Object-Oriented Analysis and Design

Understanding System Development with UML 2.0

Student Exercises

Mike O'Docherty
Manchester University, UK

John Wiley & Sons, Ltd

Contents

Introduction

Each exercise has four stages:

- Plan: Decide what the tasks are and whether there should be more than one iteration.
- Organize: Put tasks in order or in parallel and assign responsible persons.
- Do: Perform the tasks themselves.
- Reflect: Think about what's been achieved.

At the start of each exercise, every member of the team should read through the exercise to get a feel for what needs to be done. Then the team should spend around ten minutes planning and organizing.

Exercise 1

Business Modelling

Before you start producing a software system, you need to understand the business that will use the software. For a large business, you may only be able to model the area of the business that your system will automate.

Use case modelling, in a simplified form, is a good way of understanding and documenting the way a business operates. A business use case model comprises a list of actors, the use cases themselves and a glossary.

Remember that in business modelling, as in any other iterative phase, there is no strict order in which to do things. Therefore, this exercise (and most of the following ones) should be viewed as steps in a first iteration. Read the entire exercise before you start and then feel free to reorder the steps or perform them in parallel, according to the taste of your team members.

If you need to brainstorm a use case, try to use communication diagrams or activity diagrams to sketch out ideas.

Goals

At the end of this exercise, you should be able to:

- Create and maintain a project workbook.
- Choose and assign team roles.
- Start a glossary.
- Understand the business and system requirements from an informal problem statement.
- Work with customers to understand the business in detail.
- Identify and describe business actors.
- Identify and describe business use cases.
- Add detail to your use cases.

1.1 CREATING A PROJECT WORKBOOK

Take an empty binder and label it ‘Project Workbook’ (you might like to choose a team name at this point and write that on the binder too). This binder will contain all the official documentation of your project team, so that you will always know where to look for the latest detail on any part of the system.

As the work progresses, you can add new documents to your workbook, update existing documents or add entirely new versions of documents. Don't throw any documents away – it's useful to maintain a history trail. It's also a good idea to designate one person as 'Workbook Keeper' whose job it is to ensure that the workbook always gets a copy of new or updated documents. (The Workbook Keeper should also make sure that every team member gets a photocopy of the completed workbook at the end of the project.)

Ideally, we would like the maintenance of the workbook to be automated as much as possible – using some sort of CASE tool perhaps. However, for the purposes of this exercise, paper will be sufficient – it would take too long to learn how to use a CASE tool.

1.2 DECIDING TEAM ROLES

Add a divider labelled 'Team Roles' to create a new section in your binder to hold the results of this part of the exercise.

1.2.1 Invent Roles

Before you go any further, you should invent and assign some more team roles.

We already have Workbook Keeper and, implicitly, 'Team Member'. However, you will be working to tight time limits so you must organize your team in order to avoid chaos (and to make sure that you have a solution to each exercise).

Some roles to consider are:

- **Project manager** Someone who makes sure that:
  - the team produces plans and schedules
  - everyone is clear what their responsibilities are
  - progress is reviewed regularly.
- **Time Keeper** Someone who makes sure that the team keeps to the schedule.
- **Guru** Someone with a particular area of expertise.
- **Scribe** Someone who makes sure that the results of brainstorming are recorded on paper (for the Workbook Keeper to put into the workbook).
- **Glossary Keeper** Someone who is responsible for keeping the glossary up to date, adding definitions for new pieces of jargon as they arise and cross referencing terms with other documentation.

1.2.2 Assign Roles

You should share out your selected roles (fairly) and record them in the Team Roles section of your workbook. You should also rotate roles regularly – perhaps on a daily basis or per exercise.

1.3 ADDING WORKBOOK SECTIONS FOR THE BUSINESS MODEL

Add a 'Business Model' section to your workbook to hold the results of this exercise, except for the glossary.

When adding items to your workbook, leave lots of room for each entry. This will allow you to modify or add comments to individual items while you iterate, without creating new versions of documents. For example, leave half a page for each actor, two whole sides for each use case and so on.

1.4 MAINTAINING A GLOSSARY

Add a 'Glossary' section to your workbook.

