Object-Oriented Analysis and Design Through Scenario Role-Play

Jürgen Börstler

UMINF 04.04
ISSN-0348-0542

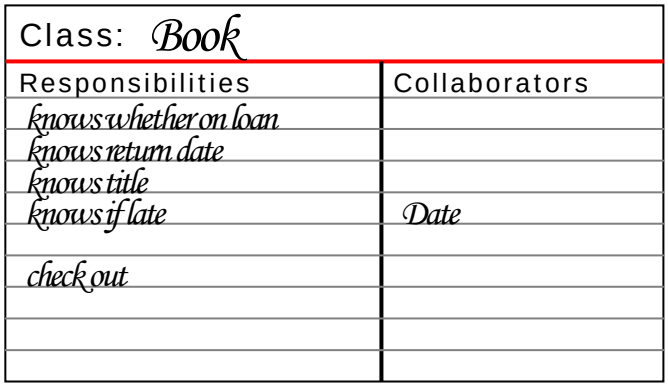
UMEÅ UNIVERSITY
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

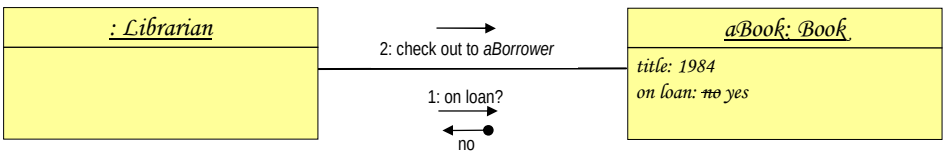
Object-Oriented Analysis and Design Through Scenario Role-Play

Jürgen Börstler

UMINF 04.04

Department of Computing Science
Umeå University, Umeå, SWEDEN





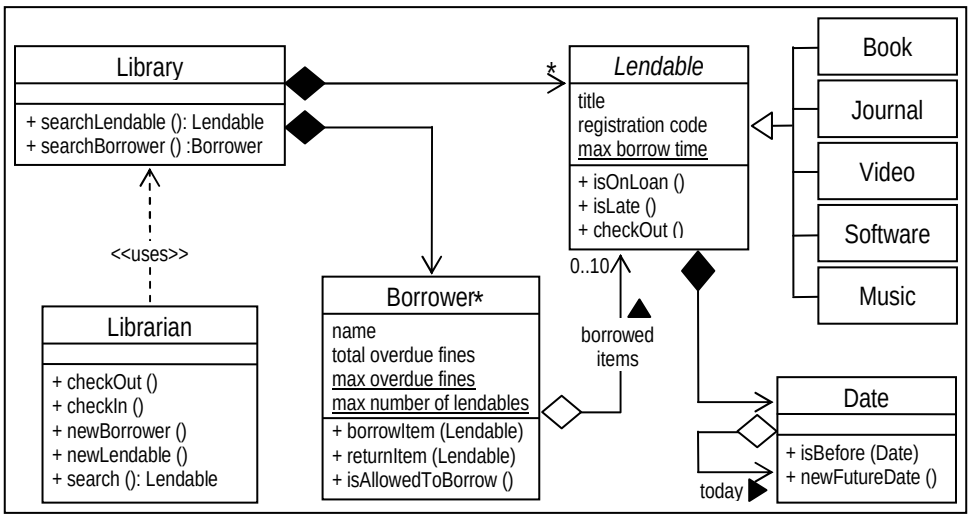


Table of Contents

1 Introduction

The usage of an object-oriented language does not in itself guarantee that the software developed will actually be object-oriented. However, object-oriented languages provide concepts that make it more straightforward to develop object-oriented systems.

> *"Object-oriented software construction is the software development method which bases the architecture of any software system on modules deduced from the types of objects it manipulates (rather than the function or functions that the system is intended to ensure)."* *(Meyer, 1997, p 116)*

In an object-oriented language, the types of objects are described by classes. These classes should be developed in a way that makes them easy to understand, maintain, and reuse. Many development methodologies or processes have been proposed to ensure these properties (Zamir, 1999). However, none of those can guarantee to be successful for all types of problems. It is always the responsibility of the developer(s) to (a) select an appropriate process and (b) take all the time necessary to study the problem and its implications in detail.

In this presentation, we focus on analysis and design. We assume the reader has basic knowledge in (object-oriented) programming. Please note that any type of software development also involves further important activities, like for example testing or documentation. These are however outside of the scope of this presentation. Note furthermore that we will not discuss any aspects of user interface design. User interface design is an important activity in system development. However, it is a complex design activity in itself and beyond the scope of this presentation.

- The goal of **Object-Oriented Analysis (OOA)** is to *fully understand the problem* and all its implications for its potential users. We want to find/identify all things and concepts, i.e. objects, in the application domain, that are relevant for solving this particular problem. Using these objects, their properties and interrelationships, we then build an object-oriented model of the problem domain. This abstract and simplified model of the problem domain will help us to better understand the actual problem and find an appropriate solution.

  Analysis is independent of actual programming languages and user interfaces[1]. This independence has several advantages. It is not necessary to know a specific programming language and/or user interface software to do analysis. This is particularly important, since analysis must be performed in close collaboration with the clients/users. The clients/users are the domain experts and cannot be expected to have a background in computer science. Decisions about actual implementation details should be deferred as long as possible. In the analysis phase, we should not need to worry about such details. They might block our minds and prevent us from finding an elegant "out of the box" solution.

  We can think of the analysis model as an idealised and general object model, which is not corrupted by implementation issues.

  How do we find/identify the relevant objects in this problem? What are their properties and behaviours in this particular context? How do they interact to accomplish the necessary tasks? We want to find out what each object knows and does and how objects collaborate.

- **Object-Oriented Design (OOD)** builds on the results from the analysis stage. Now we have to address certain (but not all) implementation issues and *describe a solution*

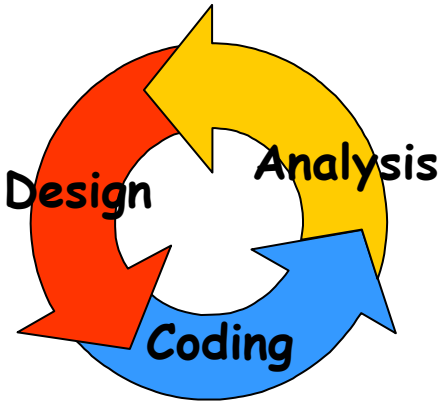
[1] As long as the user interface is not a main part of the actual problem.

to the problem. That includes decisions about actual classes and their properties. We have to define actual attributes to describe what the objects of a class know and we have to define actual methods to describe the behaviours of our objects in an appropriate way.

Now we can no longer be completely independent of programming languages. We might need to pay attention to for example typing or available libraries. It is however recommended to keep language dependencies at a minimum to make the design as general as possible.

- The goal of **Object-Oriented Programming (OOP)** is to finally implement the solution in an actual programming language.

It is important to note that software development is an iterative process. The longer one works with a problem the better one understands it. It will therefore be necessary to review and revise analysis and design models to reflect the knowledge gained.



OOPS! The development of (object-oriented) software involves further important activities, like for example testing and documentation. These are however outside of the scope of this presentation.

Figure 1: Iterative Software Development.

In this presentation, we describe a systematic approach to object-oriented development based on Responsibility-Driven Design (RDD) (Wirfs-Brock et al, 1990) and CRC-cards (Bellin & Simone, 1997). CRC-cards are simple yet powerful for the collaborative development and evaluation of object-oriented models. CRC-cards are abstract versions of potential classes in a software system. The CRC-cards form a high-level model for the software to be developed. This model is tested through scenario role-play. A scenario describes a concrete usage of the system, i.e. a kind of test case. The role-play is the actual execution of test cases. During the role-play the members of the CRC team play the role of objects and act out the scenarios interactively.

