and contractors necessarily different? No. Party A may be contractor to party B in project X, and may be client to contractor party C in project Y. We want the system to model any party as contractor or client. Client and contractor are *roles*, not entities. The basic entity is a *party*.

Next, with contractors and clients collapsed into parties, we notice that addresses, professions and skills are "children" or subtables of parties, just as tasks, charges, invoices and payments are children of projects.

Now, how do *users* and *parties* relate? The software should use its bookkeeping smarts to manage the relationship between the user and the application owner, so using the software puts every user into a direct or indirect client-contractor relationship with the application owner. A party known to the system may use the software only if the party has enrolled in the system. All users, then, are enrolled parties, or delegates of parties. But there will be parties who are known contractors or clients yet who are not users. And some parties may be enrolled, but may delegate use of the software to subordinates, for example employees. So some parties are not users, but enrolled parties *have* users.

We may not yet understand all nuances of who is who, but we have learnt this much:
- a party may *be* contractor and client in different contracts,

- enrolled parties *have* users; in fact having a user defines enrolment.

So in the first pass at an entity list, *contractors*, *clients* and *users* have become *parties* and *users* (Table 5-1, column 1), and we are down to ten entities. Five basic tasks for each yield a starting (or perhaps startling) list of 50 application tasks, and some not spotted yet. More than 50 tasks to analyse! Wasn't this supposed to be a small project? Fortunately, many tasks will be virtual copies of other tasks. For example, every task of browsing party or project subentities must scope to the respective party or project (Table 5-1, column 4)—opportunities to use one object or module to accomplish several tasks.

To model how all system actions mesh together, and how the software will contol the flow of action states, we will need quite a few meetings with prospective users, and if a client hired us to write this, with the client. We may need to build UML activity diagrams. All that is beyond our present scope.

We made a start on column 1, the entities. What about the task columns? *Add*, *browse*, *maintain* and *report* are straightforward. What about *access*? A user profile defines what a user may do in the system. At one extreme, everyone in the enterprise might be able to execute process A; at the other, only the CEO may execute process Z. In most enterprises, these permissions are cumulative as you move up the hierarchy. A project manager should be able to enter a new city name in a `cities` table, for example, but so should any data entry person in the enterprise. Only a manager can adjust a final balance.

For every user known to the system, must we keep track of 5x10=50 or more permissions? That would be a maintenance nightmare. There is a need for encapsulation under basic organising concepts. There are three main general approaches[4]: role-based, rule-based, and mapping via a special-purpose access policy language. Role-based approaches can accommodate rules if desired, and do not require language extensions, so we group permissions into *roles*. Permission management by role definition has a name, *role-based access control* (RBAC), and a *literature of its own*. RBAC defines both *user roles* and *session roles*. In this first first-pass model we begin with four *user roles*:

- the party-level *clerk*, who can do simple data entry on behalf of one party,
- the party-level manager or *owner* who can make more advanced party-level decisions, for example set budgets or suspend charges,
- an overall *administrative clerk* who can assist all clerical users, and
- an overall *application owner* or manager who can see and do everything possible in the system.

For simplicity we name the first two of these levels *clerk* and *owner*, and the latter two *admin clerk* and *admin owner*. But we do not set these four roles in stone. Or in code. *We implement them as data*, so we can add more roles later, as the need arises. Individual parties may wish to define specific user roles, and we will likely have to differentiate the overall *application owner* role into its component parts, for example database administrator (DBA), database maintener (DBMAINT), system security officer (SSO), system operator (SYSOPR), and application manager (APPMGR).

By analysing a column in our task matrix, then, we discovered a heretofore unnoticed entity, `roles`, and added that row to the task matrix.

At the end of Step 2, we have a plausible matrix of tables and tasks. We are ready to drill down to the details.

Step 3: Specify use cases

In his classic book "Object-Oriented Software Engineering"[2] , Ivar Jacobson introduced *use cases* to the development world. Put simply, a use case describes *one task that an actor can perform with the software*.

Like many brilliant ideas, this one was so simple and obvious, everyone had overlooked it. Once Jacobsen had stated it, however, it almost instantly became an important method for writing software requirements. Now use cases are an important UML feature.