1.4.1 Add New Terms and Synonyms

Every time a piece of business-specific jargon crops up, add a short description of it to the glossary. Also, every time you discover a synonym for an 'official' term, add that too.

1.4.2 Cross Reference the Business Model

When you add an item to the glossary that relates directly to the business model, make a note of that alongside the item.

> `Customer` (business actor, business object) Anyone who rents cars, browses car models or reserves car models.

1.5 CONSIDERING THE PROBLEM STATEMENT

Your customers (the department of Computer Science at Blue Sky University) have given you a problem statement in the form of an informal discussion document (see Appendix A). Read the discussion document now and make sure that you understand it.

As the exercise progresses, remember that you are not the customer. So don't make any assumptions about the customer's business, instead, ask them! (Your tutor will be happy to play the role of the customer.)

1.6 PRODUCING AN ACTOR LIST

Write down all the human and/or subsystem roles in the department's student assessment business. Remember to concentrate just on the existing business, not on how the business will operate when you have built the new AQS system.

If necessary, reduce your list of candidate actors to a minimal set. (Any rejected actors can still be synonyms in your glossary.) Add a short (one sentence) description for each actor.

1.7 PRODUCING A USE CASE LIST

List all the business use cases for the student assessment business. (Each use case should start with an actor and give value to one or more actors.)

Reduce your list of use cases to a complete minimal set, discarding any unused ones. Add a short description to each, for quick reference.

1.8 ADDING DETAILS TO THE USE CASES

Although business use cases can be described using one or two paragraphs of natural language, it's often clearer and less ambiguous to provide a sequence of steps instead.

Detail each of your use cases with a list of steps (around 10 steps should be enough).

Exercise 2

System Requirements

Use case modelling in its fullest form is an excellent way of describing the functional requirements of a software system. Use cases themselves also provide a convenient location for recording most non-functional requirements.

In this exercise, you'll build a complete use case model for the Automated Quiz System comprising actors with descriptions, use cases with descriptions and details and a use case diagram. You'll also record supplementary requirements that don't fit with any particular use case, produce some sketches of your user interface components and ensure that your glossary is up to date.

Once again, don't make any assumptions about what the customer wants out of the system: ask them. (This applies to non-functional as well as functional requirements.)

For simplicity, you can assume that the following interfaces will be modelled elsewhere:

- A tool for creating and editing quizzes.
- A tool for setting up a sitting.
- A tool for collating results from a sitting.

In other words, you only need to model in detail the function of the tool that a student sees when they sit a quiz. (You can still refer to the facilities of the other tools at a high level, in order to add context – for example, you can include them in your actor list, use case list, use case diagram and use case prioritization, but not in your use case details or user interface sketches.)

To save time, you do not need to produce a use case survey (unless you want to).

Goals

At the end of this exercise, you should be able to:

- Identify system actors.
- Identify system use cases.
- Compose a use case diagram.
- Write system use case details.
- Record supplementary requirements.
- Produce user interface sketches.
- Prioritize use cases.

2.1 ADDING WORKBOOK SECTIONS FOR THE SYSTEM REQUIREMENTS

Add ‘System Use Case Model’ and ‘User Interface Sketches’ sections to your workbook.

2.2 IDENTIFYING SYSTEM ACTORS

- Produce a list of candidate actors for AQS.
- Write a short description of each AQS actor.

2.3 IDENTIFYING SYSTEM USE CASES

- Decide which use cases AQS must support.
- Provide a short description of each use case.

2.4 SHOWING THE WHOLE SYSTEM AS A USE CASE DIAGRAM

Draw a diagram showing all AQS actors outside the system (shown as a rectangle) and all the use cases inside. Connect the actors to the use cases in which they are involved.

Can you spot any opportunities for ‘include’, ‘extend’ or ‘specialize’ relationships between your use cases? Could you build relationships by introducing some new use cases?

If you can see ideal opportunities for introducing new use cases or relationships, do so now. But try not to introduce too much complexity just for the sake of reuse.

Can you think of any ‘specialize’ relationships between actors (perhaps introducing new ones for this purpose)? Selectively introduce new actors and relationships.