A common problem with this approach is that CRC-cards are used as surrogates for classes (in the modelling activities) as well as objects (in the role-play activities). This can easily lead to confusion in novices. We have therefore developed another simple yet powerful tool to circumvent this problem. By means of role-play diagrams, we can make a clear distinction between classes (used in modelling activities) and objects (in the role-play activities). Role-play diagrams are furthermore an excellent tool to control and document the role-play activities.

The remainder of this presentation is organised as follows. In section 2, we give a very brief introduction into scenario role-playing. In sections 3-5 we discuss the role of modelling in software development. A general introduction to modelling is presented in section 3. In sections 4 and 5, we explain the modelling tools we use in our approach to object-oriented analysis and design, CRC-cards and role-play diagrams, respectively. Section 6 presents a detailed step-by-step introduction into the usage of CRC-cards and role-play diagrams for

object-oriented analysis. Object-oriented design and the Unified Modeling Language (UML) are discussed in section 7. A case study in section 8 comprises a detailed example of an actual role-play session.

2 The Power of Scenario Role-Play

Scenario role-play is usually used to evaluate human behaviour in specific yet hypothetical situations. The role-play participants are assigned roles they enact according a predefined scenario, much like actors following a script when playing the characters in play. During the role-play the participants learn a lot about themselves, the other participants, and the roles played. The power of the role-play lies in its interactivity that supports creativity and sharing of knowledge. Role-playing is effective way to simulate or explore hypothetical situations, since the characters and scripts (scenarios) can be easily varied.

In object-oriented software development, the characters are the objects in our system and the scenarios are situations of system usage. Scenario role-play is used in very early stages of software development, before any code is written (or even designed). The characters of our play(s) are therefore only vaguely defined by sketches of potential classes that might be part of our software system. Scenario role-play allows us to explore many alternatives for modelling a software system, i.e. we can test the software before it is actually build.

3 The Role of Modelling

Modelling is an important everyday human activity. Models help us to understand a complex world by focussing on those properties of "reality" we are currently interested in. Irrelevant details are neglected. However, the things we actually model, i.e. include in our models, must reflect reality as closely as possible. A road map for example is an extremely simplified model of a certain part of the earth, focussing on roads that can be travelled by cars and interesting places. Important details relating to travel are included in detail, like the types and accurate lengths and courses of roads, the location of cities, airports, and gas stations etc. Information relating to topography, geology, and vegetation, etc. are added sparingly, if at all. That is fine if you are interested in travelling from A to B, but useless for a botanist searching for a specific kind of flower.

The goal of modelling is to develop an as simple as possible model of the reality that still correctly reflects all important and interesting aspects we are interested in. Using models, we can concentrate on the essential aspects and phenomena of a problem. Models are simpler and easier to understand as reality and we can avoid wasting time on irrelevant details or aspects.

In software development, we build models of software systems to enable us to better understand the systems we are developing. In object-oriented modelling, we focus on the types of objects manipulated by the software systems (c.f. the quote in the beginning of section 1). There are several approaches to object-oriented modelling. In the modelling approach described here, we anthropomorphise objects, i.e. objects are viewed as living entities that can act independently on their own will.

The software objects we model will frequently have real world counterparts in the problem domain. This can make modelling a straightforward activity, but also lead to some confusion. Always remember that we model software objects and not their real world counterparts. Our software objects will often reflect properties of their real world counterparts. However, our software objects will also have properties that their real world counterparts never can have. A

software book object could for example be asked to check itself out from the library. It is therefore very important not to confuse the objects model with the "real thing".

Note that there are several levels of modelling. Aspects, properties, or details that are irrelevant during analysis might very well be important during implementation. Note also that there is no single correct model. Real world aspects can be modelled quite differently, depending on the actual problem. People for example will be modelled quite differently in a course registration system for a university compared to a journal system for a hospital.[2]

4 CRC-cards

CRC-cards are a simple yet powerful tool for collaborative object-oriented modelling. They are used to develop, discuss, and evaluate object-oriented models in an informal way. CRC-cards have originally been developed to teach programmers object-oriented thinking (Beck & Cunningham, 1989), but have since then found wide application outside this context (Bellin & Simone, 1997).

CRC stands for ***C***lass, ***R***esponsibilities and ***C***ollaborators. A CRC-card is a standard index card that has been divided into regions, as shown in Figure 2.

- **Class;** a CRC-card corresponds to a class, i.e. a collection of objects of the same kind. Objects are things of interest in the problem/application domain. An object can be a person, place, thing, event, or concept. The class name is written across the top of the CRC-card. A class should have a single and well-defined purpose that can be described briefly and clearly. It should be named by a noun, noun phrase, or adjective that adequately describes the abstraction. A short description of the purpose of the class is written on the back of the card.
- **Responsibilities;** a responsibility is something a class takes care of; a service the objects of a class provide for other objects. A responsibility can be either to know something or to do something. A book in a library for example should know its title and whether it is out on loan or not (among others). To do something the class usually uses the knowledge it has. If this knowledge is not sufficient for the purpose, the class can get help from collaborators (see below). The responsibilities of a class are written along the left side of the card.
- **Collaborators;** a collaborator is another class "helping" to fulfil a specific responsibility. A collaborator can for example provide further information, or actually take over parts of the original responsibility (by means of the collaborators own responsibilities). A book can for example only know whether it is overdue or not, if it also knows the current date (in Figure 2 this information is provided by the collaborator `Date`). Collaborators are written along the right side of the card in the same row as the corresponding responsibility.
- The back of the card can be used for the class description, comments and miscellaneous details.

CRC-cards are particularly suited for collaborative analysis. A group size of 4-6 people seems to work best. Smaller groups usually lack the right blend of different backgrounds. If possible there should be (at least) analysts/developers and domain experts users on the CRC-team. In

[2] The actual model elements will not even have the same names. A detailed comparison is left as an exercise.

larger groups, it becomes too difficult to reach a consensus. It is important NOT to get bogged down in discussions about implementation details. The goal is to develop, discuss, evaluate and test different object-oriented models.

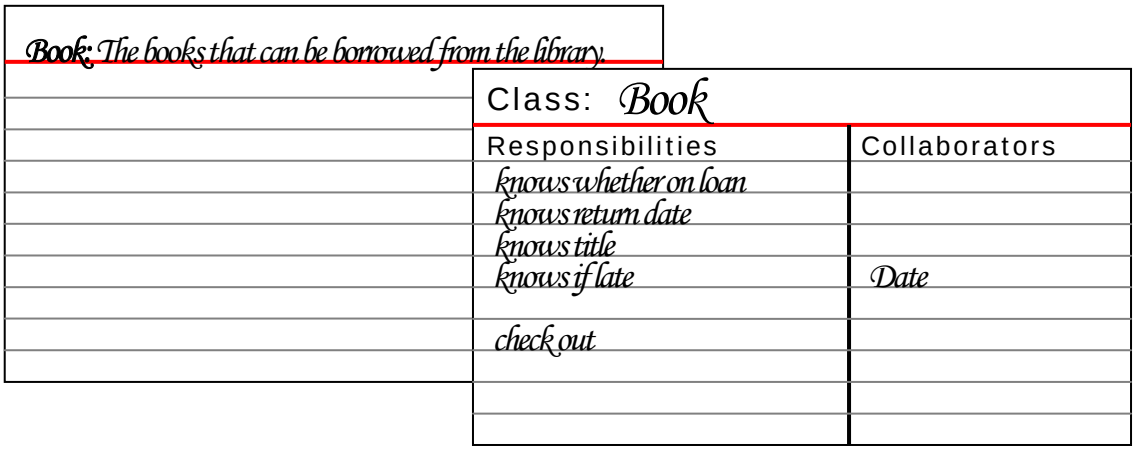


Figure 2: A CRC-card for a `Book` class.

In the CRC approach, we think of the objects as living entities that can act on their own will. Each object operates solely based on the knowledge and behaviour that is described on its corresponding CRC-card.

