A Process for 2IM24/2IM25

© 2013 TU/e, WIN Version 1.5

1 Introduction ... 3
1.1 Technical part of the process ... 3
1.2 Management part of the process ... 5
1.2.1 SPMP ... 6
1.2.2 SQAP ... 8
1.2.3 SCMP ... 8
1.3 Example ... 9
2 User Requirements phase ... 10
2.1 Example: User requirements ... 12
3 Software Requirements Phase ... 14
3.1 Example: Initial logical model ... 15
3.2 Example: Extended logical model ... 15
3.3 Example: Sequence diagram ... 17
3.4 Example: Screenshots ... 18
4 Architectural Design Phase ... 19
4.1 Example: Architectural model ... 20
5 Detailed Design Phase ... 22
6 Implementation ... 23
6.1 Example: High-level view ... 23
6.2 Example: Detailed view ... 24
7 Test Phases ... 25
7.1 Unit Tests ... 25
7.2 Integration Tests ... 25
7.3 System Test ... 25
7.4 Acceptance Test ... 25
8 Concluding remarks ... 27
References ... 28
Appendix A: The Coffee Machine ... 29
Appendix B: Overview of phases and deliverables ... 32

1 Introduction

This document describes a specific software development process for the *Software Engineering Project for the Minor Programs* (2IM24/2IM25). Many books have been written about software engineering, e.g. [1] and [2]. The process described here is a very much simplified and adapted version of the Software Engineering Standards of the European Space Agency (ESA) [5], which is a relatively compact standard. The process is divided into several phases: the User Requirements (UR), Software Requirements (SR), Architectural Design (AD), Detailed Design (DD), Implementation and Testing phases. Previously, this course was given as 2IM22; this is the first year this project is given as 2IM24 and 2IM25. As this is also the last year this course is given, it was decided not to change the name of the course website.

Although the deliverables that accompany these phases should be delivered consecutively, this does not mean that the phases are executed strictly sequentially. For example, some implementation work will generally be done early in the process, to eliminate risks late in the process. Participants of the 2IM24/2IM25 projects generally have little programming experience. Therefore, it is important that some team members start as soon as possible with experiments involving prototypes that explore possible solutions for the products to be delivered. The knowledge obtained with these prototypes ensures that the implementation will not suffer major delays due to *this* gap in the knowledge of the team. In general, it is important to identify and eliminate risks as early as possible.

It is important that team members spend their time as evenly as possible from the beginning of the project, spending approximately 1/3 of the available hours during the first quartile and 2/3 of the available hours during the second quartile. The reason to spend less time during the first quartile is that, in general, the early phases require more feedback which slows down the process. Teams tend to spend too few hours in the beginning of a project as the deadline is still far away. However, when the project team is too slow in the early phases of the project, this usually results in a lack of time at the end. This, in turn, results in loss of quality and/or loss of implemented functionality. Therefore, a schedule is set in Table 2 of Appendix B. The schedule contains hard deadlines that will be strictly adhered to and soft deadlines that serve as guidelines.

Sections 1.1 and 1.2 give an overview of the technical and management aspects of the process. In the chapters that follow, the phases and the contents of the deliverables of the phases are described in more detail. Due to the limited amount of time available during 2IM24/2IM25 projects, not all phases need to be (fully) executed (see also Appendix B: Overview of phases and deliverables).

1.1 Technical part of the process

The UR and SR phases are concerned with requirements (i.e. the “what”), whereas the AD and DD phases are concerned with implementation (i.e., the “how”). A starting point is a short informal description of the customer’s wishes. The role of the customer in 2IM24/2IM25 is played by the teaching staff.

During the UR phase, requirements are elicited from the customer. A logical (UML) model based on these requirements is constructed in the SR phase. Modeling the requirements serves several purposes: the model is used to check if the requirements are complete and to ensure that the team understands the problem domain. The logical model also serves to model the external behavior (functionality) of the system to be built: a stimulus from the environment can be "followed" through the model until a response is produced (see Appendix A: The Coffee Machine).
In the SR phase, a prototype is constructed as well. The prototype models the user interface. This way, both the functionality of the system to be built and the user interface, i.e., all external features of the system are described. The prototype can consist of drawings, screenshots or even a working program with limited functionality.

During the AD phase, an architectural (UML) model is designed. The system is described in terms of a number of modules (components, packages), each of which has a well-defined purpose. The interfaces of each module are described and a logical model is designed for each module. After the interfaces and the functionality of all modules are designed in detail, the modules can be constructed by independent teams as both the external interfaces and the functionality have been decided upon. During the DD phase, the internal structure of each module is designed in detail. In the Implementation phase, the modules are implemented. In the SEP projects for the minors, The DD and Implementation phases are combined for practical purposes. During the Testing phases, the implementation is tested

After a module is implemented, it can be tested in isolation by a Unit Test (UT). When modules are completed, they can be assembled and tested in combination – the Integration Tests (IT). Assembly and testing takes place in the inverse direction of the dependencies between the modules. The running example used in this document consists of three modules:

1. the database which does not depend on any other module,
2. the database access package which depends on the database, and
3. the main (user interface) package which depends on the database package and, therefore, indirectly also on the database itself.

Assembly in this case is to first assemble and test the database and the database access package and then to add the main package and test the entire system. After the IT, a System Test (ST) follows in which the entire functionality is tested. Finally, the Acceptance Test (AT) is executed by the customer. If this test passes, the system can be transferred to the customer in the Transfer (TR) phase and the project ends. For 2IM24/2IM25, this process is somewhat simplified.

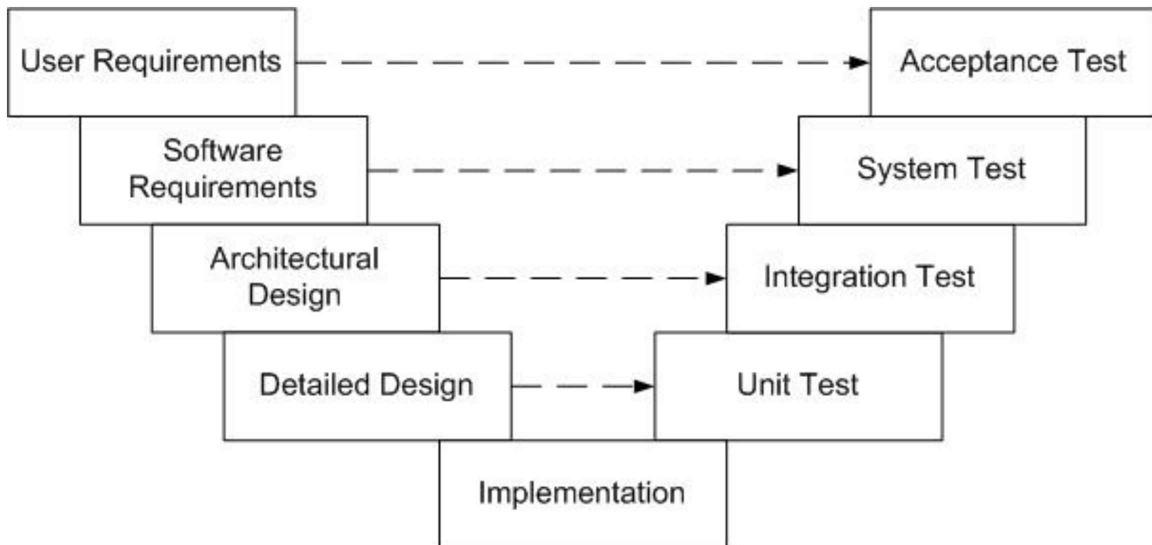

Figure 1: Relation between the construction and test phases.

There are direct relations between the test phases and the construction phases (Figure 1). The AT is based on the User Requirements Document and must be agreed on with the customer. Although the details of the tests can only be filled in later, the decision what to test and the method of testing should be decided during the UR phase. A flow scheme to test the requirements is provided on the course web site. This way, it is ensured that the requirements *can* be tested. It prevents one from writing down requirements such as "the user interface must be user-friendly". A report detailing the what and how to test (one or two sentences per requirement) must be delivered. After the User Interface is agreed on with the customer in the SR phase, the AT can be finished, because then the precise procedure for each test can be established.

The System Test is based on the software requirements. In general, this is an extended version of the AT and also stress tests (these test if the system can endure peak loads) are executed if relevant. Likewise, the IT is based on the architectural design and the UT on the detailed design.

It is advisable to explicitly include the tests in the planning: specification, implementation and execution. The planning then more clearly shows when the implementation should be finished, namely well ahead of the project deadline… From week 15,, no new features should be added; only the functionality realized up to then should be tested and debugged.