Developers of large, complex and expensive software almost always employ use cases. Unfortunately, developers of smaller and less expensive systems often skip this stage, believing that the time and effort required is not available within the budget. We beg to differ. If you don't have a full description of the system's functionality, how will you code it, how will you know when it works correctly, how will you know when it is finished? How will you know which capabilities are critical, which are nice-to-have, and which cannot be delivered on time? Often the largest problem of software development is managing the expectations of your users. Use cases are the most effective way to do that.

There are two kinds of use case: *essential*, for design, and *real*, for prototypes or actual builds. Here we are concerned with the design phase. In their excellent book on UML[3], Fowler and Scott offered this definition of the use case:

> *A use case is a typical interaction between a user and a computer system [that] captures some user-visible function. [and] achieves a discrete goal for the user.*

It is a story, or case, of one specific use of a system. The word *user* is more intuitive here than *actor*. We go with *user*.

We could take Fowler's definition to absurd extremes: the user types the A key, and the system responds by adding the letter A to the document. So, *typical interaction* has to be undertood with common sense. If you like, insert a word like *significant* before *user-visible* .

And right away we have an opportunity to apply some of that commonsense. Every task implies an *access* use case: is the user permitted to do this, or not? We are not going to list all access use cases separately. Thanks to RBAC, access will map to user roles, maintenance of which can also be analysed as one use case. We will treat just one access use case, *user login*, separately. Otherwise we analyse access (Table 5-1, column 3) only where it needs attention. Similarly we know in advance that every table we derive from this analysis will need *maintenance* (Table 5-1, column 4) and therefore a maintenance use case where permitted users may add, modify and report table contents under the system's constraints. We will take up maintenance use cases in detail only where they present particular issues (hoping that sooner or later the system will acquire a module that automatically generates, for each of its tables, a maintenance interface implementing all the system's rules for that table.)

What goes into the analysis of a use case?

A use case describes how a user will interact with the system. For example when a user logs in:

- the system offers input fields for username and password
- on submission, the system validates user input
- if validation passes, the system logs user in
- otherwise, the system offers choice of re-login or registration

As we analyse use cases, to simplify later collection of column and table requirements we will mark items that need to be database columns `thus` when they first come to our attention, and their tables **`thus`**. In most cases, the table to which a column belongs will be obvious. In some cases it will not, or what is initially obvious will change as a result of normalising the database.

Use case analysis is not for the literal-minded. We already saw that access, user roles and maintenance for most or perhaps all tables can be abstracted as single use cases. Not all remaining use cases need separate analysis either. Because our focus here is on database design, we break use cases down only as far as needed to reveal consequences for database structure.

Nor need we begin at the beginning of the software use cycle and proceed lockstep to the end. True, there is an intuitive advantage in sussing out use cases by walking a prospective user from login to session termination. But restricting ourselves to that sequence may obscure use case structure. So we will remember to look for opportunities for generalisation, and we begin at the top left corner of the task matrix (Table 5-1), with what creating principals and creating users have in common--the basics of creating a **`party`**.

Use case: add a party

Under what circumstances may a user add a party?

- when the user is enrolling in the system as a party, or
- when the user wishes to add the party as a client or contractor.

Remember that adding a party makes the new party a potential contractor or client. The system displays a form for filling out `name`, `company name`, **`addresses`**, `email address`, `voice phone`, `fax phone`, `cell phone`, and **`professions`** and **`skills`** offerings. **`Addresses`**, **`professions`** and **`skills`** will have add/edit forms that must be available to the *add party* page. Any party may have multiples of each. When enrolling, the party being entered is to be a user, so after party information is complete, the page must send the user to

- the *add account* page to establish the party's contract or project with the admin owner, and
- the *add user profile* page.

So adding a party may invoke up to five other use cases. When the user submits input, the system validates the input. If it passes, it creates a new party with at least one address,

zero or more professions and skills entries, and zero mor more user rows. If the input fails validation checks, the user is returned to the form. Every input form has to do all this. To minimise repetition, we call it *validation/submission*.

Use case: add a user profile

A user may get to the *add user profile* functionality

- from the task of enrolling in the system
- from the task of adding a user

The user profile consists of `username`, `password`, `user role`, and a `key` pointer to a row in the delegates or parties table. The system determines the value of key from the identity of the party whose page the user has arrived from, and puts up username and password input fields for the user to fill out. It determines `role` from the same page:

- if the page data was of a party, the user is an *owner*
- if the page data was that of a delegate, the user is a *clerk*
- if the page data was that of the application owner, the user is *admin owner*
- if the page data is an admin owner delegate, the user is an *admin clerk*

More elaborate role definitions are left as exercises for the reader.