5 Role-Play Diagrams

Common role-play approaches use the CRC-cards as the "characters" in the role-play (Bellin & Simone, 1997). The CRC-cards "stand in" for the actual objects (Beck & Cunningham, 1989). This can however be very confusing for novices, since the concepts of class and object are not clearly distinguished. We therefore developed role-play diagrams (RPDs) to reinforce this distinction. RPDs do furthermore provide an excellent documentation of the role-play. They can also easily be used to keep track of the role-play as it unfolds and support recovery of the latest consistent state of the role-play.

Role-Play Diagrams (RPDs) are used to document object interaction. The objects in a RPD are instances of the classes modelled by CRC-cards. RPDs are a new type of informal diagrams that cover the most important aspects from UML object and collaboration diagrams[3] (Booch et al, 1999). RPDs are simpler and more informal than their UML counterparts and therefore better suited for recording scenarios in parallel to the role-play activities.

Objects in RPDs are denoted by object cards. An object card is an instance of a CRC-card showing the instance's name, class name and current knowledge as shown in Figure 3. All information presented on the object card must match responsibilities on its corresponding CRC-card. However, we usually only list those properties that are relevant for the current role-play activity. Large post-it notes work fine for this purpose. For each new object, we create a new object card and put it on the white-board or a large sheet of paper. An object card is initialised with the knowledge for this specific instance, using the corresponding CRC-card as a template. In a library system, we might for example have the following object card for the book *1984* (see Figure 3).

[3] In UML2.0, collaboration diagrams have been renamed to communication diagrams (OMG, 2003).

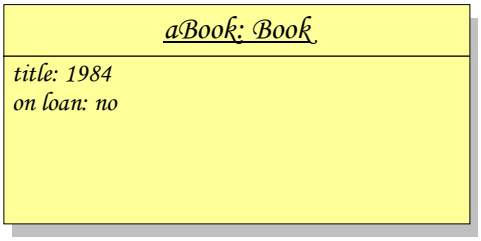


Figure 3: Object card for a `Book` object.

Objects that "know" each other are connected by a line. During the role-play, communication between objects is only possible if their object cards are connected. An object card can only be connected to the collaborators listed on its corresponding CRC-card. New objects can always be created, if their corresponding CRC-card is listed as a collaborator of the creator.

Actual communication is described in the form `number: request` and can be denoted on the communication links as in the example below (see Figure 4). `Number` keeps track of the ordering of requests and `request` corresponds to the actual service requested. A small arrow denotes the direction of the request. Requests can furthermore be annotated with data/object flows to explicitly document information that is sent or returned[4].

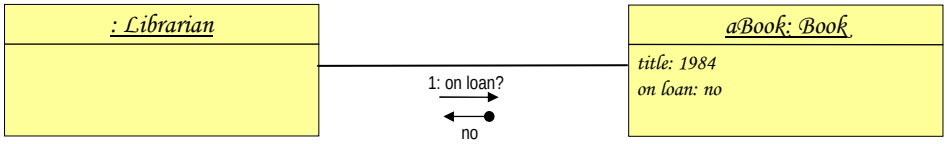


Figure 4: A request between connected object cards.

RPDs must always be consistent with the corresponding CRC-card model. Requests to and knowledge on an object card must match the responsibilities on the corresponding CRC-card. Our example above requires that class `Book` responsibilities to handle the `on loan` and `check out` requests (which is the case according to the CRC-card in Figure 2). Information on the object cards is updated to reflect any changes (see Figure 5).

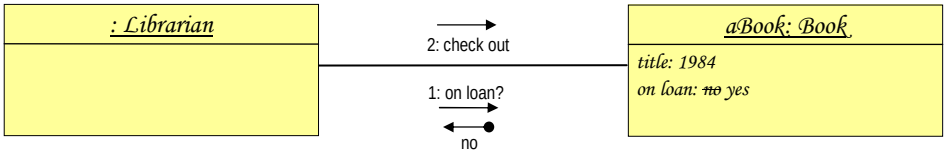


Figure 5: Figure 4 after the second request has been handled.

At the end of a scenario role-play, the RPD provides an exact documentation of what happened, including the state changes. If necessary, this information can easily be translated into for example UML sequence diagrams.

6 OO Analysis

The CRC-card approach is especially well suited, when the problem is not well defined. During OOA the problem and application domains are analysed to understand the problem at hand and identify the things that actually should be done and those that are outside the system

[4] This particular notation is borrowed from the Booch notation's object diagrams (Booch, 1994), that did not make it into the UML standard.

that should be developed. To accomplish that we

- *find candidate classes* by means of brainstorming (section 6.1);
- *filter* the list of *candidates* (6.2);
- *create CRC-cards* for the remaining candidates (6.3);
- *allocate responsibilities* to CRC-cards/classes (6.4);
- *define scenarios* to test/evaluate our model (6.5);
- *prepare the group session* (6.6);
- *"role-play" scenarios* using CRC-cards (6.7);
- *record scenarios* (6.8); and
- *update* CRC-cards and scenarios to reflect your findings (6.9).

Please note that these steps are not performed in strict sequence. When filtering the list of candidates we usually have to consider possible responsibilities to make informed decisions. The last three steps are always performed in parallel.

6.1 Find Candidate Classes

The purpose of this first step is to generate a list of objects that might be of interest for the problem at hand. An object should usually have properties and behaviour. A book object in a library system might for example have (among others) the property `on loan` and the behaviour `compute return date`. If objects differ only in the values of their properties, they are of the same kind and we consider them as objects of the same class. All books in a library for example would be considered as objects of the class `Book` (see figure 2).

If the only difference between objects `Alien` and `Star Trek` is their title and registration number, we would consider them as of the same kind of object. In a simple libary system, they would for example both be objects of the class `Lendable`. However if one of them is a video and the other one is a cassette we would probably associate different behaviours with them. When designing a home entertainment system `Alien` and `Star Trek` would be objects of class `Video` and `Cassette`, respectively.

Classes should be named by a noun/noun phrase or adjective in singular form. Anything that cannot be named given these restrictions is most probably not a suitable candidate for a class (but maybe a responsibility, see section Filter Candidates).

To get the brainstorming started the following tools are very helpful to generate candidates.

- **Checklist approach;** Walk through a list of typical sources for objects, like the following; people, places, tangible things, roles, organisations, abstract things, events, interactions, (sub-)systems, etc.
- **Noun extraction approach;** Underline all noun phrases in the problem statement or other documentation that might be available.[5] This works quite well, if documentation is not too extensive.

Please remember that brainstorming is an idea-generating technique, not an idea-evaluating technique. During the brainstorming session there should be little or no discussion on the

[5] This approach will likely generate a lot of "noise" due to text that is not relevant for the actual problem.

suitability of candidate objects. All candidates should be recorded. Discussions should be deferred until the next step (Filter Candidates). It is recommended to go round in turns give every voice a chance) and collect all candidates visibly for later evaluation (e.g. on a whiteboard or flipchart). The brainstorming stops, when the group runs out of ideas or when a predefined time limit is reached (e.g. 30 minutes).

6.2 Filter Candidates

In the second step, we evaluate our list of candidate classes. The goal is to cut down the number of candidates to a manageable size, preferably no more than ten. All candidates irrelevant for the problem at hand are discarded.

According to Wirfs-Brock and Kean (2003, p 106) "[c]andidates generally represent work performed by your software, things your software affects, information, control and decision making, ways to structure and arrange groups of objects, and representations of things in the world that your software needs to know something about".