1.2 Management part of the process

Several management documents must be delivered. Most of these documents need to be updated as the project progresses. For example, the Software Project Management Plan (SPMP) initially contains the start and end dates for all phases, but only a detailed planning for the first two phases (the SR phase starts before the deadline of the document). Also included in the SPMP is the planning for the various testing activities. The SPMP also specifies how the project is organized: it specifies the roles that the team members have in the different phases. The SPMP is updated at the end of each phase with the detailed planning for the next phase. The other management documents set out rules

for various activities. The Software Quality Assurance Plan (SQAP) describes how software quality is ensured (e.g. describes coding and documentation standards) and the Software Configuration Management Plan (SCMP) describes how the various deliverables are managed. In particular, it details the structure of the project repository and version management.

Management documents should be kept as short as possible. Do not produce long documents with many rules that nobody adheres to. Instead, state only some basic rules that you think everybody can strictly follow.

The Project Manager (PM) is responsible for the SPMP. The PM is responsible for the identification of the tasks to be accomplished and the allocation of manpower to each task. This Therefore, the PM must know how much time each team member can spend and when the team members are available. Notice that the PM does not make all decisions on his/her own (primus inter pares)!

Team members should keep track of the time they spend on the project. The PM must collect these data and report these to Senior Management (SM): the teaching staff. It is important to administer time accurately as the data will be useful for future generations of students. Do not forget to include the hours spent during the regular classes! A spread sheet is available from the 2IM22/2IM24/2IM25 web site http://www.win.tue.nl/2IM22.

The Quality Manager (QM) is responsible for the SQAP. The QM must ensure that products adhere to the standards set out in the SQAP and therefore plays an important role in (organization of) reviews.

The Configuration Manager (CM) is responsible for the SCMP. The CM initially organizes the project repository (in Subversion) as described in the SCMP and ensures that the repository remains organized as described in the SCMP.

The following documents should be produced; the items listed are the main contents. Each of these documents has a main part and an appendix for each phase that gives specific details for that phase.

1.2.1 SPMP

The SPMP should contain:

- Project organization:
  Who is responsible for which task, in particular PM, QM, CM and the team leaders for the different phases. Make sure a backup is available for each role, to ensure continuity in case of, e.g., illness.
- Other organizational aspects:
  E.g., when meetings are planned.
- Effort estimation for each phase:
  Estimates can be obtained from the SPMPs of projects of previous years (follow the links to the projects from the main page of the web site http://www.win.tue.nl/2IM22) or from the SEP projects for Computer

Science students (http://wwwis.win.tue.nl/2IP35). This will at least give an estimate of the relative lengths of the phases. The particulars of your project may give rise to allot more time for a particular phase.
In the planning, it should be assumed that all requirements of high and medium priority will be implemented.

- Planning when various test activities are done:
  Plan, design, prerequisites and execution
- How time registration is set up
- Risk analysis and actions to be taken to reduce risks:
  For each risk, analyze its impact, the probability that it occurs, how it can be avoided and how the risk can be mitigated[1].

Notice that the total amount of time to be spent should equal the number of man-hours available.

For each phase (add before start of phase):

- A list of deliverables
- When deliverables are reviewed and the team members that do the review (notice that the SQAP describes which documents are reviewed and how the reviews are organized, so the SQAP is needed as input for this)
- Task assignment
- Effort estimation for each task and deliverable
  Estimates can be obtained from the SPMPs of previous 2IM22 or SEP projects or by using your own judgment.
- Risk analysis (see above)

Make sure to assign tasks to the proper phase, e.g., coding experiments should be counted as time spent for the Implementation phase. The time spent in the weekly instructions can be attributed to the current phase, or to, e.g., the phase that is explained during the instruction.

On a weekly basis, the following tables should be added to the SPMP:

1. The total number of hours spent per task and the number of hours required to complete the task
2. The total number of hours spent per phase and the number of hours required to complete the phase
3. Revised estimates of the time allotted for tasks and phases
4. For each team member the total number of hours spent and the number of hours to be spent per week for the rest of the project

These tables should be discussed with the entire team, so the project's status is known to all team members.

[1] A high risk with a high impact needs to be addressed immediately!

When more time than estimated was spent to complete a phase, less time is available for later phases and tasks (4. above). In this case, requirements of medium priority must be dropped in order to finish the project within budget. If the first phases take less time than estimated, some of the low priority requirements can be included for implementation. In either case, the SPMP should identify these requirements. Notice that a project can only be successfully concluded if all requirements are implemented or if the team members have all spent their available hours (approximately).

When significant deviations from the estimated effort occur, document what caused these deviations.

1.2.2 SQAP

The SQAP contains:

- Documentation standards (e.g., layout, cover, standard chapters)

  Each document should have a cover with at least the group logo, title, authors, project name, date and version number (see SCMP)
- Design standards (in SQAP/SR, /AD)
- Coding standards (in SQAP/DD), e.g. naming conventions
- Software quality assurance metrics, e.g. maximum length of procedures, maximum number of parameters of procedures and percentage of comments in the code (in SQAP/DD)
- How is it verified that products adhere to these standards
- Organization of reviews: who, what, how

  Usually, at least one of the authors, an independent team member and the SQAM should take part in a review. At least all product documents, the SCMP, SQAP and SPMP should be reviewed. Artifacts such as minutes do not need to be reviewed. The artifact to be reviewed can be read beforehand or during the review. It is convenient to enter defects into a template for reviews, together with an indication of the severity, if and how the defect will be solved and by whom
- Optionally, a problem reporting procedure

1.2.3 SCMP

The SCMP describes:

- Structure of document storage including that for source code, minutes, problem reports and document templates
- Identification of documents (e.g. version numbers)

  A common format for version numbers is x.y.z, where x and y start at 0. z starts at 1 and is increased with each minor change. y is increased after it is approved in an internal review, after which z is set to 0. and x is increased only when the document is approved by the customer (e.g., in case of the requirements document) or when the document is approved by SM (e.g., the SQAP), after which y and z are set to 0. When defects are discovered and the

document is changed, z is increased. This way, it is easy to see the status of a document, e.g., a document x.0.0 is a document that has not been changed since the customer or SM has approved it.

This document should contain a 1 to 3 page description of how you plan to structure your repository in Subversion. In particular, it should also define how the program code is managed to allow several people to work on the same module and to define where the most recent versions can be found.

1.3 Example

As a running example, we use a very simple (database) application that contains students, courses and the participation of students in courses. Students can be added and removed and changes can be made to the student data. Courses can be added, removed and listed. Students can register for courses and a grade can be set for courses that a student has completed. The example is only partially elaborated, in particular for the architecture and implementation phases.

The example includes

- a short initial informal description from the customer
- the user requirements derived from it
- a first version of the logical model
- the logical model
- parts of the prototype
- the architectural model
- a high-level view of the implementation
- a detailed view of the implementation

2 User Requirements phase

During the User requirements phase, the requirements are elicited from the customer. One of the main problems in this phase is that the views of the customer and the development team are usually different. The UR and SR phases are meant to let these views on the system converge as much as possible. This is accomplished by, e.g., frequent meetings with the customer and construction of prototypes. The User Requirements Document (URD) serves as a contract between the team and the customer. For information about requirements and the requirements process, see [3], [8] and [11].

The URD should start with a general description of the product's functionality, the environment in which it is used and a characterization of its users. Then the individual requirements should be listed, grouped by functionality, and a number of scenarios (i.e. examples of typical uses of the system) should be given. If appropriate, also include a glossary of domain-specific terms. You may use a spreadsheet for the functional and extra-functional requirements.

Requirements should be stated unambiguously in plain short sentences. The quality of each requirement can be analyzed with the aid of the flow scheme of Figure 2, which is similar to the procedure described in [3]. Notice that it is also checked if requirements can be tested. The results of this last check should be recorded in the User Requirements Analysis report (URAR) as these can be used later in the process. Notice that only the testing method is identified, *not* the detailed procedure. In the running example, there is a requirement: "The system provides functionality to add a student". The URAR should contain at least one test case for adding a student. The method of testing specified in the URAR could be "inspection of the database after closing the application" to see if the student has actually been added. Notice that there are tools to assess the quality of requirements.

Requirements should be accompanied by a priority. The highest priority requirements must be implemented in the final product, the lowest priority requirements will only be present if time allows this. The priorities indicate the order in which functionality is implemented. Obviously, the lower priority requirements must be taken into account in the design, so that the design does not *prevent or hamper* implementation of these requirements.

In addition to the functional and so-called extra-functional requirements, the URD should describe a number of scenarios that illustrate typical uses of the system. These scenarios may also be useful for the ATP.

During this phase, the project plan and the other management documents are written, globally for the entire project and in detail for the UR and SR phases.

Already in this phase, experiments must be conducted in areas where the team's expertise is lacking (high risks). For example, if a web application should be delivered, it can be investigated which scripting languages are available and which seem to be most suitable.