2.5 DETAILING YOUR USE CASES

- For each use case that specializes another, indicate that fact at the top of the use case.
- Add any preconditions or postconditions that you can think of. If there are no preconditions, just write ‘None’. Since every use case is supposed to provide value to one or more

actors, there should always be postconditions along the lines of 'the value was provided' or 'the failure was reported'.
- Add the individual steps in the execution of the use case, indicating where other use cases are included or may be used to extend the current one.
- Add any abnormal paths to a list at the end of the use case.
- Whenever a non-functional requirement fits a use case, add it in a section at the end of the use case. (Non-functional requirements may come from the problem statement or you may discover them for yourself, perhaps in discussion with the customers.)

2.6 RECORDING SUPPLEMENTARY REQUIREMENTS

In your workbook, add any non-functional requirements that do not relate to a particular use case to a 'Supplementary Requirements' page in your 'System Requirements' section. (These requirements may come from the problem statement or you may discover them for yourself.)

2.7 PRODUCING USER INTERFACE SKETCHES

For each user interface in your system, sketch the main window. You will have one user interface for each actor who retrieves information from the system, adds new information or directs the system to do something.

Most user interfaces have sub-windows that appear as needed (they may pop up or they may overlay the main window). Decide which sub-windows you need for your user interfaces, how they will appear and what they will look like.

For each user interface, illustrate your use cases by showing the order in which the main window and the sub-windows appear in a typical interaction.

2.8 PRIORITIZING REQUIREMENTS

Place your use cases in order of implementation priority. Use green amber and red markers to indicate the desirability of each of your use cases, and any supplementary requirements, for the first release. (Customer wishes should be the most important consideration here, followed by spiral planning.)

Exercise 3

Static Analysis

During analysis, we try to identify and describe all the business entities involved in the eventual system, along with their attributes and relationships. This is the purpose of static analysis, which we'll be looking at in this exercise. As a further discovery and verification process, we also perform dynamic analysis (see Exercise 4).

As you go through this exercise, remember to keep up to date the project glossary and the non-functional requirements (in both the Use Cases section and the Supplementary Requirements section).

Goals

At the end of this exercise, you should be able to:

- Identify analysis classes representing business entities.
- Identify relationships and multiplicities between analysis classes.
- Show analysis classes, relationships and multiplicities on a class diagram.
- Find analysis class attributes.
- Update the project glossary and non-functional requirements.

3.1 IDENTIFYING CLASSES

3.1.1 Produce a List of Candidate Classes

Read through your system use cases, listing the nouns that you come across. This list forms your initial set of candidate classes.

Add any classes that you discovered during your earlier brainstorming. (Since you were thinking about the possible system when you produced them, they may well be relevant.)

Do you think that it would be worthwhile re-examining the problem statement or the business model at this stage? (Probably not, because the problem statement and the business model cover much more ground than the specific, focussed system use cases.)

3.1.2 Filter your Candidate List

Examine each candidate class and reject it if it is:

- The system itself: This will become many separate classes.
- An actor with no special information: We don't need to model human actors or system actors unless we process information about them internally.
- A user interface: We will reason about interfaces during dynamic analysis, but we won't actually model them in detail until design.
- Trivial: This could be a trivial type (such as a string) or one that's likely to become an attribute (such as a question number).

While you're considering which classes to keep and which to reject, further classes will occur to you: add these to the list of candidates and then apply the same filtering rules to them.

For each class that has survived your filtering process, write its name on a sticky note or index card (this will make it easier to prototype your class diagram on a white board or chalk board in the next step).

3.2 CONNECTING CLASSES USING RELATIONSHIPS

3.2.1 Connect Related Classes

Draw relationships between pairs of classes that seem to be strongly connected. At this stage, just show inheritance and association – you can refine the associations in the next step if appropriate.

If there is a candidate for inheritance, consider introducing abstract classes as necessary (for elegance and for future extension). Whenever you are tempted to use inheritance, make sure that it satisfies the following criteria:

- It makes the diagram clearer and therefore easier to understand (especially since non-technical customers may see the diagram).
- It is not there to share implementation (sharing implementation is a design issue, not an analysis issue).

If you find a class that is associated with an existing relationship between two other classes, consider making it an association class.