- Merge synonyms into the candidate with the most suitable name (see comments on naming classes in section (Create CRC-cards).
- Discard candidates you cannot properly name by a noun, noun phrase, or adjective. Look particularly for nouns that are actually verbs and/or describe services. Such candidate classes are usually responsibilities in disguise and should therefore be turned into a responsibility of a suitable candidate class. In a library we might for example have the candidates `searching`, `lending` and `book`, derived from the sentence "… searching for and lending of books …." In this example `searching` and `lending` should be discarded as candidates and turned into responsibilities of one or more other candidate classes.
- Discard candidates that seem insignificant or vague, especially if you cannot describe their purpose.
- Discard candidates that describe implementation details. The purpose of analysis is to model the problem, not the solution. Considering implementation details too early will only constrain our solution space.
- Discard candidates that model (simple) properties of other candidates. They are often (knowing) responsibilities of other candidates. In a library system for example, registration codes should be modelled as responsibility of for example a class `Book` and not as a class on its own.
- Discard candidates that do not have any responsibilities.
- Discard candidates that model user interfaces. User interface details are considered as implementation details. Usually such details are not important to understand the problem.[6]
- Discard candidates that are outside the system, like users and external systems. You need only keep such candidates, if the system's behaviour depends on their properties (i.e. needs to keep data about them).
- Discard candidates that are outside the actual problem space. Candidate classes should

[6] To make the role-playing of scenarios more intuitive we will however assume one central GUI object (see sections 6.7 and 6.8).

be based on actual or highly probable requirements or demands. Always keep in mind what the intended software system is supposed to do. You should always try to foresee changes, extensions and new demands, but you should still deal with the actual problem at hand. Do not try to solve the wrong problem. Do not make the model more complicated than necessary. In a general library system for example, it might be possible to propose unavailable items for purchase. However, you should carefully evaluate whether this is required in the actual system.

- Discard candidates that are "names" for the system itself, like "the system" or "the application". Our goal is to find a collection of classes that together form a model for our system.
- Discard candidates that are instances of other candidates.
- Merge candidates representing actual objects of the same class. A CRC card models a class, i.e. a collection of objects of the same kind. We want to abstract from concrete objects and model classes only.
- Merge candidates with largely overlapping responsibilities. Try to find one abstraction that covers all candidates (if possible). In a library system, for example there might be different kinds of entities that can be borrowed, like books and films. If these are handled in exactly the same way by the system, it might be better to merge them into a new candidate `Lendable` and treat them as of the same kind.

The goal of the first step is to find a number of useful candidates. It is not necessary to find all candidates at once. More candidates will likely be uncovered during later steps (see section Update CRC-cards and Scenarios).

6.3 Create CRC-cards

Now we produce one CRC card for each remaining candidate class, according to our description in section CRC-cards. A good class captures a single and well-defined abstraction that can be adequately named by a noun, noun phrase, or adjective. The responsibilities of a good class can be described briefly and clearly.

Class names should be chosen carefully. A good name is specific and descriptive and can therefore easily be associated with the intended responsibilities of the class. In a library system, `Borrower` would be a much better class name for describing the class of people that borrow books as for example `User`, `Person`, or `Client`.

The back of the card should be used to briefly and clearly describe the purpose of the class to make sure that the class name is interpreted correctly. If this is not possible using only a few sentences your class has probably too many purposes and responsibilities and you should consider splitting it up. Make sure that all members of your CRC team agree on the class name and purpose and interpret the class description in the same way.

Although we noted that user interface design is outside the scope of our approach, we suggest to prepare two additional cards; one for the user interface and another one for interfaces with external systems. However, these cards are not real CRC-cards. They are only used for taking notes on interface aspects that must be considered later/during design.

6.4 Allocate Responsibilities

In the next step, we assign responsibilities to classes. Responsibilities belong to the service

providers and are listed on the CRC-cards that provide the services. Responsibilities and "system intelligence" should be shared and distributed among classes. There should be no single class responsible for all knowledge or all behaviour. All classes should usually have knowledge and behaviour. One can think of the objects as living entities that can act on their own will. Each object operates based on its knowledge and behaviour as described on its corresponding CRC-card.

First, we add responsibilities that are obvious from the problem description or application domain. In a library system, for example we need to lend out books. It seems therefore sensible to add a responsibility `check out` to the `Book` card, since books have the knowledge to support this responsibility (see figure 2). To identify further responsibilities we could do another brainstorming session or look for the verbs in our problem description, similar to the noun extraction approach for the candidate classes in section Find Candidate Classes. However, it is not at all necessary to identify all responsibilities in this step. It is sufficient to identify a number of key responsibilities to provide a starting point for the scenario role-play.

"[R]esponsibility denotes both duty and power … an object should fulfil its obligations, and should have the ability to accomplish them" (Biddle et al, 2002, p. 203). For each responsibility, we evaluate whether the objects need collaborators to accomplish their obligations. In case we need collaborators, we have to make sure that the collaborators actually are equipped with the necessary responsibilities. Quite often, we identify similar or even identical responsibilities for different classes. This is a sign for collaboration between these classes. Please note that classes should not have too many overlapping responsibilities. Such classes should be merged (see section Filter Candidates). On the other hand, you should avoid having a single class responsible for everything important in your model. This class will dominate your model and degrade other classes to "dumb" helper classes.

Please make sure that you do not discuss implementation details. This is not of interest in this stage and in fact counterproductive, since it prevents you from identifying alternative distributions of responsibilities.

6.5 Define Scenarios

Scenarios are the test cases for our CRC-card model. Each scenario is a specific example of system usage. A scenario is like a script in a play that is enacted by the objects defined by our CRC-card model. When enacting a scenario we follow defined responsibilities and collaborations to "test" whether our model can handle all eventualities that may occur (see section "Role-Play" Scenarios). Scenarios describe concrete "what happens if" situations.

Scenarios must be concrete and clearly defined. Otherwise, we can easily get lost in details, when the scenarios are role-played. A scenario compares quite well to a traditional system test case and comprises (1) the system function to be "tested", (2) actual test data (assumptions) and (3) expected result(s). A scenario for a library system could be the following:

> *John Doe will borrow the book 1984; John Doe is a registered borrower and has never borrowed anything before; the book 1984 is available. After borrowing, John Doe will be registered as borrower of 1984 and 1984 will have a valid return date.*

To really validate a model we would need to test very many scenarios. For example to test what happens when John Doe is not a registered borrower, has outstanding fines, or overdue books. What if the book 1984 is on loan, etc? It is important to define a limited number of scenarios that nevertheless cover all interesting situations.

Scenarios need not cover a complete user function, like the example above. It is suitable to split long or complex scenarios into parts that can be handled separately, like for example

The available book 1984 is checked out to registered borrower John Doe.

When role-playing more complex scenarios we can then skip over the details that have been tested already.

6.6 Prepare Group Session

A CRC-team should consist of about 4-6 people with different backgrounds. If possible there should be (at least) analysts/developers and domain experts users on the CRC-team. Before the actual role-play starts, the group members should make sure that they agree on all CRC-card descriptions.

Make sure you have all the materials available to perform the role-play smoothly, like some extra index cards for new CRC-cards and the materials for recording the scenarios (see section Role-Play Diagrams).

One team member should act as a scribe. The scribe will record the role-play as it unfolds and help the team to stay on the track of a scenario (see section Record Scenarios for details). The CRC-cards are then distributed among the other team members. Each team member is responsible for the CRC-cards assigned to him or her. This means that he or she is also responsible for all object cards that are instances of these CRC-cards and will therefore act out all objects corresponding to those cards. Acting is done by saying aloud what a certain object will do upon a request. All acting must be done according to the responsibilities noted on the CRC-cards. You must not assume anything that is not denoted on your CRC-card(s) and you must act only when it is your "turn". The scribe can step in when user interaction or interaction with external systems is required.

During the first role-plays, we will uncover many missing details. It will take a few scenarios before the model stabilises. We strongly recommend starting with a very simple scenario for a "normal case". Otherwise, there is a high chance that you get lost in discussions about large numbers of missing details.

6.7 "Role-Play" Scenarios

By means of role-playing scenarios we kind of simulate how the future system would work, assumed it is build according to our CRC-card model. Please remember that we are mainly interested in the problem domain. For now, we want to neglect implementation details and issues related to user interfaces or interfaces to other systems.