Likewise, the customer may prescribe a certain programming language that the team is not familiar (enough) with. In this case, team members can start experimenting to accomplish tasks that will most likely be needed in the final implementation. This results in a number of "prototypes" that will speed up the final implementation. These activities are not restricted to the UR phase, but continue during the entire project.

Deliverables for this phase are the URD, URAR, SPMP, SCMP and SQAP.

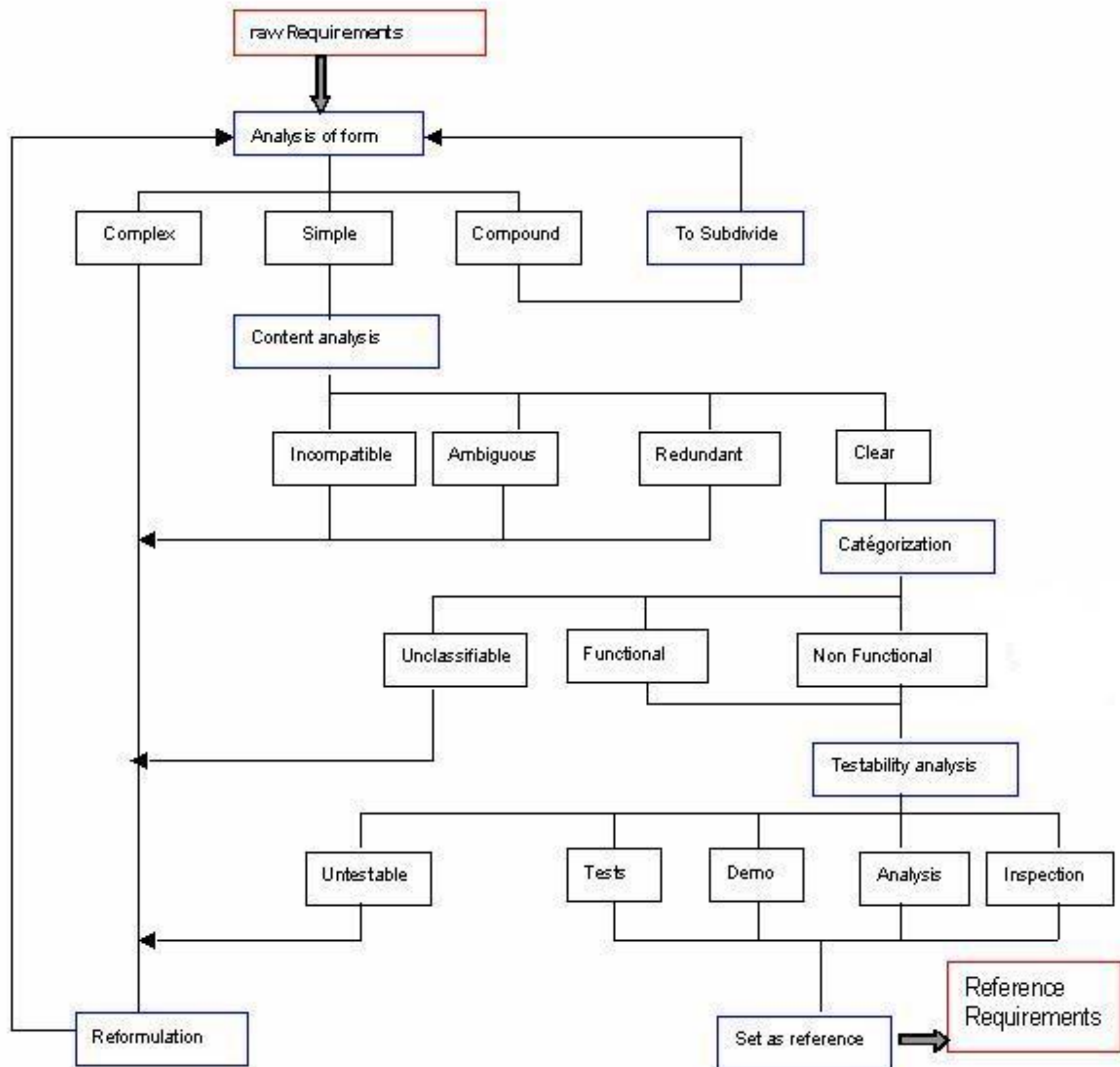

Figure 2: Requirements analysis

2.1 Example: User requirements

Initial informal description of the system by the customer

.

The system is used to administer students, courses and the course participation of students. The system must be implemented in Java and use a relational database to store the data. The system must have a graphical user interface.

User Requirements

Functional Requirements (priorities not shown)[2]

Functionality concerning students

The user can add and remove students and change the data recorded for each student.

UR-1 For each student, the system stores[3]
  a. a unique student identifier
  b. last name
  c. initials
  d. date of birth

UR-2 The system provides functionality to list the students

UR-3 The system provides functionality to select a student

UR-4 The system provides functionality to remove a selected student

UR-5 The system provides functionality to change the data of a selected student

UR-6 The system provides functionality to add a student

Functionality concerning courses

The user can add and remove courses and change the data recorded for each course.

UR-7 For each course, the system stores
  a. a unique course identifier
  b. course name

UR-8 The system provides functionality to list the courses

UR-9 The system provides functionality to select a course

UR-10 The system provides functionality to remove a selected course

UR-11 The system provides functionality to change the data of a selected course

UR-12 The system provides functionality to add a course

Functionality concerning course participation

UR-13 For each course participation, the system stores

---

[2] Many requirements documents use the style “The system shall…”. As this sounds rather archaic, present tense is used here. This is actually equally commanding.

[3] Notice that this is actually shorthand for 4 different requirements, which should be individually traced as UR-1a, UR-1b, etc.

a. a course
b. a student
c. a grade

UR-14 The system provides functionality to list the course participations
UR-15 The system provides functionality to select a course participation
UR-16 The system provides functionality to remove a selected course participation
UR-17 The system provides functionality to add a course participation
UR-18 When adding a course participation, the user can select the course from a list provided by the system
UR-19 When adding a course participation, the user can select the student from a list provided by the system

Extra-functional requirements

UR-20 The system is implemented in Java
UR-21 The data is stored in a relational database
UR-22 The system has a graphical user interface

3 Software Requirements Phase

During the SR phase, a model is made to describe the problem. This description should not take any implementation details into account. The reason for this phase is for the team to get an understanding of the problem and the entities that play a role in it. The model should describe the system as seen "from the outside". With the model, it must be possible to simulate its behavior. The model is given as a UML [6], [9], [10], [12] class diagram, use case diagrams and sequence diagrams. The model describes *what* the product should do, not *how*.

A good starting point for the model is the requirements document. A noun that occurs in the requirements often indicates that a particular class is needed in the model. Likewise, an adjective may indicate that a particular attribute is needed and a verb can indicate the need for a method. If no attributes and methods are found with a candidate for a class, it is likely that the class is irrelevant to the model. This leads to an initial logical model (see section 3.1). From the scenarios in the URD, a number of typical use cases can be derived. Each use case is a "driver" for a sequence diagram. With the classes, methods and attributes in the initial logical model, try to represent a typical use case with a Sequence Diagram (SD). This usually leads to the conclusion that additional classes, methods and attributes are needed in the class diagram. Add these to the class diagram and repeat the process until you have an extended logical model with which you can trace an input (a method call from the "outside" on one of its classes) through the model until the expected output is produced. It is very well possible that missing or incomplete User Requirements are discovered in the process. This is another reason to go through the SR phase.

Appendix A: The Coffee Machine shows the end product of this process. In this appendix, it is shown how the classes and methods are derived from a short description of the functionality and an SD shows how user actions result in a cup of coffee. Notice that the coffee machine might be a modern coffee machine as they can be found throughout the TU/e or it might be a box with a goblin inside. Although this is stretching it a bit, it might describe the process of ordering a cup of coffee in a restaurant and the actions of a waiter.

The Software Requirements Document (SRD) should contain a general description, the initial logical and extended logical models (see the next sections), use cases and sequence diagrams, along with some clarifying text. In addition, each class, method and attribute in the *extended* logical model should be documented. For a class, explain its function and possible invariant properties. For a method, describe its function and pre- and post-conditions. For an attribute, describe its purpose. Each of these S*oftware Requirements* should be numbered and it should be documented from which User Requirement it is derived. Never renumber your User Requirements after this in order to avoid rework of the SRD! This tracing is also done from User Requirements (URs) to Software Requirements (SRs), to verify that all URs are covered by SRs

The SRD contains a *problem description,* so do not use implementation terminology!