3.2.2 Refine the Associations

Some of your associations may be modelled more precisely as aggregation. Can you find any examples?

Do you think any of your aggregations qualify as compositions? (Hint: it's probably too early to tell at this stage.)

3.2.3 Describe the Relationships

For every relationship, except inheritance, you should now consider adding some kind of description:

- An association name, describing what the association represents.
- A source role, describing the role of the source.
- A target role, describing the role of the target.

(You don't need to provide all three – provide just the association name or the role names, as you prefer.)

In the case of aggregation (or composition) you may decide that the description is obvious and, therefore, you may choose to omit it.

3.2.4 Add Multiplicities

Specify the number of objects involved in each relationship (except inheritance, which is implicitly one-to-one). Specify multiplicities in one of the following ways:

- `m` – Exactly `m` objects.
- `m..n` – From `m` to `n` objects, inclusive.
- `m..*` – At least `m` objects.
- `*` – Any number of objects, including zero (it is shorthand for `0..*`).

3.3 ADDING CLASS DESCRIPTIONS TO YOUR GLOSSARY

Once you have agreed your basic class diagram, add a short description for each class to your glossary. Some of them may already appear; as long as the existing description is still appropriate, you can simply specify that the entry is also an analysis class.

3.4 ADDING ATTRIBUTES

Now look for the attributes (properties) for each class of object. Record each class's attributes on a separate sheet of paper. Avoid recording attributes that can be derived from other attributes (for example, a Circle has a radius and a diameter, but we could pick either as an attribute and the other would be derived).

On the same piece of paper, give each attribute a short description.

Specify the type of each attribute, as appropriate. For any particular attribute, you may not know the type at this stage or you may not want to specify the type until design.

Stick to a simple set of types for your attributes, such as the Java primitives (int, float, char, boolean) and simple classes (e.g. String). If you're unfamiliar with other Java types, ask your instructor.

Exercise 4

Dynamic Analysis

Now that you've completed your first pass through static analysis, it's time to use dynamic analysis to verify what you have discovered. This will involve composing communication diagrams (use case realization), listing and describing operations and building a state model for taking a quiz.

As you go through the dynamic analysis process, you will probably discover mistakes or omissions in your class diagram, so you should correct it as you go – but try very hard not to re-engineer it completely. You will also be inventing controller and boundary classes, so you should add descriptions for them to your glossary.

Goals

At the end of this exercise, you should be able to:

- Verify an analysis class model.
- Realize use cases.
- Draw communication diagrams.
- Identify and describe operations.
- Build and draw a state model.

4.1 REALIZING USE CASES

4.1.1 Pick a Use Case

Pick a use case to realize. It's a good idea to start with the first use case a student executes when taking a quiz and then work forwards through the main flow of the other use cases; once you've finished with the main flow, you can do the peripheral use cases.

4.1.2 Draw a Communication Diagram

Use cases usually start with an actor telling a boundary object (a user interface) to do something or to retrieve some information. In such a case, you should draw the actor and the boundary with a line between them indicating a communication path. Next, label the actor and boundary with their names and classes (the classes may be sufficient). Then, draw a message being sent to the boundary by the actor. Proceed through the use case step by step, inventing message flows between objects to satisfy the use case steps. Don't forget to add any new controllers, boundaries, entities or homes to your glossary.

The following tips may help you draw the diagram:

- Make your boundary objects large, encapsulating a number of related tasks (they can be decomposed during design).
- Ensure that entity objects are in your class diagram.
- Introduce controllers whenever you need to centralize the control of a complex business process.
- Introduce homes (special-purpose controllers) whenever you need to create an entity or retrieve an entity from a pool (database) of previously created entities.
- Use abbreviations for parameters and then use notes to spell out the abbreviations.
- Show the assignment of return values only when they are not obvious (you will provide more detail in the operation descriptions).
- Name values only when they are used elsewhere in the diagram (for example, if a return value is subsequently used as a parameter).

4.2 DETAILING THE OPERATIONS

List each class (entities, interfaces and controllers) on paper, one class per side.