And here we find another unanticipated use case. How is the system to know that a just-registered user is actually a clerk for existing user so-and-so? Obviously, only the user's *manager* can specify this, and must have done so before the clerk logs on, so we make a note that user profile maintenance must include, for managers, a form for *pre-registering clerks* that permits entry of enough information to permit the clerk to log in.

Use case: user login

User login determines a candidate user's access to the software. A **user**, registered or not, arrives at the application entry page. If the user is registered, she selects the dialog for registered users and submits her `name` and `password` for login. The system validates the name, password and account `status`, and logs her in, noting her user role, if she passes validation checks.

Validation fails if the user is unknown, if the password is incorrect, or if there is a problem with the account (overdue fees, inactive for some other reason). In the latter case, the user may proceed to the user account page to straighten out the access problem.

The user who has not yet registered may select the user registration process, or visit a page describing the purpose of the web site (a use case not shown in Table 5-1, and which makes that matrix ragged by adding only to the *users* row).

So the *user login* use case has these elements:

- prompt user for username and password
- login if authenticated, otherwise re-prompt
- connect to *user account*, *registration info* and *account profile* pages

Use case: user account page

The account page should be available from the user profile page, and from the user login page when there is an account problem that blocks login. Logically, a user account is a project with the *admin owner* as contractor and the user/party as client. Its subprojects (tasks) will be subscription terms. It can handle multiple **invoices** and `payments`, specifies the user's payment information, which the user may update, and shows account status (active, inactive, suspended, etc). Sensitive information is stored in encrypted fields.

Obviously this use case is an instance of the *review/maintain project* use case, which the system should implement in a perfectly general way. The user account page will then be just a call to the general account review page, specifying the user as client and the admin owner as contractor. We leave that general design until we are doing project use cases.

Use case: administer user roles

Obviously, role administration use cases will be available only to *admin owners* and *admin clerks*, who have to be able to determine

- permissions for every role for every use case, and
- user role assignments for every user

A basic RBAC provides a set of defined user roles (`roleID, name, rank`). Rather than assume that our list of fifty-something use cases and four user roles is set in stone, the system should offer to admin owners and admin clerks dynamic cross-tabbed updateable views of users, roles and use cases with add, edit and report functionality built in, permitting editing and addng of roles, and assignment of roles to users and to use cases.

Use case: permission check

This is related to user role adninistration. For each application activity that a user wishes to execute, the system must determine whether that user's role permits that action. Permission checking will use the database, but does not imply any additional columns or tables (unless the client needs to keep records of permission checks, but we aren't going there yet).

Use case: pre-register additional party user

This is a variant of the *add user profile* use case, with these differences:

- a user who is a *party* is creating a user profile for a subordinate or co-worker

- the new user profile will 'inherit' all the party's projects, or a subset of them; it is the party's responsibility to make these assignments

- the new user profile will 'inherit' all the party's permissions, or a subset of them; again the party makes these assignments

- a party may wish to define clerk role classes for grouping project permissions; we make a note that project and user tables will need a nullable `clerk group` attribute

All we know, at this point, is that somewhere in the application, *owner parties* need access to this functionality. The details, we leave for later.

Use Case: Contractor selects project

Most user activities are scoped by the project to which they relate, so to begin work on a project, a user must select the project of interest. On being asked, the system provides browse access to all projects for which the user has permission, and provides an option to select a project or create a new one.

Use Case: Contractor adds project

The user indicates a desire to create a new project. In database terms, this means adding a row to the **projects** table. The system responds by displaying a form for the user to enter the project `name`, `client`, `budget estimate`, `commitestimate`, `chargesToDate`, `startingdate`, `targetcompletiondate`, `description`, and a nullable `externalkey` column for optionally relating the project to an external tracking or accounting system.

When the user is specifying the project client, the system presents the user's client list, and also provides the option to create a new client. Here is an example of one use case making use of another. Rather than describe the task of creating a new client as part of the current use case, we simply mention that it provides this option.

A project has one or more tasks, each with its own estimate, so the *add project* use case implies a *task browser* use case (**tasks** table: `projectID`, `taskID`, `name`, `description`, `subcontractorID`, `datecreated`, `createdby`, `startDate`, `targetdonedate`, `completiondate`, `signedoffby`, `signoffdate`, `estimate`), with the capability to add, modify and void.