In addition to the model, a prototype should be delivered that demonstrates the Graphical User Interface (GUI). This can be done with a number of screenshots and some description that demonstrates the use cases. Another possibility is a mock-up with which one can navigate between the different screens that contain all the controls of the final product but without any further functionality behind it. Notice that the GUI prototype is a contract between the team and the customer once the SRD has been accepted! Because the GUI is agreed upon during the SR phase, the ATP can now be completed: the test cases can now be elaborated to contain the precise sequence of actions needed to perform the test and the test procedures can be added (see section 7.4)

Deliverables for this phase are the SRD, the STP, the completed ATP, and the additions to the management documents. Since the ATP is only needed during the last weeks, the hard deadline is set to week 14.

3.1 Example: Initial logical model

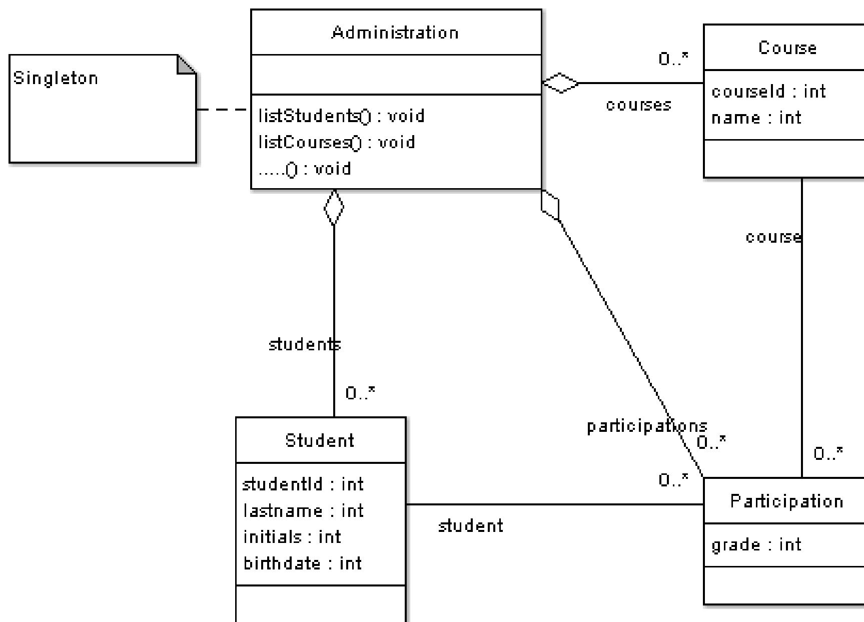

Figure 3: Initial version of the logical model[4]

3.2 Example: Extended logical model

When elaborating the initial version of the logical model, it became apparent that the Administration class would become too large. This class is the logical choice to 'own' the Students, Courses and Participation sets. Since there would be add, change, remove, etc.

[4] Notice that only part of the operations and attributes have been included in the figure. Setters and getters are never included. Attributes that are graphically represented (on the edges) are not shown in the boxes.

methods for each of these, it was decided to introduce Administration classes for each of these sets.

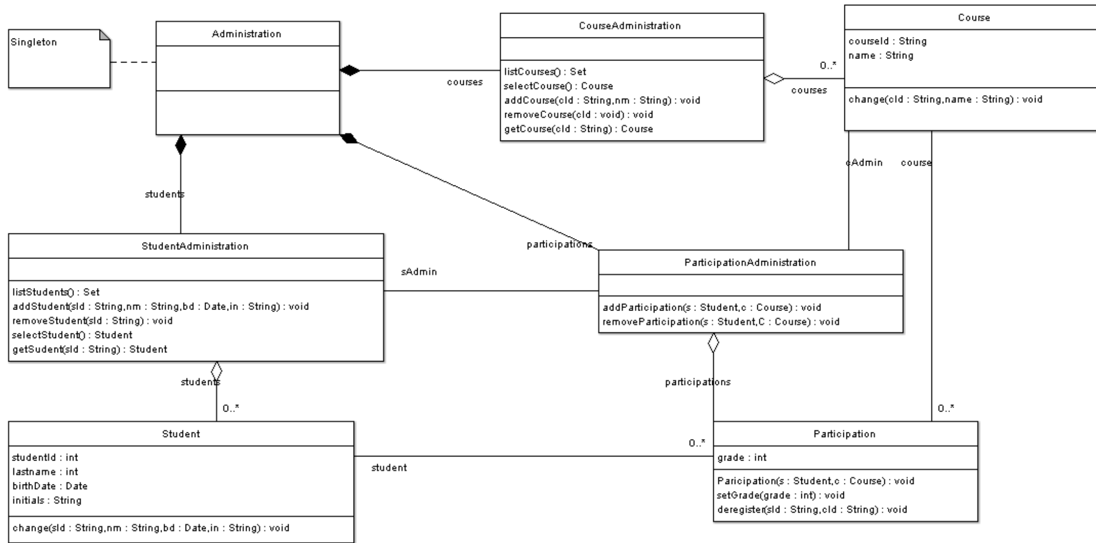

Figure 4: Extended logical model.

3.3 Example: Sequence diagram

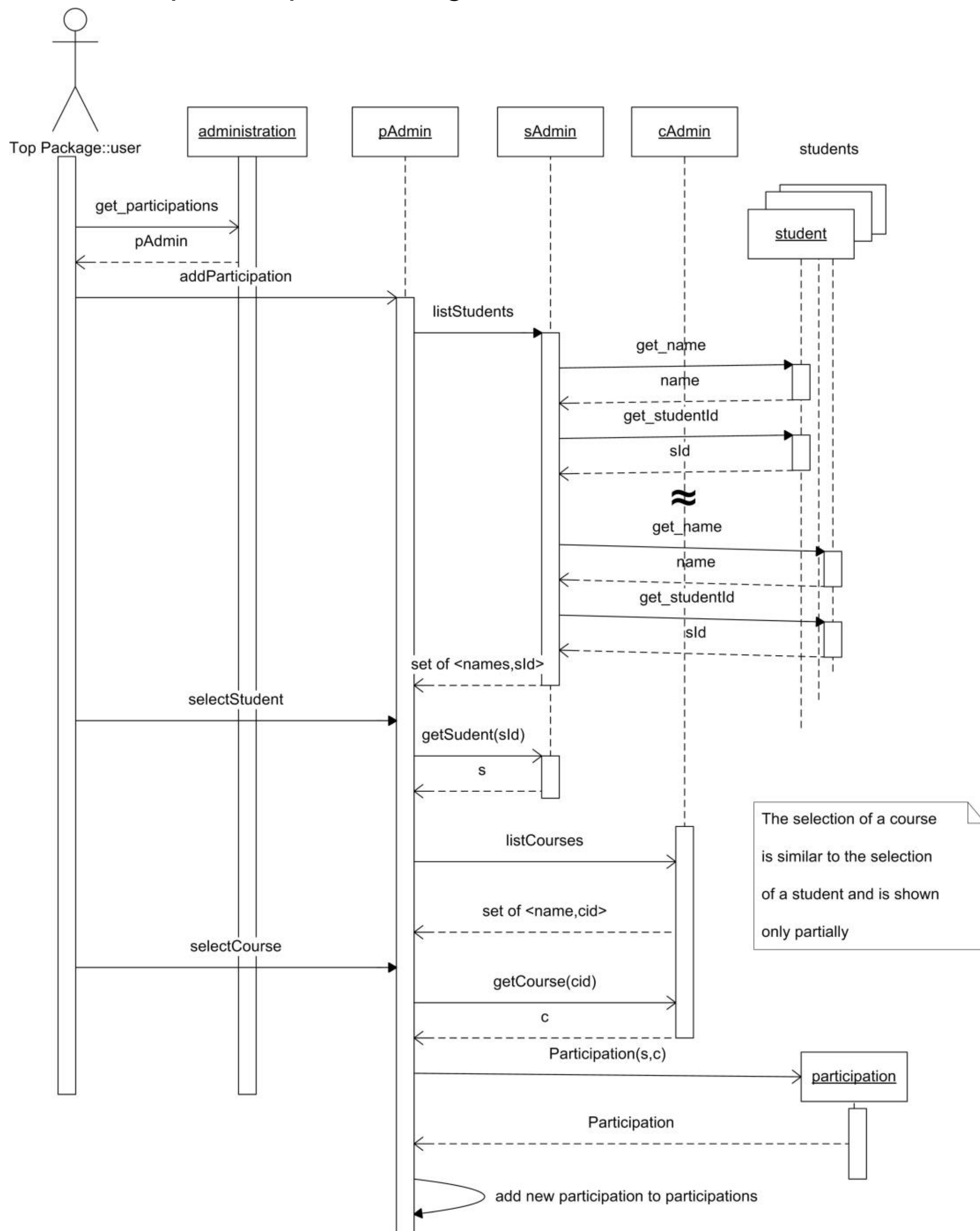

Figure 5: Sequence diagram.

3.4 Example: Screenshots

Figure 6 and Figure 7 show screenshots of the prototype: the main menu and the student table.