For each class, walk through the communication diagrams and record every message sent to an object of that class as an operation. As you do this, be sure to show the types of the parameters and return values (if any). Occasionally, you won't know the types yet or you won't want to commit yourself until design – in such cases, just leave the type blank.

Bear in mind that parameter and return types should match the attribute types you chose in the static analysis phase – for example, if you've specified that a `Student`'s library number is a `String`, it should be a `String` whenever it's used as a parameter or returned as a value.

For each operation, add a short sentence or two describing what the operation does, what its parameters are for and what it returns (if anything).

4.3 BUILDING A STATE MODEL

Think about what happens when a student takes a quiz. For example, you may decide that the student first needs to log on, then answer some questions, then get their results and so on. How would you model this life cycle as a state machine? It may help you to consider the GUI sketches for the quiz tool that you produced earlier.

List all the states in the quiz life cycle (restrict yourself to a single student taking a quiz rather than an entire sitting). It may help to think of the states a student goes through from the system's viewpoint. Draw each state as a round-cornered box.

Walk through your life cycle showing how the student moves from one state to another with transitions corresponding to the commands they give the system and the questions they ask.

Verify that your state model is correct by walking through a couple of typical scenarios of a student taking the quiz.

Exercise 5

System Design

In this exercise, you'll design the architecture of the AQS system. This involves making decisions about the physical machines involved, the subnodes that run on each machine, the packages you need to partition your classes logically and the dependencies that exist.

You'll also need to think about concurrency issues, the technologies you're going to use to help you implement the system and how you can make your system secure. Security includes making the system safe from hackers who try to attack the server; making the system private so as to frustrate eavesdroppers who try to decode Internet traffic; and protecting client and server machines from accidental damage by your software.

As you go along, remember to update your glossary if you reuse any existing terms or introduce any new ones.

Goals

At the end of this exercise, you should be able to:

- Design a system topology.
- Select appropriate technologies.
- Plan the judicious use of processes and threads.
- Model processes as subnodes.
- Decompose a system into layers.
- Group related classes into packages.
- Design for security.
- Design for concurrency.
- Draw a UML deployment diagram.

5.1 CHOOSING A SYSTEM TOPOLOGY

5.1.1 Decide how many Tiers Are Needed

Decide whether your system will have one, two or three tiers. (Remember that each tier may contain any number of physical machines.)

Bear in mind that you will need at least two tiers because early system requirements dictated that:

- Questions should be retrieved on demand.
- Each answer should be submitted as soon as it is given.

In other words, you could not, for example, implement a single-tier system with all the software and questions provided on CD and the answer books collected by e-mail.

5.1.2 Decide on the Machines Involved

Decide how your tiers will be distributed across physical machines. Because of the open-ended system requirements, you should avoid introducing constraints relating to the type of hardware needed (e.g. PC, workstation, mainframe).

For simplicity, ignore any complicating issues such as load balancing, replication, fail-over, redundancy and backups.

5.1.3 Draw a Simple Deployment Diagram

Draw a simple deployment diagram showing only device nodes – each node represents a physical machine in a typical configuration of the system. For each node, you should choose a type and you should also indicate multiplicities.

5.2 SELECTING TECHNOLOGIES

Choose which technologies will be required on client machines. Be careful not to place unreasonable constraints on the owner of the client machine (e.g. complexity of installation or complexity of connecting to the next tier). Record your choices in writing in the System Design section of your workbook.

Choose the technologies that will be used on your other tiers and record your decisions in writing.

If necessary, decide which protocols will be used between your tiers (you may be constrained by the technologies you've already chosen). Add details of the protocols that will be used to your workbook.

5.3 CHOOSING SYSTEM LAYERS

Select the layers that you need to partition the system logically. Do you think you need any vertical subsystems? Would you need any if you had been considering the needs of the Lecturer and Overseer actors in detail? Record your basic layers using a layer diagram.

If you feel it's appropriate or necessary, introduce one or more translating layers. Add these layers to your layer diagram.

Make decisions about how the layers will communicate (e.g. how control flows, whether events are used, whether layers are open or closed). Record your decisions on your layer diagram.

5.4 CONSIDERING PROCESSES AND THREADS

Decide which processes must, or should, run on each node. Add the processes that you've designed, or discovered, as execution-environment subnodes on your deployment diagram.