Note the simplifying assumption, which could come back to bite us later, of there being one subcontractor at most per task. And estimates change, so lurking behind the task browser use case is *estimate change tracking*, which we implement with the simple concept of budget items.

Use case: contractor adds budget item to project task

To cost a project task, a contractor must be able to add and edit task budgets. A simple model is to have a **budgitemtypes** table—for example with types='original', 'change' and 'final', but we can leave this entirely to the users—and a **budgetitems** table which records the item type, amount, date entered and so on, to which a user will write an entry for every addition or update of a task budget. The project management interface, then, needs to provide convenient browse access to task-scoped budget items.

Use case: contractor adds or edits profession or skill entries

Skills vary within professions, so treating their use cases as equivalent is suspect. Database design can be an obsessive's delight—until the obsessively designed system collapses under the weight of its own quibbling. For simplicity's sake we let `skills` stand free, and we provide users with a way of linking a `skill` to a `profession`.

Now we see that a party

- is a potential contractor if the system has `professions` or `skills` for the `party`,
- is obviously a contractor if there are `project`s where the `party` is contractor.

When a party indicates a desire to add a profession or skill, the system will present a list of the party's registered professions or skills, if any, and an opportunity to add to either list from a lookup list of available possibilities, with options to select from or to add to the lookup list, and to link a skill to a profession. This use case thus becomes several: for each of `professions` and `skills`, the system permits browsing one's own list, adding to it, deleting from it, browsing the master list, adding to the master list, or (subject to permission) editing the master list. On this last use case, can party B edit party A's master list entry? Complexities loom. We opt for the simplest possible solution:

- a profession or skill has a flag determining whether other parties may modify it
- a profession or skill added by an *admin owner* or *admin clerk* is unmodifiable

Analysing use cases for professions and skills, uncovered the specifications for these tables: **professions** (profID, profName, profDesc, modifiable), **skills** (skillID, skillName, skillDesc, modifiable), **partyprofessions** (partyprofID, partyID) and **partyskills** (profID; partyskillID, partyID, skillID).

Use case: Contractor adds charges to a project

The user indicates a desire to add charges to the selected project. The system responds by displaying a form displaying `charges` information (date, chargetype, unit, quantity, subcontractor if any, invoiceno if any, description, amount). The user submits the form, and the system adds the charge to the selected project.

Again another use case turns up: *charge type selection* (e.g., labour, materials, expense, rental, adjustment, etc), so we need a `chargetypes` table (chargeTypeID, name, description, code, externalCode). And that simple approach hides the deeper problem of meta-chargetypes, e.g., category (labour, materials), type (original, adjustment), above or below the line, which we also leave as an exercise for the reader.

Use case: Contractor suspends a charge

For audit integrity, the system prohibits charge deletion, but a contractor must be able to move charges between `charges` and `suspendedcharges` tables. A suspended charge has all charge attributes plus suspensedate, suspendedby, suspensereason, resolution. This can also lead us into a set of related use cases and their subtables (for example `sentcorrections` and `resolvedsuspensecases`) which we also defer for now.

Use case: Contractor resolves suspended charge

The system permits the contractor to write off a suspended project charge, to re-post it, to adjust it before re-posting it, or to return it to its source for further processing, providing documentation in each instance.

Use case: Contractor creates and sends an invoice

Enter project, amount, date, and client. Select the project task items to invoice. The system prepares the invoice and on approval

- prints or transmits it,
- adds an invoice record for it.

The `invoices` table thus has `invID, projectID, createdby, invdate, amount,` and a child table `invoicelineitems` with `invitemID, taskID, description, amount, qty, unitprice, tax, discount.` Relational purists may be appalled that we risk skew by storing an easily calculable `invoice.`amount, but accountants will be appalled if we do not, and accountants control our pay cheques.

Use case: Contractor enters a client payment

Enter payment `amount, invoicenumber, date, payment method, authorisationnumber, authorisationdate, authorisationmethod, authorisedby.` The system enters it in the `payments` table. This use case implies another use case: `paymenttypes` (purchase order, cash, credit card, on account, etc).

Use case: Report revenue and expenses

The bottom line use case—report revenue, expenses, and their details, for the beancounters . Here, unexpected details often turn up. Although not central to this chapter, it is a use case to analyse very carefully in real-life application design.