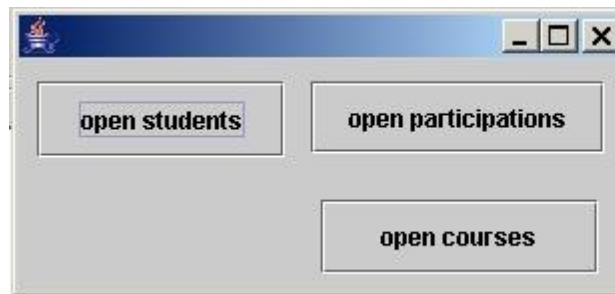

Figure 6: The main menu.

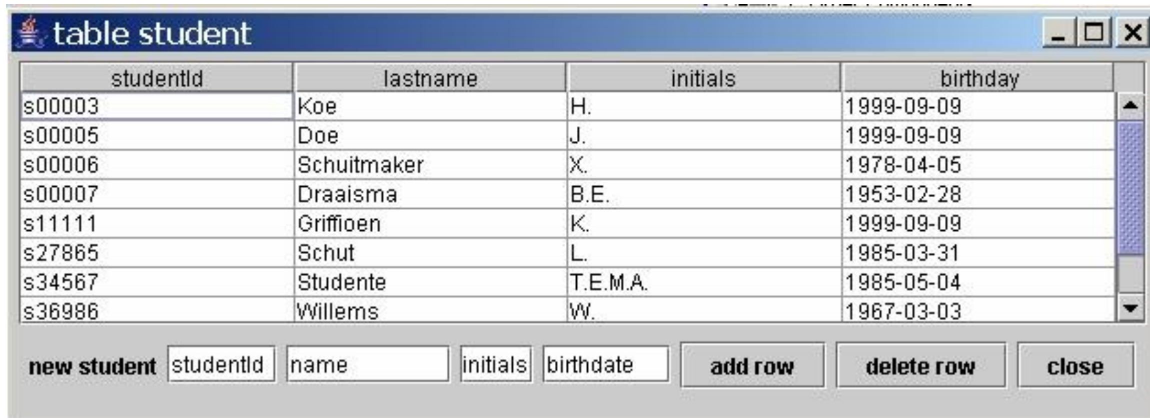

Figure 7: The student screen.

4 Architectural Design Phase

The goal of the Architectural Design (AD) phase is to divide the system into a number of coherent pieces (packages) that can be developed in isolation by different groups. This division also reduces the complexity, because each package has a lower complexity than the entire application.

The AD phase is the first phase that concerns the construction of the software. The extended logical model is a good starting point for an architectural model, but it needs to be extended with some implementation details. The GUI must be added to the model as it is not modeled in the SR phase. In an application that uses a database, entities in the database will have a representation in both the database model and the software model. Likewise, some of these entities will also have a representation in the GUI. Normally, classes from the logical model will also need to be extended with extra attributes.

Packages should be chosen such that a package implements, e.g., a logically coherent set of functions or services. There should be fewer class associations between classes in different packages than between classes in the same package. In order to work independently on different packages, it is necessary to define the interfaces between the packages. These interfaces consist of all the methods that can be called from “outside” the package including their signature, i.e., their name, parameters, and return types.

The AD Document (ADD) should contain the model and describe the interfaces between the packages. Document these *Architectural Requirements* in the same way as the Software Requirements. Document the function of each package as well. Trace the Architectural Requirements to the Software Requirements and v.v. The functionality of each package is described by a logical model as in the SRD, i.e., for each package there should be class diagram of the classes in that package. Often, parts of the extended logical model can be re-used here. Notice that sequence diagrams can help you in the design process!

After the architecture has been designed, it is also possible to determine how to assemble and test the system. After the packages are implemented and tested in isolation, one can start testing packages together. Packages are assembled starting with a package that does not depend on other packages. The first package to add to this is a package that only depends on the first. In this way, one continues until the entire system is assembled. The advantage of this approach is that only test drivers[5] are needed, and no test stubs[6]. The Integration Test Plan (ITP) describes the order in which the system is assembled, which extra facilities are needed, and which tests will be done to verify the correct functioning of the system.

The architecture diagram for the running example is shown in Figure 8. The system is split into three packages: the GUI, the database and a package that serves as an

[5] Drivers are classes from which the methods of the newly added package are called.
[6] Stubs replace called classes by dummy implementations.

abstraction layer for the database. In this package, the data from the database are converted to a format suitable to store in Java arrays. Only the functionality for the student table is elaborated. Each package still contains only logical models, so most implementation details have not been added yet.

Notice that the interface between the DBAbstractionLayer package and the GUI already requires knowledge of implementation details: remove and change must have the row number as parameter (not shown below). This can be avoided by passing the entire record in both cases. However, an additional internal action is then required in the DBAbstractionLayer package to locate the record in the database, which results in a locate(record) method in this package. This introduces unnecessary overhead in the application as the record number is returned by the TableModel (a standard Java class) that implements the StudentAdministration in the GUI. This is a typical architectural issue: some generality is sacrificed for performance.

It is essential that prototypes are built to experiment with implementation details. In this case this would mean to find out how TableModels work, how to query the database and how to build the GUI.

Deliverables of this phase: ADD, ITP and additions to the management documents.

4.1 Example: Architectural model

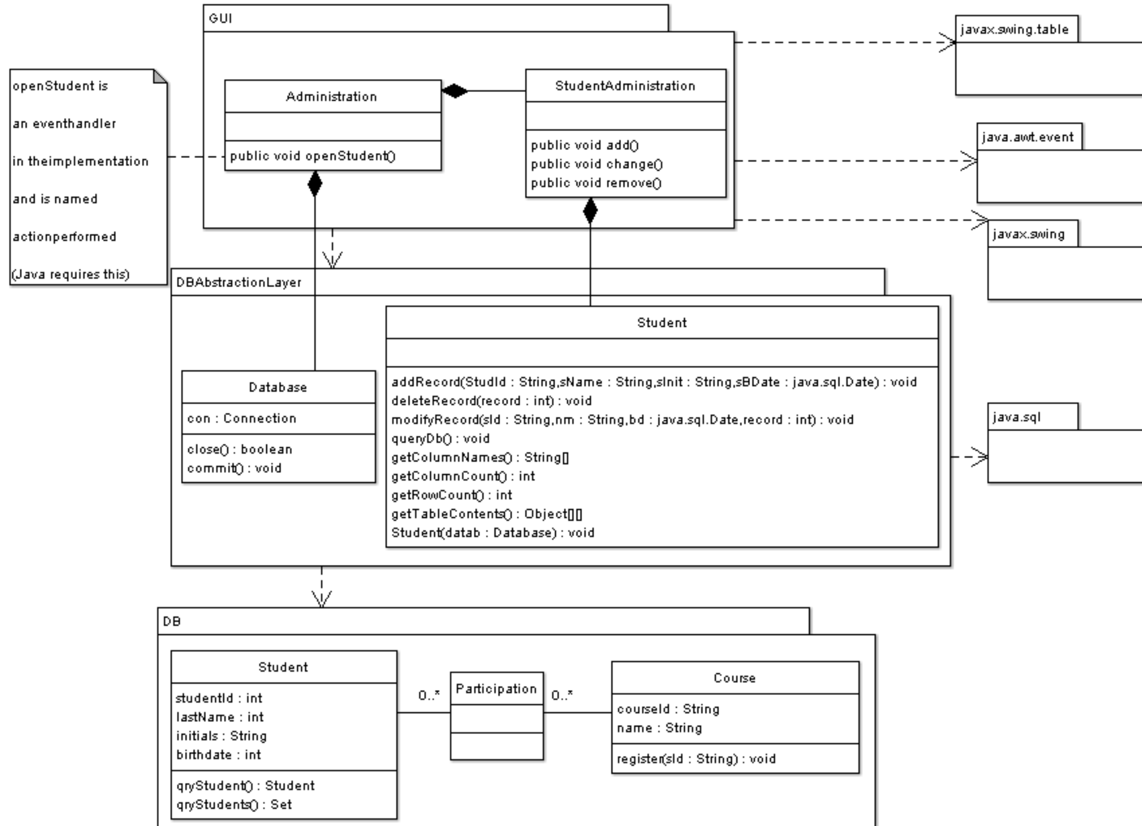

Figure 8: Architectural model (only Students shown).

The interface of the DBAbstractionLayer package (for the GUI package) consists of the methods of the classes Student and Database and the attribute of Database (setters and getters of the attribute are implicitly given). The interface of the DB package consists of the table structure and the queries. Notice that only a part of the Java packages is shown (general utility packages are left out).