The CRC approach anthropomorphises objects, i.e. an object is interpreted as a living entity that can act independently on its own will. An object takes a self-centred view. A book object in a library system would for example take the following view; "Hi, I'm the book XYZ, what can I do for you?" It should not be necessary to ask a third party object about things that this specific book naturally should know by itself (for example whether it is on loan).

Before we start the actual role-play we review all assumptions made in the selected scenario. For our example, *John Doe will borrow the book 1984* from section 6.5 we have the following assumptions. Explicitly we have stated that *John Doe* is a registered borrower and has never borrowed before. Therefore, he will not have any outstanding fines. The book is available (i.e. not on loan). Prepare a Role-Play Diagram (RPD) that models a suitable starting situation. To simplify role-playing, we assume there is one central user interface object that takes care of

all user interaction and another one that takes care of all interaction to external systems (in case there are any).

A possible starting situation for a simple library system could look like the one shown in Figure 6. In this RPD we have assumed two books and two borrowers[7]. We have assumed a `Librarian` object that "knows" all books and registered borrowers. We furthermore assume that on start-up of the system the `theGUI` only knows about the `Librarian` object.

What does *John Doe will borrow the book 1984* mean for our system? How do we translate this request to an object's responsibility?

- Which object will start up the work that needs to be done? Mark the corresponding object card in your RPD as "active" and give "control" to the holder of the corresponding CRC-card.

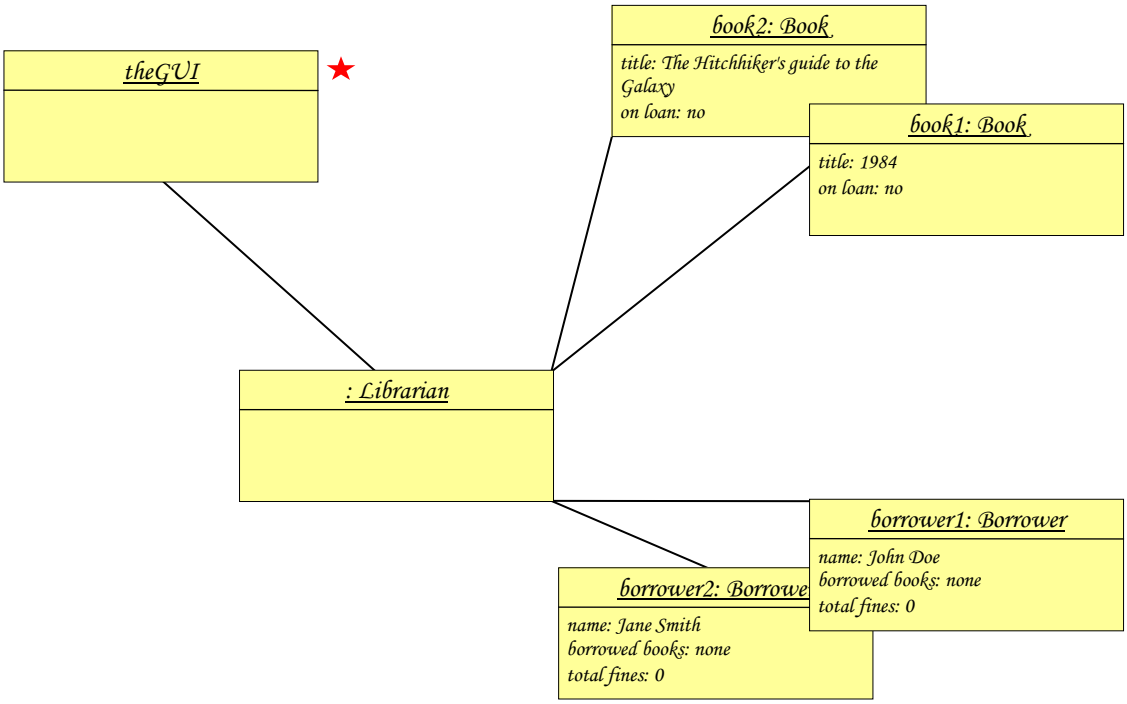


Figure 6: Possible starting situation for scenario *John Doe will borrow the book 1984* with `theGUI` marked as active.

- What does the active object need to know to do the requested work? Does it have the necessary responsibilities to fulfil the request? The active object says aloud all actions the active object would perform. Upon the request `on loan`, the `aBook` object in Figure 4 might for example respond as follows.

  > *"I have the responsibility to know whether I am on loan (c.f. the corresponding CRC-card in Figure 2). Currently I am not on loan (c.f. `aBook`'s object card in Figure 4). The answer is no."*[8]

  After a request has been successfully handled, control goes back to the requester.

  If responsibilities are missing on the corresponding CRC-card, add them now and

[7] For the role-play, two books and borrowers are sufficient. Adding further books or borrowers would make the scenario more realistic, but would not add to the scenario's inherent complexity.

[8] The `aBook` object is an instance of the `Book` class. It would therefore be the responsibility of the team member responsible for the `Book` CRC-card to "play" that part of the scenario.

update the object cards accordingly (see section Update CRC-cards and Scenarios for more information on updating). If too much is missing to continue the role-play, you can stop and revise the CRC model before continuing the role-play.

- Are there other objects your object needs to collaborate with? Are their object cards present in the current RPD? Are their object cards connected to the current object's object card? Remember that communication is only possible between connected object cards. An object can only collaborate with another object, if it exactly knows who the actual collaborator is.[9]

  Add object cards to your RPD as necessary and connect them to the objects that know about them. Consider carefully how a "new" object is created and which object is responsible for its creation. Also, consider carefully how objects get to know about each other. An object always knows the objects it created. Other objects must be somehow "announced" to it. This can for example be done through a request carrying this information (as for example in request `3: borrow book1` in Figure 7). Collaborators can also be obtained as a result of a request. A search request for a book for example might return a specific book object.

- Be careful not to skip over details you think are obvious (the devil is often in the details). It is important to carefully walk through a scenario step by step. Sometimes however, you want to skip over certain details to be able to come to an end with a scenario. Make sure to address these details later, maybe as an additional scenario to complete your role-play.

6.8 Record Scenarios

The scribe records scenarios as they unfold during the role-play. Recording starts with an initial RPD modelling the starting situation for a certain scenario (see section "Role-Play" Scenarios and Figure 6). The scribe keeps track of all activities and updates the RPD accordingly. Recording scenarios in this way is very useful in several ways.

- During the role-play, the CRC team can easily check the current knowledge of the involved objects or whether two objects can communicate. This keeps the role-play on track and minimises the number of details the CRC team needs to memorise.
- Completed scenarios can be easily replayed after changes to the CRC model to verify whether they still work.
- They can be used to explain to others how the system works.
- If you get lost in a scenario, it is easy to backtrack a few steps and recover a useful status to continue the role-play.
- They can guide the implementation of the system.
- They can be used as system documentation.[10]

In Figure 7 below, we show the resulting RPD after completion of the scenario *John Doe will*

[9] The CRC-card model is static, i.e. on the CRC-card you can only document the class of the collaborator(s). The actual object(s) needed for collaboration will usually be determined dynamically during the role-play, i.e. execution.

[10] The simplest way would be to take photos of the RPDs after the role-play. You could also transform them into UML communication or sequence diagrams (see sections 7.4 and 7.5 for details).

borrow the book 1984.[11] After completion of a scenario "control" returns back to the starting object (i.e. `theGUI`, c.f. Figure 6). Please note that the connecting line between `book1` and `borrower1` is not available at scenario start. It is drawn after request `3`, i.e. when the librarian object has identified the actual objects and can tell `borrower1` to which book object it has to connect.