Use case: Report projects and their status

List all project attributes for all or specified projects.

Reviewing the Specification

We do not imagine that each of these use case specifications is perfect or final. They illustrate the process, is all. With them, the application rapidly takes shape. We can almost see the web pages required, because use cases are driving them. It's a good time to go to bed, get some sleep and prepare for morning's harsh light.

Given that this sample project does not have a real client, we phone a few colleagues and arrange some walkthroughs.

Brando, a graphic artist, points out that most of his gigs are flat-rate, $300 no matter how

long it takes. Very occasionally he gets an hourly rate. We have this covered.

Meryl, a website developer, works for a regular hourly rate. We have her case covered.

Isabella, a freelance architect, points out that in her business, the project description might be a hundred printed pages, plus several graphics printable only on an expensive plotter, plus an AutoCAD VR-walkthrough. Make a note to include an `infopath` column in the projects table.

Marla, a private investigator, points out that in her business you keep your head down and your mouth shut, and you definitely do not enter your client list on the internet. She is not a prospective user.

Asanga, a WannaBeGuru (Linux), points out that the most important thing in his life is figuring out how to finance his daughter's admission to Harvard. To him, we can only suggest that our app may not be helpful, unless he is prepared to define his daughter as a project. We'd rather not go there.

Pat, a Husserlian psychologist (that discipline is like proscuito—best sliced thin), adds that she bills many clients at unpredictable rates. As fans of and participants in academe, we are more than receptive to this perspective. We have her use case covered.

Andrea, a Celtic jewelry artist, objects that there are no projects, there is only art. "I do it all, right down to the layout of the newsletter, which I email to my list in PDF format." We have her covered: subcontractors are optional.

Glen, a jazz saxophonist, points out that on most of his projects he is a sideman, while on a few he is the principal. Each project has a Principal (what we have called a `contractor`), this much is obvious. On each project there are zero or more subcontractors. Now we begin to appreciate Glen's problem. On Project P he is the lead, and you work for him. On Project Q you are the lead and he works for you. As you log your hours on various projects, you must select the project. The system should know whether you are the principal or the sideman. But at the end of the day, you want to know how much money will arrive before the end of the month from all projects. Do we have this covered? Yes. When Glen is principal, he can set himself up as contractor and set up each sideman as the contractor in a project subtask. When he is sideman, he needn't concern himself with all that--as far as he is concerned, he is the contractor and the principal is his client.

Several colleagues remind us that there are persons and companies. This amounts to deciding whether **parties**`.name` or **parties**`.company` is the name of interest.

Step 4: Map and derive the database structure

Now from the case analyses we derive *entities*, their *attributes*, and the *relationships* between entities. We may start with an entity-relationship diagram (ERD) from which

we deduce tables, columns, keys and relationships. Or as in Fig 5-4 we can go straight to the tables. Either way, the result defines all *tables*—their names, *primary keys* (PKs), other columns, and relationships expressed as *foreign keys* (FKs) referencing other tables.

A *relationship* between tables A and B defines how A and B relate via their keys:

> **1:1**: for a PK value in A there is at most *one* matching key value in B, *and vice versa;*
>
> **1:many**: for a PK value in A there can be *man*y matching FK values in B; for each matching row in B, however, the FK value matches *one* matching PK value in A;
>
> **many:many:** a row in A can match *many* rows in B, and *vice versa*. An M:M relationship needs a bridge table C where each row in C refers to one A row and one B row.

With most modelling tools, you begin with the table names and PKs, then add in the columns (for example `username` and `password` for the `users` table), then add the relationships between entities. Even if your tool is a text editor, this approach works best.

Starting then at the top left corner of Fig 5-4, roles may consist of multiple use cases, a user may play multiple roles, and roles may be played by multiple users, so `roles` is 1:M with `usecases`, and `roles` and `users` table are M:M, mediated by the `userroles` table's pair of 1:M relationships with `roles` and `users`.

The `users` table will have a double 1:M relationship with every table that tracks modifications by recording the `userID` of the user who adds (insert privilege) or modifies (update privilege) a row. Since users modify the users table, it has a reflexive relationship with itself—that is, every row has an `entered_by` value that points to another `userID`.