5 Detailed Design Phase

In the Detailed Design phase, the architectural design is refined to the point where it can be directly implemented (coded). However, as the participants in this course are not, in general, experienced programmers, a detailed design would probably only be a waste of time. Therefore, the detailed design and implementation phase coincide for 2IM24/2IM25. However, the Detailed Design Document (DDD) should record some implementation considerations (design decisions) in addition to the documentation of the classes in the implementation.

6 Implementation

In this phase, different groups can implement the different packages. Important issues in this phase are version management and to make sure that everyone has access to the latest versions of packages. Make sure to divide the work properly, so that it is unlikely that two people are working on the same file at the same time. Therefore, it is imperative to use Subversion (SVN) and TortoiseSVN to manage your code. A repository will be created for each group on the department's SVN server.

Document the final implementation in the Detailed Design Document (DDD): packages, classes, methods and attributes. For each of these, document in the header resp. declaration, its purpose and the Software Requirement(s) from which it originates. For a class, document the class invariants, if applicable. For a function, document parameters, return type and pre- and post-conditions. It is strongly recommended to use a documentation generator for this purpose. Use either a documentation generator built into your development environment (e.g., JavaDoc in Netbeans), or use a general purpose documentation generator, e.g., [7]. To use a documentation generator, the information for the DDD is entered as comments in a special format in the code. Because this information is part of the code, it is easy to keep the documentation in line with the code.

Stop adding and finishing new functionality by week14, after that, only test the code and just *repair* defects found.

Deliverables: DDD, Software User Manual (SUM), code and updates of the management documents. The SUM describes installation and use of your product.

6.1 Example: High-level view

Figure 9 shows a high-level view of the implementation (excluding the database).

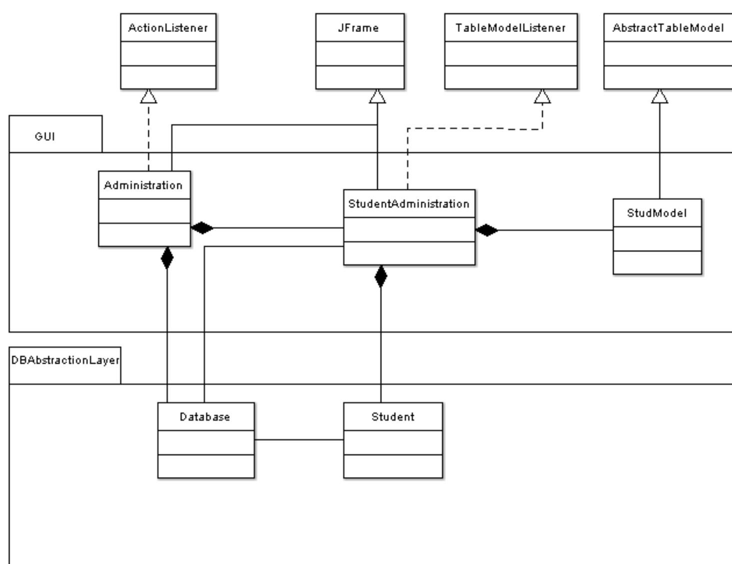

Figure 9: High level view of the implementation.

6.2 Example: Detailed view

Figure 10 shows the classes as reverse-engineered from the implementation. Some other (nameless) classes were also found. These were MouseAdapter and Runnable classes from the Java packages. Notice the differences as compared to the Architectural Model.

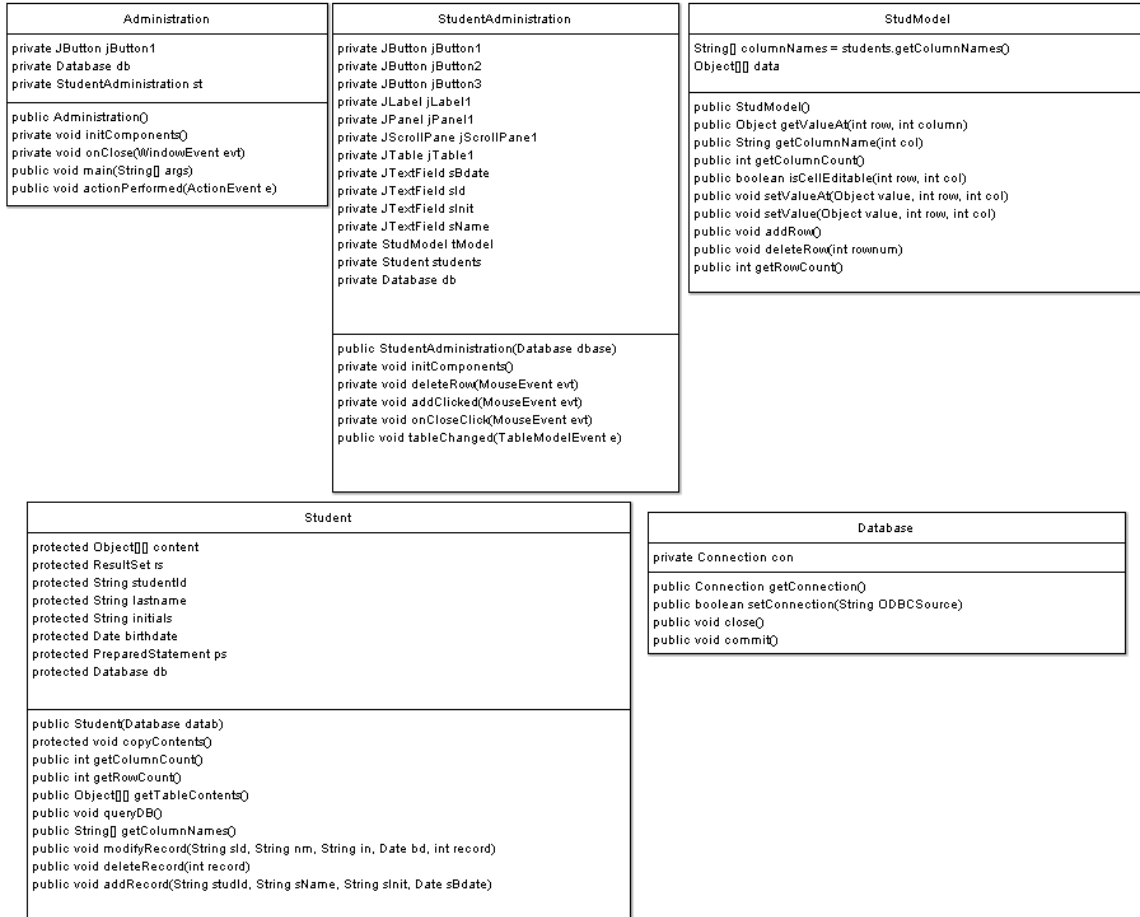

Figure 10: Implementation details.

7 Test Phases

Each test involves specification (what will be tested), design (how will the test be done, inputs, expected outputs) and execution. The AT additionally requires some preparation because it is executed in the customer's (in this case the staff's) environment.

7.1 Unit Tests

Due to time constraints, it may not be feasible to do unit tests for all classes and packages. However, some classes have to be tested with the unit test facilities provided by your development environment, as explained in the lectures by Dr. Verhoeff.

Do not wait with testing until you have a complete first version: implement part of the functionality and test this part before continuing! Obviously, during the testing you'll have to go back to the implementation phase very often. In principle, one should the repeat all previous testing, which might take too much time for 2IM24/2IM25. However, with the test facilities of your development environment, it is easy to automatically repeat all previous tests.

Deliverables: updates of management documents, a Unit Test Plan (UTP) and Unit Test Reports (UTRs). As the number of hours available per student were lower in the previous years, the UTP and UTRs were not required in previous years, so no examples are available. Guidelines for the contents of these documents will be provided.

7.2 Integration Tests

In this phase, the system is assembled according to the ITP, typically in the direction opposite to the dependencies.

Deliverables: update of management documents.

7.3 System Test

In this phase, the system is exhaustively tested to make sure the system will not fail the AT. The ST should be a superset of the AT. Include test cases that test extreme circumstances. For example, if you have an array A that can contain up to 10 items, make sure you have a test case that checks the case when it contains 10 items and check what happens if you try to store 11 items in the array if there is any chance that this might occur.

Deliverables: none

7.4 Acceptance Test

The ATP consists of test *procedures* and is constructed during the SR phase. A test procedure defines what the inputs are, which test *cases* are executed and what the expected outputs are. The scenarios from the URD may provide some clues for the construction of the test procedures. A test procedure groups different test cases to reduce

the amount of work involved (i.e. preparing the initial state for the test case). The URAR is a good starting point for the ATP test cases. In the running example, a test case that verifies that it is not possible to create duplicate student identifiers can be combined with, e.g., test cases that verify if students can be added and that the system can provide a list of students.