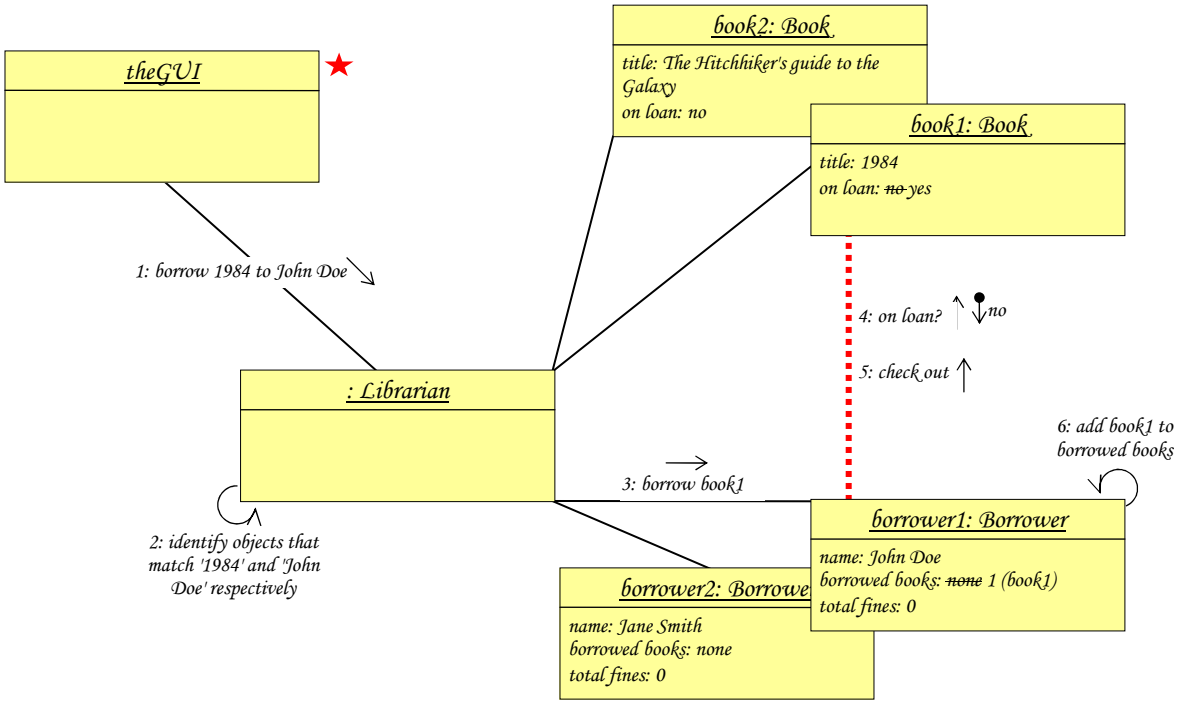


Figure 7: RPD from Figure 6 after completion of scenario *John Doe will borrow the book 1984*.

6.9 Update CRC-cards and Scenarios

During the role-play, you will likely realise that your initial models are incomplete, inconsistent, or even partly wrong. The role-play uncovers many problems that need to be fixed. Whenever such problems are detected, they need to be discussed and solved properly. All changes to your models must be made in a consistent way. Whenever a CRC-card is changed, you have to make sure that scenarios and RPDs are updated accordingly. Extensive changes might make it necessary to add scenarios to evaluate the new or changed parts.

Missing responsibilities or collaborators can be added to suitable CRC-cards during the role-play. Be careful to distribute responsibilities as discussed in section Allocate Responsibilities. Whenever you have a CRC-card with no room left for new responsibilities, you should consider splitting it.

6.10 Advantages of the CRC/RPD Approach

CRC-cards are a simple yet powerful tool for collaborative object-oriented modelling. They help to explore a problem space and to better understand the problem at hand. Using scenario role-play, different object-oriented models can be evaluated in an early stage of software development. The approach described here gives a good understanding of the objects,

[11] The scenario is somewhat simplified. We have for example ignored the handling of return dates. More details can be found in the case study, see section 8.6.

responsibilities, and interactions involved in a solution for the problem.

The main advantages can be summarised as follows.

- The approach is low-tech and independent of programming languages. This makes it well suited for collaborative modelling in teams with people with very different backgrounds (analysts, developers, users, etc.).
- The CRC-cards and RPDs form an excellent documentation of the analysis phase and initial design of a system. Using the RPDs one can easily see how the classes are supposed to work.
- Through scenario role-play, one can easily test alternative analysis and design models using different sets of cards and/or varying responsibilities. This allows for many tests long before any code needs to be written.

7 OO Design

The CRC/RPD approach can also be successfully applied in the design phase. We take our models from the analysis phase as input and refine and extend them by addressing high-level implementation issues, like for example the actual organisation, management and storage of objects, code management, user interface details, etc. During design, we also start to consider programming language issues, like for example typing, visibility, packaging, or the usage of library classes. Language dependencies should however be kept at a minimum to make the design as general as possible.

In the design phase, we make decisions about actual classes and their properties. Responsibilities will gradually be refined into attributes and methods. For the collaborators we determine how the actual collaborations are established. This can be done by creation, sending arguments, or receiving results (see also the discussion about "announcing" collaborators in section "Role-Play" Scenarios). This will lead to further refinements of our attributes and methods lists. We also decide whether both collaborators need to know about each. Quite often one-way communication is sufficient making the design somewhat simpler.[12]

For the attributes, we will furthermore decide whether they are class wide properties or object specific properties. Class wide properties are identical for all instances of a class. Class wide properties can therefore be referenced through the class itself; there is no need for "announcing" specific objects. This simplifies collaboration.

During design, we should follow established principles that will lead us to solutions that are easy to understand and to maintain. Some of these principles are automatically enforced by the RPD/CRC approach, like separation of concerns, even distribution of responsibilities, and modularity.

In a good design information hiding should be enforced. A good class has a single and well-defined purpose. As few as possible properties should be public. Attributes should never be public. Classes should communicate with as few other classes as possible and all communication should be explicit.[13] If communication must take place, as little as possible

[12] Please note that it is not necessary to know the caller for returning information upon a request. This wouldn't even be possible. Library classes for example cannot know in advance by whom they will be used in the future.

[13] This requirement of explicit communication is also known as the Law Of Demeter (Lieberherr & Holland, 1989). It states that a method `m` in a class `C` must only refer objects that are either created by `m`, attributes of `C`,

information should be exchanged. Following these principles minimises dependencies between classes. Changes made in one class will therefore only propagate to few other classes, if at all. Figure 8 shows how the number of classes in a system affects its overall complexity. If a system is implemented using only few classes, we will only need little interaction between classes. However, the single classes will become very large and complex. If the system is implemented using many small and simple classes, we will need complex interactions between classes to achieve our goals. Unfortunately, we rarely can achieve both, small classes and little interaction between classes. It is therefore important to balance class and interaction complexities in the best way.

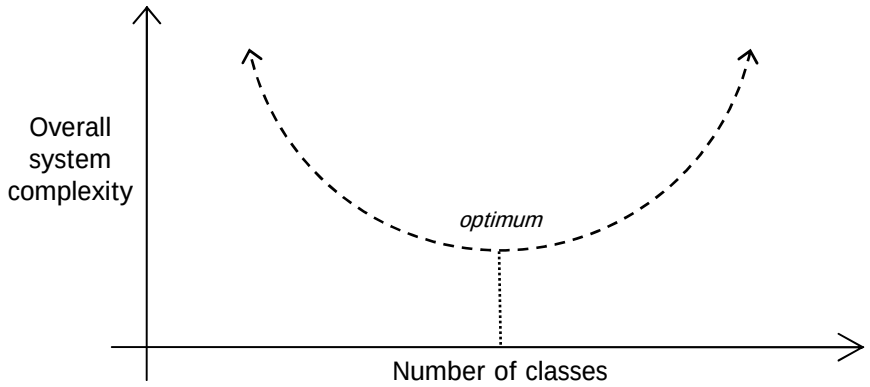


Figure 8: Balancing the number of classes in a given system.