To the right, **`parties`**, with everything except its PK and name columns exported to **`addresses`**, **`party_professions`**, and **`party_skills,`** has a double 1:M relationship with **`contractor_client`** to model contractor-client relationships. In turn, a **`contractor_client`** row may be referenced by multiple **`projects`**, which in turn may have multiple **`project_tasks`**, each of which has a pointer to a row in **`tasktypes`**. Similarly, in the top right corner **`addresstypes`** has a 1:M relationship with `addresses`.

The **`professions`** table is the system's list of professions, and has a 1:M relationship to the **`skills`** table. Notice its relexive relation, to model the possible dependency of one profession on another. The system will keep **`party_skills`** in a table where each row points at one **`skills`** row and one `parties` row, and **`party_professions`** in a table where each row will point at one **`professions`** row and one **`parties`** row.

Project task budgeting is modelled by the fact that any **`project_tasks`** row may be referenced by multiple rows in **`budgetitems`**, which in turn has a 1:M relationship to **`charges`** to permit tracking of charges against task budgets. To handle charge errors, allow movement of charges to a **`suspense`** table, and thence to **`correctedcharges`**. **`Invoices`** may have multiple **`invoiceitems`**, each of which points to a row in **`project_tasks`**. The **`pmt_types`** table has a 1:M relationship to **`payments`**, to which **`invoiceitems`** has a 1:M relationship to support partial payments.
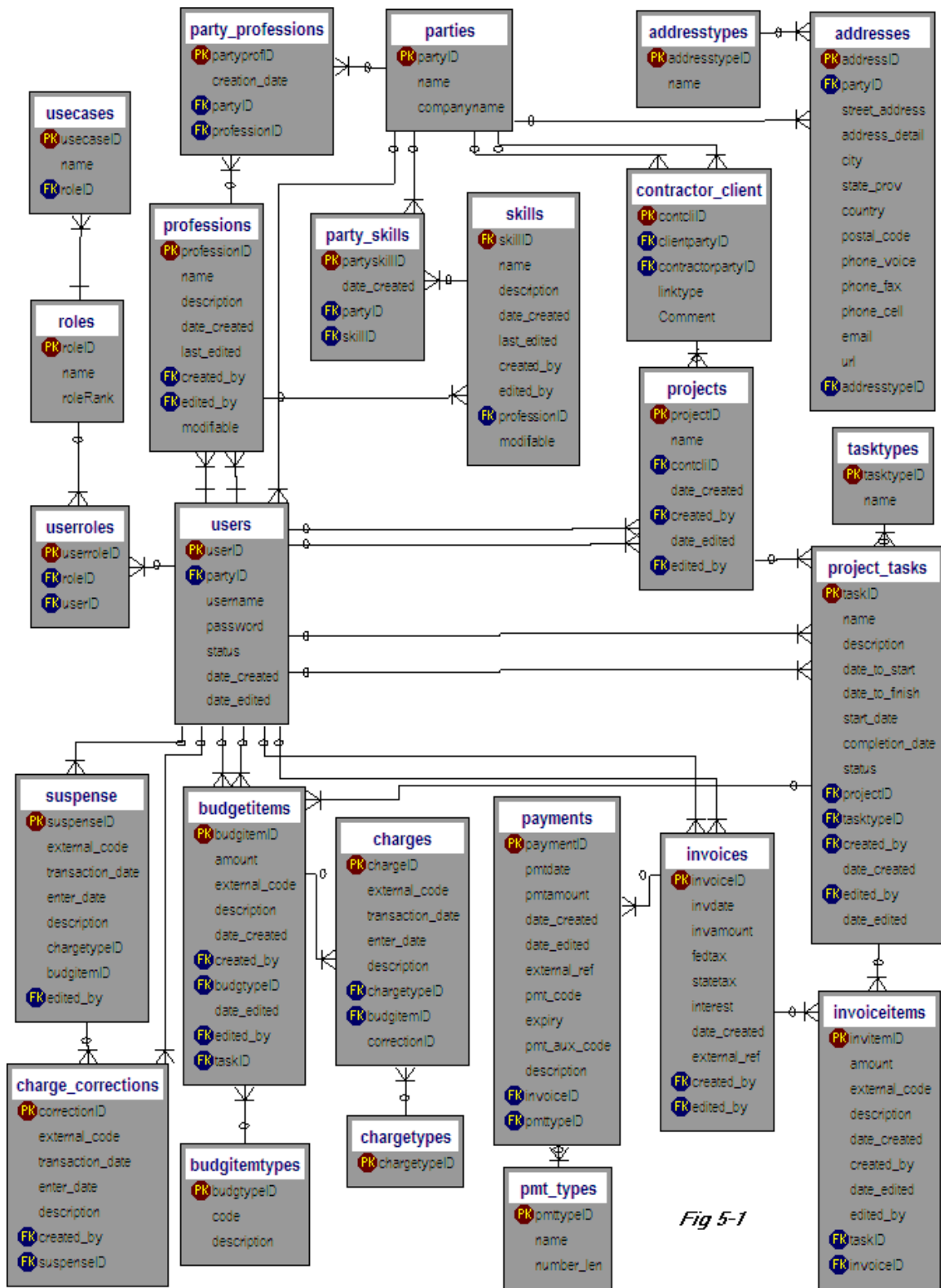
Figure 5-4: Schema for tracker database