Prepare the AT well. For example, if a prerequisite of the AT is that a certain application is installed on the customer's PC, make sure the customer knows this in advance. If, for some reason, the customer can not or would not like to install this application, a solution to this problem should be found. These kinds of issues should not be resolved shortly before the AT, because an alternative solution may require some time to arrange. Making sure that all arrangements for the AT are handled timely is a task of the PM and should be included in the planning.

Provide the customer with

- A CD with your installation files and input files for the test
- A copy of the ATP
- A copy of the SUM

The customer must do the acceptance test on his/her own without aid from the team. The group must record which tests pass or fail and the result should be recorded in a test report. If all tests pass and all the highest priority requirements are tested, the customer may accept the product.

Deliverables: Acceptance Test Report (ATR), presentation (see chapter 8).

8 Concluding remarks

In this document, a software development process for 2IM24/2IM25 has been described. The process covers both the technical part (i.e. the phases UR, SR, AD, DD, implementation, UT, IT, ST, and AT) as well as the managerial part, and includes various examples. Deliverables are described for each of the phases of the technical part. Moreover, it is described which management documents must be created or updated during each phase. This document briefly describes what the management documents should contain. An overview of the phases and deliverables can be found in Appendix B: Overview of phases and deliverables. Appendix B also contains a table that gives hard and soft deadlines for deliverables and planned activities during class hours.

The Project is concluded with a presentation for all groups and the teaching staff. The presentation should cover both the product (including a demonstration) and the process. Include some statistics; include at least the number of lines of (real) code and hours spent (per phase and in total). Recommendations for improvements are appreciated. Also cover topics such as what you have learnt, was it enjoyable and did you get insight into the software engineering process.

References

The ESA standard [5] is available from:
http://wwwis.win.tue.nl/2R690/doc/ECSS-E-ST-40C(6March2009).pdf.
Note that [4] is based on the ESA standard.

[1] I. Sommerville, *Software Engineering,* Addison-Wesley, ISBN 0-321-31379-8, 8th edition, 2007.

[2] T.C. Lethbridge and R. Laganière, *Object-Oriented Software Engineering – Practical Software Development using UML and Java*, McGrawHill, ISBN 0-07-70109082, 2nd edition, 2005.

[3] Suzanne Robertson and James Robertson, *Mastering the requirements process*, Addison-Wesley, 1999.

[4] C.Mazza, J.Fairclough, B.Melton, D. de Pablo, A.Scheffer, R.Stevens, M.Jones, G.Alvesi, *Software Engineering Guides*, Prentice Hall, ISBN 0-13-449281-1, 1996.

[5] ESA European Cooperation for Space Standardization, ECSS-E-ST-40C, European Space Agency, 6 March 2009.

[6] Formal descriptions available from http://www.omg.org/spec/UML/

[7] Robodoc: http://www.xs4all.nl/~rfsber/Robo/robodoc.html;
NaturalDocs: http://www.naturaldocs.org/

[8] M. Mannion and B. Keepence, *SMART Requirements,* Software Engineering Notes Vol. 20, 2, pp. 42-47, 1995, ACM Press

[9] M. Fowler, *UML Distilled,* Addison Wesley, ISBN 0-321-19368-7, 3rd edition, 2003

[10] J. Arlow, I. Neustadt, *UML 2 and the Unified Process*, Addison-Wesley, ISBN: 0201770601, 2002

[11] K. Wiegers, *Software Requirements*, Microsoft Press, ISBN 0-7356-1879-8, 2nd Edition, 2003

[12] UML Basics, http://docs.kde.org/, type in search box: UML

Appendix A: The Coffee Machine

The following text is a very short description of a coffee machine:

The coffee machine *delivers* coffee in different strengths and with different amounts of sugar and milk, as *select*ed by the customer. The coffee machine *present*s menus that allow making the different choices. The product is *serve*d in a cup.

This example model is not meant to be complete, e.g., payment for the product is not modeled and no facilities are present in the model to signal if products or cups have run out. It is merely tried to illustrate how class diagrams and sequence diagrams can be used together to develop a useful analysis model.

Notes on modeling

The underlined words give a good clue about classes, the italic ones about methods and highlighted words about attributes.

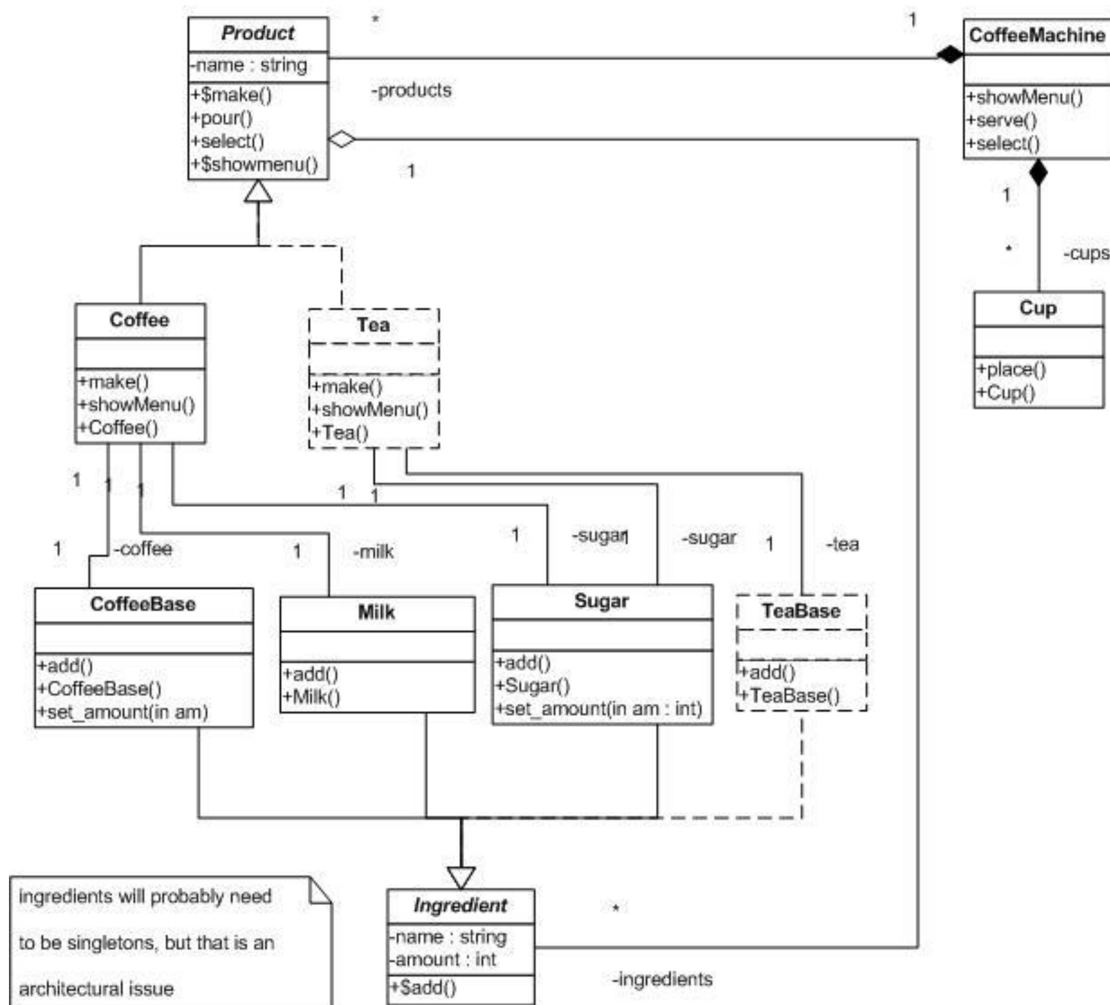

Figure 11: Class diagram.

Some aspects were not modeled in the class diagram shown in Figure 11 either for simplicity (menu) or in order not to implicitly make design decisions, e.g. whether the coffee comes in big thermos flasks or as beans that need grinding, addition of water and filtering. As there is no such detail in the spec, we need to ask the customer. So modeling the problem induces questions about details that need answers from the client.

The cup, in principle, places itself. Notice that the reference to the cups should not be named "cupholder" – this is too much directed to implementation. For most coffee machines, this will be the reality, but it is not justified by this requirements specification.

Tea is added only to show that the model is easily extended with other products. To be able to easily extend the model for other products (tea, milk, soup, cocoa) the class Product has been introduced. To generate a menu, the names of all products in the products aggregation can easily be collected without knowing about individual products.

The ingredients class was introduced to be able to treat all ingredients in a uniform way. The *add* method is abstract in the Ingredients class and must be implemented by each individual ingredient, which makes sense as adding, e.g. coffee from a reservoir is quite different from adding sugar.

Each product knows what its ingredients are, so the knowledge about the preparation of the product is entirely with the individual products.