The CRC/RPD approach can be applied exactly as described before. The only difference is that more attention is given to solution domain issues. Evaluating designs in this way has many advantages.

- Models are easy to change; so many design alternatives can be tested.
- Design decisions and the interactions between objects are well documented.
- It is less likely that implementation will run into problems caused by bad design decisions.
- The resulting designs will be less vulnerable to changes, since many alternatives have already been tested.

We might also choose to use more formal notations for our classes and scenarios. At the end of the design phase, we want to have models that can actually be used as input to software development tools. Examples of such more formal models are described in the remainder of this section.

7.1 The Unified Modeling Language

The Unified Modeling Language (UML) is a graphical notation for software system modelling that is widely used in the software industry. The UML was developed in the beginning/mid 90's and version 1.1 was standardised in 1997. A major revision to the UML (UML 2.0) was released in 2003. UML 2.0 comprises 13 diagram types[14] to model physical, static, and dynamic aspects of a software system.

---

arguments of `m`, global variables, or the pseudo variables `self`/`this`.

[14] The UML standard describes diagram elements in great detail, but does not exactly prescribe which diagram elements can be used with which types of diagrams (OMG, 2003).

We will shortly describe three types of diagrams: class diagrams, communication diagrams (formerly called collaboration diagrams), and sequence diagrams. Class diagrams are used to describe the static structure of a software system. A CRC-card model can more formally be described by class diagram. Communication and sequence diagrams are used to describe object interactions. They can be used to more formally describe scenarios.

Please note that the UML is a notation, not a method. The UML does not define when, why, and how to use which diagrams. It only describes the meaning of legally composed diagram elements.

In the following subsections, we introduce the basic notations for class, communication, and sequence diagrams. The UML provides many more diagram elements as described here. Our purpose is to provide a starting point. A comprehensive overview of the UML 2.0 can be found in (Fowler, 2004).

7.2 UML Class Diagrams

Class diagrams describe the classes of a software system and their interrelationships. Classes are denoted by rectangles with several so-called compartments, usually three. The topmost compartment contains the class name. The second and third compartments contain attributes and methods, respectively. Further optional compartments can contain further information, like for example responsibilities.

Attributes and methods can be described in various degrees of detail. For each feature[15] annotations can be added to define visibility, type, multiplicity, parameters, initialisation, and constraints (if applicable). Visibility is shown by symbols preceding the feature name; – (private), # (protected), + (public). Most of these details are however not important on analysis and design level. Class names should start with uppercase letters, feature names with lowercase letters.

Comment notes can be connected to any diagram element.

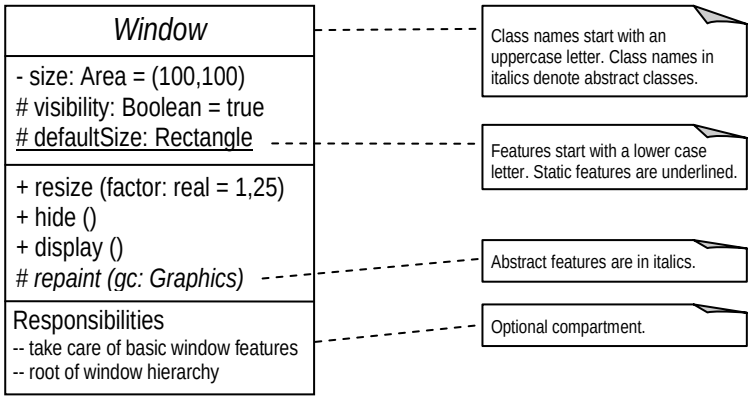


Figure 9: Example class with implementation level details and comment notes.

Relationships between classes describe in which ways the objects of the connected classes can collaborate. Dependency defines a *usage relationship*. A change in the used class (class `B` in Figure 10) may affect any classes that use it. The used class however is not affected by changes to the classes that use it.

[15] In UML, properties of diagram elements are collectively referred to as features, i.e. attributes and methods are features of a class.

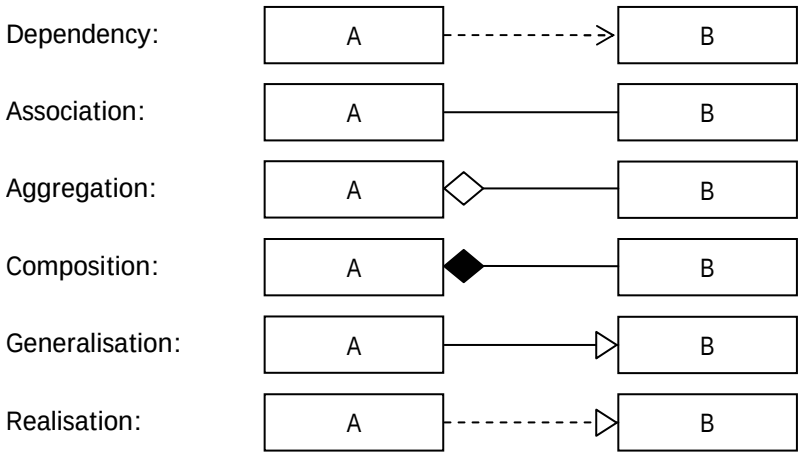


Figure 10: Notations for different class relationships in the UML.

An association describes a structural relationship. Associations should be named using verb phrases and can have cardinalities. Solid arrows near the association names signify reading directions. Associations can be navigated in both directions, if not stated otherwise, i.e. the objects of the associated classes know about each other. In Figure 11 we have four associations. One of them (`passed courses`) is unidirectional, i.e. a `Student` object knows its passed courses (of type `Course`), but a `Course` object cannot navigate to the students that have passed it. Cardinalities express multiplicity of the association. The bidirectional association `teaches` for example describes that (a) each `Lecturer` object can relate to `0-3` objects of type `CourseOffering` and (b) each `CourseOffering` object can relate to `1-3 Lecturer` objects. A star denotes unbounded multiplicity (including zero). In Figure 11, a `Course` object can have any number of `pre-requisite` relationships (of type `Course`) and can also be the prerequisite of an unlimited number of `Course` objects.

Aggregation and composition denote a *part-of* or *has-relationship*. They are specific associations. The diamond is drawn at the composite end of the relationship. In our example in Figure 11, each `Course` object has an unlimited number of `CourseOffering` objects. Composition is the stronger of both relationships. Contrary to aggregation it means ownership, i.e. deleting the composite will also delete all its components. In aggregation, components can be shared.

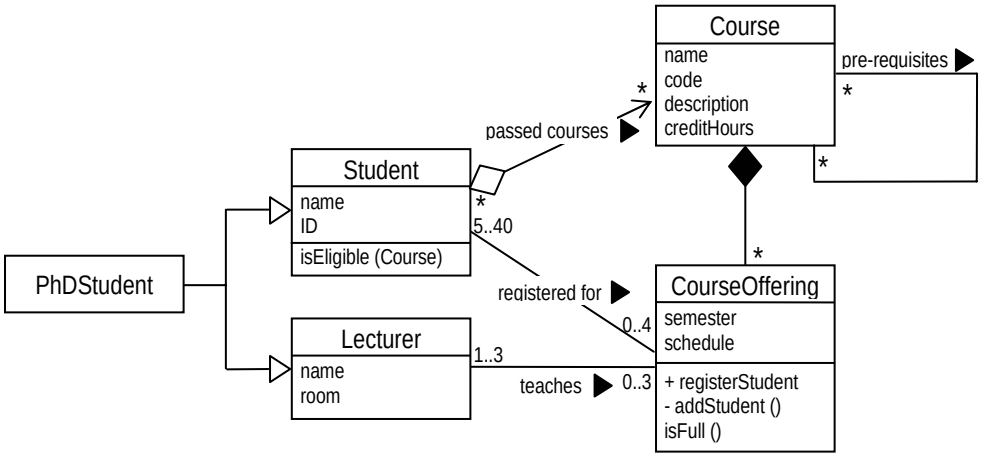


Figure 11: Example UML class diagram with analysis level and (some) design level details.