Testing and correcting the database model

Test one is to create the actual database. If we wrote the spec with a database modelling tool like Dezign that can generate DDL, we can create the entire database script with a button click. Otherwise, we have to write the SQL scripts ourselves. However we get it done, with script at hand we fire up a GUI MySQL client and tell it to run `tracker.sql`, or we hand the script to the command-line MySQL client with this:

```
mysql -uUSR -pPWD <tracker.sql
```

If the server bellyaches about errors, we correct the charts and script and try again, over and over again if necessary, till we get it right..

Test two is to enter representative data. Run:

```
mysql -uUSR -pPWD <tracker_pop.sql
```

Then write the queries for every required application output--invoices, customer and sales lists, inventory lists, or whatever the application will be required to produce. This is the very best way to find errors in your model. Notice the implication for the application development trajectory: *write all queries before writing any application cod*e.

Test three, too often skipped, is to build detailed sequence/flow diagrams for each use case, and verify that the database receives and returns exactly what is expected.

Test four is to translate those diagrams into pseudocode. Amazingly, even at this stage, you will find holes that have to be plugged.

Structuring the application

Can you now just write the code? Not quite. Where is the code to run? What code runs on the client? On the server? In middleware? How many tiers is the application to have? Here, there are two contrasting philosophies to choose from, for guidance: n-tier design, and 2-tier design with stored procedures.

N-tier design provides flexibility. A correct model should let you swap out the frontend (user interface), the business logic or the database without requiring much change to other layers. The front end communicates with the business logic layer, which in turn communicates with the database. There may be another layer between business logic and database to translate for a particular DBMS. The frontend never directly communicates with the database. The business logic layer uses a generic API to communicate witht the database, whose job is conceived, mainly, as simply information storage.

But no software executes in zero time so this flexibilty costs performance. This is where stored procedures usually make their pitch, inviting you to embed business logic in the data layer for a gain in performance. An excellent trade, usually, but not yet with MySQL, whose stored routines are not compiled.

Still, distiguishing between business logic and database logic helps. Whether a customer gets credit or not is a business decision; *that decision belongs in the business layer*. Whether an project item may be entered without a foreign key tying it to the `projects`

table is database logic; *that code belongs in the data layer*. Before any code is written, you will save yourself many later hassles if you take the time now to get very clear on how your application is to implement this distinction.

Plan reporting now

Information worth saving in a database will almost inevitably need to be reported across years or decades. *Pentaho* and *Jaspersoft* make good open source ETL tools that work well with MySQL.

Summary

In this chapter we modelled specification analysis as it relates to database design. The four modelling steps are (1) identify users, (2) identify tasks, (3) analyse use cases, and (4) derive required tables. We applied the model to a small sample project that we will revisit to create the actual database in Chapter 6, to design some queries in Chapter 9, to illustrate PHP, Perl, Java and .NET application development in Chapters 12 through 15, and to look at writing RBAC-based security routines in Chapter 19.

References

1. Quatrani, Terry. Introduction to the Unified Modeling Language. *http://www.rational.com/media/uml/intro_rdn.pdf.*

2. Jacobson, I. Object-Oriented Software Engineering. 1992. Addison-Wesley, Reading, MA.

3. Fowler, M & Scott, K. UML Distilled. 1997. Addison Wesley Longman, Inc, Reading, MA.

4. Gebel, G. Access Management: Is there a role for RBAC? (no longer publicly available).

5. Dalbey, J: Pseudocode Standard. *http://www.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html*