In an actual implementation, the cups aggregation would probably be replaced by a class to control a cup holder, which actually might involve quite a bit of logic, e.g., when users may place their own cups. Likewise, the Coffee class is most likely replaced by the control software for an instant coffee container.

Use dynamics (sequence diagrams, state charts) from the start. In this case, start e.g. with the classes coffee machine, coffee, sugar, milk and menu, place the attributes and methods that you identify. Then, try to make a Sequence Diagram for this model; see Figure 12 for an example. From the SDs you will find missing methods, classes and attributes. Then go back to the CD, add these things and back to SD. This is the fastest way to come up with a model: **do not focus too long on the static aspects only**, because once you get to the dynamic aspects you have to change too much! In the static diagrams, consider likely extensions so your model is extendible, without compromising its simplicity. In this case, each product controls its own recipe and the abstract product and ingredient classes make the model flexible: as long as it can be poured into a cup any product can still be incorporated.

Note that Figure 12 starts with the initialization (i.e. creation) of the system, and a user subsequently selects coffee with sugar, the strength of the coffee, the amount of sugar, and the actual service (i.e. "pour").

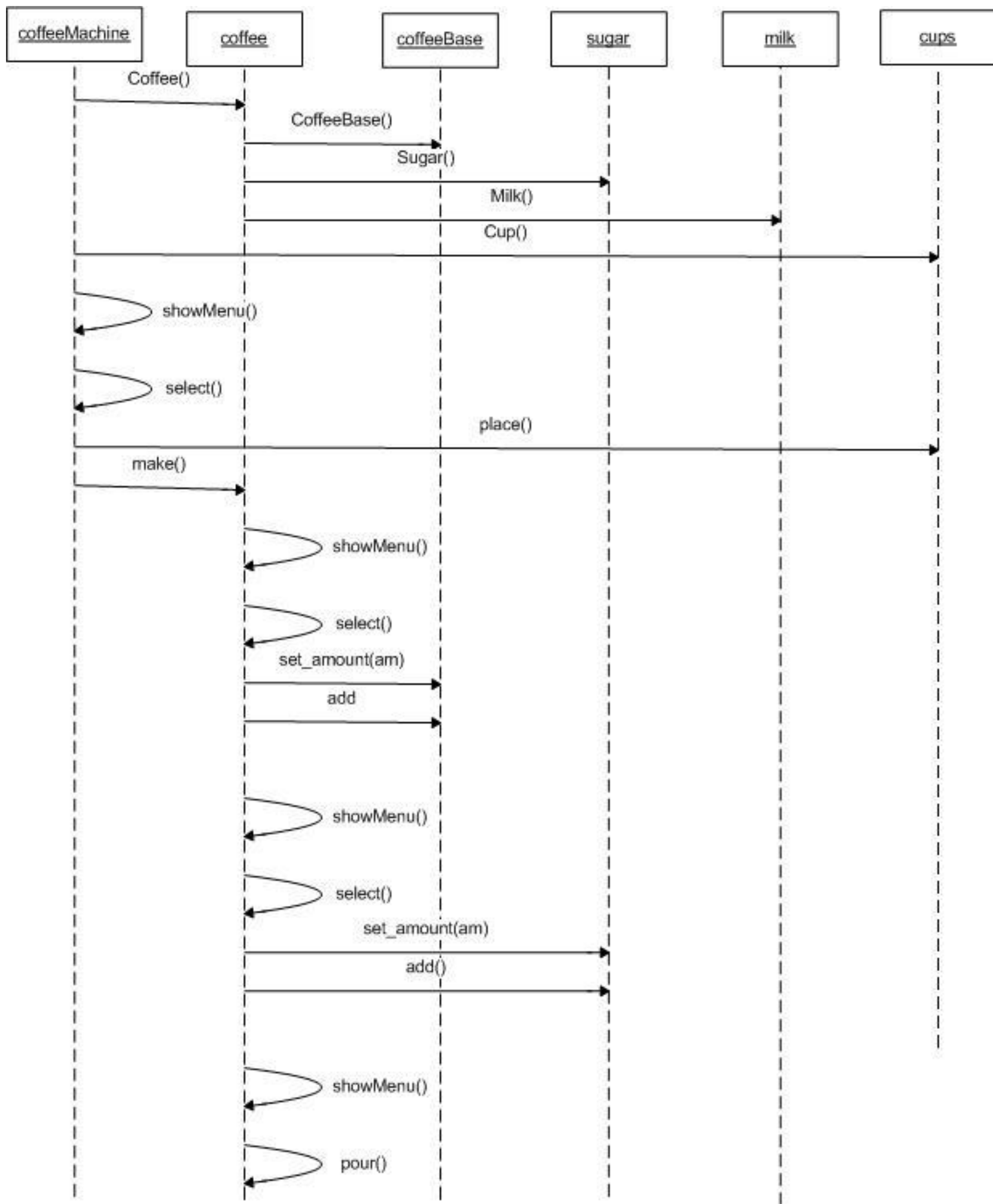

Figure 12: Sequence diagram.

Appendix B: Overview of phases and deliverables

An overview of the phases of the described software development process for 2IM24/2IM25 is given in Table 1. This table also indicates which deliverables are not required for 2IM24/2IM25. The postfix "*" in the column *Deliverables* denotes that an update of the deliverable is required (preparing for the *next* phase).

Phase	Deliverables	Not required for 2IM24/2IM25
User Requirements (UR)	SPMP, SCMP, SQAP, URD, URAR	
Software Requirements (SR)	SRD, STP, ATP SPMP*, SCMP*, SQAP*	
Architectural Design (AD)	ADD, ITP, SPMP*, SCMP*, SQAP*	
Detailed Design (DD)	DDD, UTP, SPMP*, SCMP*, SQAP*	SPMP*, SCMP*, SQAP*
Implementation	code, SUM, DDD, SPMP*, SCMP*, SQAP*	
Unit Test (UT)	UTR, SPMP*, SCMP*, SQAP*	
Integration Test (IT)	ITR, SPMP*, SCMP*, SQAP*	ITR
System Test (ST)	STR, SPMP*, SCMP*, SQAP*	STR
Acceptance Test (AT)	presentation, ATR	

Table 1: Overview of phases and deliverables.

Table 2 gives the scheduled activities and the deadlines for the project. Time during classes is spent on explanations about the process and the different phases, on answering any questions that may arise and on discussing deliverables with individual teams. The teams can request presentations of issues of general concern that need clarification.

When a hard deadline is indicated in the schedule, the final version of the deliverable must be handed over during class hours. This version will be used to grade the performance of the team. Obviously, if there are legitimate reasons for a delay, an extension of a hard deadline can be negotiated. Soft deadlines are not explicitly marked as such. They indicate the state in which particular documents should be at that point in time. If this is not the case, the team should put in extra hours to catch up. This is also the case when an extension of a hard deadline is granted.

The phase extensions of the management documents can be delivered to SM at the end of the project. Of course these extensions must be ready before the start of the relevant phase and the extensions can be discussed during class hours.

week	Class hours	Activities non-class hours following week
1 4/9	Introduction Assignments and process	During the entire project: research and prototyping activities. Organize project Management documents (/UR)
2 11/9	First hour: Introduction Software Engineering	URD, URAR Management documents
3 18/9	First hour: Introduction SE Discuss URD, URAR and management documents	URD, URAR Management documents Discuss requirements and URD with customer
4 25/9	First hour: Introduction SE Introduction UML Discuss documents	URD, URAR, management documents (/SR) Discuss URD with customer and get approval
5 2/10	**Deadline** Approved URD, URAR and management documents Modeling exercise	SRD (models, prototype)
6 9/10	Discuss models, prototype	First version SRD(models, prototype, SRs, UCs)
7 16/10	Discuss first version SRD	Second version SRD
8 23/10	Discuss second version SRD Discuss AD phase	Final version SRD
9 12/11	**Deadline** SRD, updates management documents (/AD)	ADD (models) Updates management documents (/DD)
10 19/11	**Deadline** updates management documents (/DD) Discuss AD models	First version ADD (models, requirements) Coding
11 26/11	Discuss first version ADD	Final version ADD Coding
12 3/12	**Deadline** ADD	Coding/ Unit testing User Manual, ATP,UTP,UTR
13 10/12	Discuss ATP, UTP, UTR, User Manual	Coding User Manual, ATP,UTP,UTR
14 17/12	**Deadline** approved ATP, User Manual Discuss UTP, UTR	Coding User Manual, ATP, Integration & System Testing, Debugging 1st AT
15 7/1	**Deadline** 1st AT, UTP Test reports	Coding, testing, debugging, finish documentation 2nd AT, UTR
16 14/1	**Deadline** 2nd AT, UTR Test reports (ATR, UTR) Final presentation Delivery project/product documentation	Possibly: final presentations + delivery documentation at a later date, to be discussed

Table 2: Schedule and deadlines