Decide whether any of your processes should use threads explicitly or, indeed, whether any of them uses threads as a side effect of some other technology that you've already decided to use.

Record in your workbook whether you will introduce any threads and whether any of your layers will need to be thread-safe (either because of your threads or because of one of your chosen technologies).

5.5 PACKAGING CLASSES

Group your classes into related packages. You will not know yet exactly which classes you'll end up with, but you should still be able to invent sensible groupings for them. Place each of your packages in the appropriate position on your deployment diagram.

5.6 COMPLETING THE DEPLOYMENT DIAGRAM

Decide how you will deploy your system as artifacts (e.g. web server folders, Java archives and database schema). Add these to your deployment diagram.

On your deployment diagram, show dependencies (use and manifest) between nodes, subnodes, packages and artifacts, as appropriate.

5.7 THINKING ABOUT CONCURRENCY

Write down, in your workbook, how you intend to control concurrent access to quizzes (and their parts), sittings, students and answer books.

Write down how you will deal with concurrent access to student information stored in the Library System.

5.8 THINKING ABOUT SECURITY

What steps can you take to protect your server from attack by hackers? For example, a hacker might wish to steal your intellectual property or crash your server. Record your findings in your workbook.

Write down how you intend to prevent eavesdroppers deciphering sensitive information passed over the Internet.

Write down how you will reassure the administrators of your client machines that your software couldn’t possibly do them any harm.

Exercise 6

Subsystem Design

Now that you have defined the AQS technologies and layers in system design, you can turn your attention to the internals of the layers and the communication between them. This exercise has three separate steps:

- Transforming analysis classes into design classes for the business logic, adding fields as you go; this step is documented using a class diagram.
- Transforming design classes and fields into a database schema and depicting the tables using straightforward notation.
- Defining business services, designing server classes and examining the interaction between the client, the server classes and the business logic. This is documented using class diagrams and sequence diagrams. It will also help you to discover messages for your business logic classes.

As ever, you should update your glossary where appropriate.

Goals

At the end of this exercise, you should be able to:

- Transform analysis classes into design classes, taking account of extra classes, excess classes, attributes, compositions, associations, association classes, inheritance and universal identifiers.
- Define a generic database schema as a direct mapping of design classes.
- Discover business services.
- Classify business services and introduce supporting classes as necessary.
- Show object and layer interaction using sequence diagrams.

6.1 DESIGNING THE BUSINESS LOGIC

6.1.1 Choose Classes and Fields

Using your analysis class model as a guide, choose which design classes you need. Add them to a new class diagram.

On your class diagram, for each class, add any fields that you need to record attributes. Remember that each entity class should have, at least, a universal identifier. Don't forget to indicate the visibility of each field (using $+$, $-$, $\sim$ or #).

If you discover a piece of information that needs to be stored for an entire class of objects, add a class field to the class. Class fields are underlined to distinguish them from instance fields.

6.1.2 Add Relationships

Add the relationships between classes (inheritance, aggregation, composition and plain association) and multiplicities to your class diagram. For composition and aggregation, you can retain the diamond symbols as extra documentation.

Every non-inheritance relationship should show the directions of the relationship as one or two arrow heads.

The name and visibility of fields used to record relationships can be indicated next to their corresponding arrow heads; for relationships implemented as collections, you should add the collection class name as well.

6.1.3 Update the Glossary

Add descriptions of the business logic classes to the glossary. You may already have glossary entries for these classes, as a result of analysis for example, so you may only need to indicate that these entries also correspond to design classes.

6.2 DESIGNING THE DATABASE SCHEMA

6.2.1 Agree a Notation for your Schema

Agree a notation within your group for describing a schema. For example, you could show a table in the following text-only format:

```
CARMODEL
+ID:INTEGER, NAME:VARCHAR(256), >MAKEID:INTEGER
```

In this notation, + indicates a primary key and > indicates a foreign key.

6.2.2 Decide on SQL Types

Choose a handful of SQL types to use for your column data. For example, you should be able to make do with:

- INTEGER: A 32-bit number.

- DECIMAL: A real number, of appropriate precision (only use decimals when essential, INTEGERs are usually better).
- VARCHAR(X): A string, of X characters.
- BOOLEAN: A 1-bit value – 0 for FALSE, 1 for TRUE.
- DATE: A day in the Gregorian calendar.
- TIME: A particular combination of hours, minutes and seconds within a Gregorian day.
- TIMESTAMP: A combined DATE and TIME.

These generic SQL-92 types vary from one database implementation to another, but it's reasonable to assume that similar types are available.

6.2.3 Introduce Entity Tables

Add a table to your schema for each persistent business entity. Use the universal identifier as the primary key. Add foreign keys for the persistent one-to-many and one-to-one relationships.

6.2.4 Introduce Link Tables

Add link (association) tables for the persistent many-to-many relationships and persistent association classes.

6.2.5 Update the Glossary

In your glossary, add references to your database tables.

6.3 DESIGNING THE BUSINESS SERVICES

6.3.1 Determine the Business Services

With reference to the GUI sketches, determine which abstract services must be provided to the client tier by the middle tier and list these in the project workbook. Each business service in the list can be as simple as 'Reserve a Car Model'.

Group the business services into messages on concrete server classes. For each message, you'll need to determine the parameters passed by the client and the result returned by the server, if any.

You are free to decide exactly how information flows back and forth (e.g. lightweight copies or proxies). However, you should make sure that:

- The client never passes information to the server if the server already has that information (in its database).

- The client is never given the same piece of information twice (in other words, the client must cache information).

Universal identifiers may help you here. You may wish to relax the rules a little to handle authentication using session keys, since this mechanism may rely on redundant information.

6.3.2 Draw Sequence Diagrams

For each business service, draw a sequence diagram that shows message flow from client GUI to server classes and on to business logic classes. This will serve as final verification of the feasibility of the architecture.

For simplicity, you should try to show only the messages that are sent between objects, rather than messages sent by an object to itself. Do not show any details of interaction between the business logic and lower layers, although interaction between business logic classes is sometimes appropriate.

If any interaction is too complicated to show pictorially, you might consider adding a brief note instead.

6.3.3 List Business Logic Messages

In the project workbook, list the messages that you have discovered for the business logic classes, complete with parameters and return types. You can read these from your completed sequence diagrams.

If you were using a CASE tool, the messages could be added directly to your design class diagram. You are unlikely to have room to do this on paper.

Once you've finished this process, you should have a minimally complete set of server classes and business logic classes, with fields and messages. These classes can be taken forward to the specification stage.

6.3.4 Update the Glossary

In the glossary, add references to the server classes.

Exercise 7

Class-Based Specification

To finish off your development, it's time to dabble in the noble art of specification. We don't expect you to be particularly formal or exhaustive, but we would like you to think about preconditions, postconditions and invariants as they apply to everyday classes. Also, you should think carefully about how you can use checked and unchecked exceptions to signal error conditions to client software.

Goals

At the end of this exercise, you should be able to:

- Define pre-conditions and post-conditions for messages.
- Define class invariants.
- Choose how to use exceptions.

7.1 PICKING CLASSES

Choose a server class to specify. Preferably, this should be the largest one you have. Server classes are good candidates for specification because the server needs to protect itself from the client by having a strong and rigidly enforced contract.

Select a business logic class as well. Business logic classes often have a good set of invariants. Again, the class you choose should be large enough that you can say something interesting about it.

7.2 ADDING INVARIANTS

Take two clean sheets of paper, one for each of the selected classes. Write the name of each class at the top of the paper and then list any invariants you can think of (recall that invariants indicate obligations for both buyer and supplier).

You should find it easy to produce invariants for the attributes of an object (i.e. values available externally via setters and getters).

You can write your invariants using any combination of natural language and OCL-like or Eiffel-like notation – as long as they're clear, they will suit your purpose.

7.3 ADDING PRECONDITIONS

Write down the messages of the classes, listing the preconditions as you go. Remember that a precondition is a buyer's obligation.

Preconditions may relate to parameters, public attributes or the result of a `boolean` method (as in `checksOutOk(aParam)`).

You can assume that an invariant is automatically a precondition.

7.4 ADDING POSTCONDITIONS

Add postconditions for the messages. These will typically refer to reply values and object attributes.

7.5 PRODUCING AN EXCEPTION STRATEGY

Where do you think unchecked exceptions might come in handy? (Hint: An unchecked exception can be used by a supplier to signal that the buyer made an inappropriate request or that an internal error – a bug – was encountered.)

Where could you use checked exceptions? (Hint: Checked exceptions cannot be ignored by the buyer, so they are good for building firewalls against users or external systems.)

Appendix A

Discussion Document for the Automated Quiz System

A.1 PROJECT BACKGROUND

Traditionally, a student's performance on each taught module in the Computer Science Department's Masters course can be assessed in two ways:

- By written work before, during and after the module, which is marked by the lecturers or demonstrators.
- By a written exam, which is taken by the student under formal invigilated conditions and then marked by the lecturers.

The lecturers may choose what weight is given to each form of assessment or, indeed, whether one of the two forms is dropped altogether.

For audit purposes, copies of all assessment material and marking schemes must be presented to the department's Postgraduate Office for safekeeping. Exam material is subject to peer review. After the completion of each module's assessment or exam, as appropriate, student marks are presented to the Postgraduate Office.

A.2 PROJECT AIM

The department is keen to automate some of the assessment process, not just to reduce workload and therefore costs, but also to remove the subjectivity inherent in traditional marking.

To begin with, the department would like to automate the written work of suitable courses by introducing an electronic multiple-choice testing tool – the working name for this tool is the *Automated Quiz System (AQS)*. If AQS proves successful and it can be shown that invigilation in the context of networked computers can be achieved reliably, the testing tool will also be used for exams (where feasible and desirable).

In keeping with the department's objective of promoting distance learning, it is necessary for AQS to be Internet-enabled – registered students should be able to take a test from their home PC, with all the questions and results stored in the department. Making AQS Internet-capable will also, of course, make it suitable for use within the department.

A.3 QUIZ CHARACTERISTICS

It is expected that an AQS quiz will comprise a sequence of single-choice and multiple-choice questions, to be answered in a set time.

Each question will contain text describing the problem to be answered and, optionally, pictures that provide further information.

There should be a list of two or more textual choices for the student to select from. In order to get a question right, the student must select all of the correct choices and none of the incorrect ones. (To make things a little easier, the student should be told whether there is exactly one correct choice or more than one.)

Students can be shown their score as soon as they submit their paper or their time runs out.

A.4 PROJECT CONSTRAINTS

Due to the reliability and performance problems of the Internet, the following policies will need to be adopted:

- Questions must be downloaded to the client on demand – i.e. Question 1 is downloaded initially, then Question 2 is downloaded when the student has answered Question 1 and so on. (As a further optimization, the questions can be cached on the client so that they do not need to be downloaded more than once.)
- Each answer given by the student must be submitted to the server immediately. That way, if the client or network crashes part way through a sitting, at most one answer will be sacrificed.

Each test will have a maximum amount of time for answering the questions – the test must stop automatically if the maximum time runs out. Students should be able to stop the test early if they wish. Due to the informal nature of assessed work, it would be appropriate for the student to be told their score for the test immediately the test is finished – this facility may need to be disabled if AQS is used for examinations in the future.

A.5 PROJECT PLATFORM

The department has many networked computers, ranging from large Unix servers through PCs to small network computers, running a variety of operating systems. To make matters worse, Internet clients used for distance learning could run almost any combination of hardware and software.

As a result, no assumptions can be made about client machines and, for the sake of portability and flexibility, as few assumptions as possible should be made about server capabilities.

All software used for the project must be cheap, preferably free, because the department is a non-profit organization.

Considering the project platform, it is envisaged that open and royalty-free Internet technologies will be of great benefit. These technologies include Java, Web browsers, Web servers, HTTP and TCP/IP.

A.6 INTEGRATION WITH THE LIBRARY SYSTEM

The University Library is, separately, commissioning a project to export their existing Student Information System over the university LAN, using CORBA as the middleware. As soon as possible, AQS should use this new facility to access student names and library numbers.