Generalisation denotes subtyping. In our example in Figure 11, `PhDStudent` is a subtype of

both `Student` and `Lecturer`. I.e. an object of type `PhDStudent` can be used wherever objects of type `Student` or `Lecturer` are required.

Realisation denotes the implements relationship and corresponds to Java's implements construct.

7.3 UML Object Diagrams

An object diagram is a snapshot of a system state. It shows the instances of interest of selected classes at a point in time. The notation is quite similar to class diagrams. However, object diagrams are more concrete than class diagrams and show actual objects and some of their values and interrelationships. Object names should start with lowercase letters and are underlined to discriminate object and class boxes.

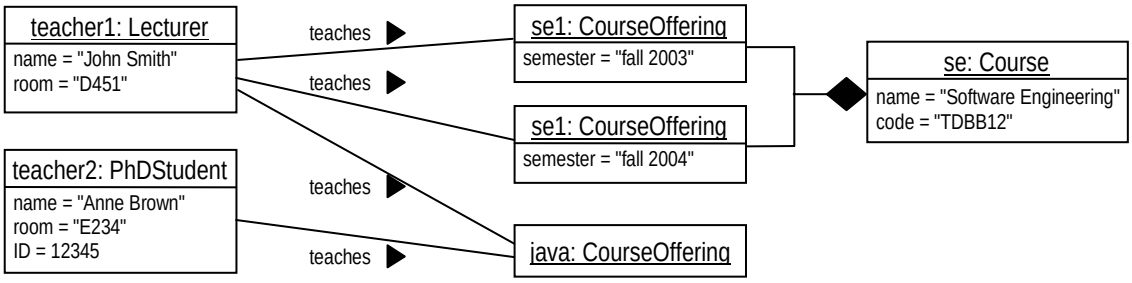


Figure 12: Object diagram corresponding to the class diagram in Figure 11.

7.4 UML Communication Diagrams

Communication diagrams describe object interactions by focussing on the structural relationships between objects. Communication diagrams have much in common with our RPDs, but are more formal and provide a richer notation for details. The notation for messages for example includes a more elaborate numbering scheme, guards (enclosed in "[]"), parameters and assignment of results (as in message `1.1`). The state of objects is however not shown in communication diagrams. Communication diagrams do also provide notational elements to describe asynchronous communication, branching, iteration, etc.

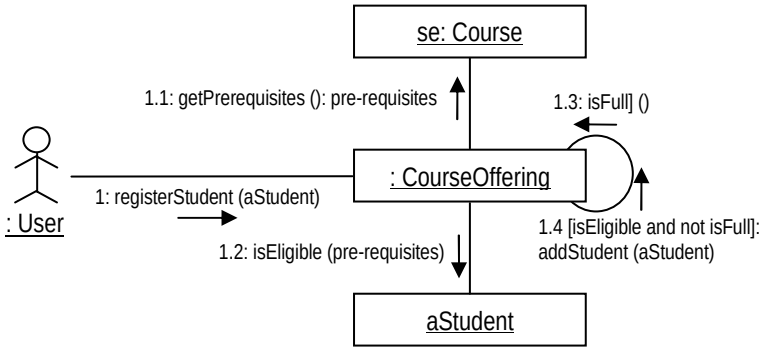


Figure 13: Communication diagram corresponding to the class diagram in Figure 11.

In communication diagrams, we do not show actual values of objects or details about actual relationships (as in object diagrams). Objects can be denoted with their type (as in `se: Course`) or without their type (as in `aStudent`) and can even be anonymous (as in `: CourseOffering`). The stick figure represents a so-called actor, a person or external system that can interact with the system.

7.5 UML Sequence Diagrams

Sequence diagrams describe object interactions by focussing on the relative timing of interactions. The interacting objects are listed on the top of the diagram. A lifeline below an object represents its lifetime. Sequence diagrams and communication diagrams are (almost) semantically equivalent. However, sequence diagrams are more popular, since they show the order and timing of interactions more clearly (from top to bottom of the diagram).

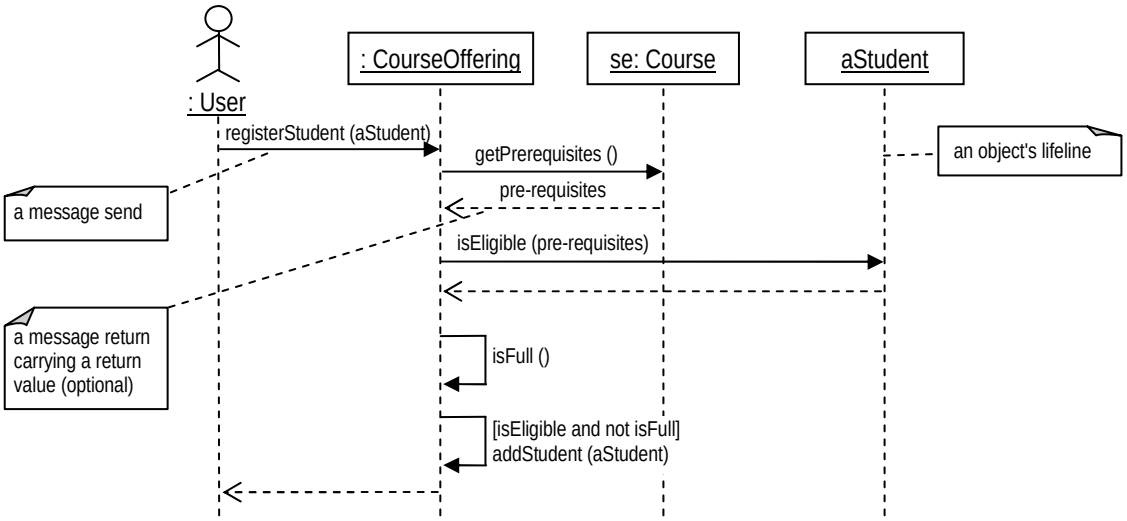


Figure 14: Sequence diagram equivalent to the communication diagram in Figure 13.

8 Case Study: a Library System

In this section, we present a case study for the application of our approach. We are modelling a library system for a university department. The example presented here is a slightly revised version of the CRC case study presented by Wilkinson (1995).

We start with the following problem statement.

> This application will support the operations of a technical library for a university department. This includes the searching for and lending of technical library materials, including books, videos, and technical journals. All library items have registration code (research area code + running number).
>
> Each borrower can borrow up to 10 items. Each type of library item can be borrowed for a different period of time (books 6 weeks, journals 3 days, videos 1 week). If returned after their due date, the employee will be charged a fine, based on the type of item (books 5:-/day, journals and videos 20:-/day).
>
> Materials will be lent to employees only if they have (1) no overdue lendables, (2) fewer than 10 articles out, and (3) total fines less than 100:-.

In the following subsections, we apply the CRC/RPD approach step by step as described in sections 6.1-6.9. In the final subsection, we present possible further requirements for our library system. These can be used for an exercise turn to practise the approach.

8.1 Find Candidate Classes

Using the noun extraction approach, we underline all noun phrases in our problem description.

> This application will support the operations of a technical library for a university department. This includes the searching for and lending of technical library materials, including books, videos, and technical journals. All library items have registration code (research area code + running number).
>
> Each employee can borrow up to 10 items. Each type of library item can be borrowed for a different period of time (books 6 weeks, journals 3 days, videos 1 week). If returned after their due date, the borrower will be charged a fine, based on the type of item (books 5:-/day, journals and videos 20:-/day).
>
> Materials will be lent to users only if they have (1) no overdue lendables, (2) fewer than 10 articles out, and (3) total fines less than 100:-.

Discarding the most obvious duplicates and synonyms, we get the following initial list of candidates: application, operations, technical library, university department, searching, lending, technical library materials, books, videos, technical journals, library items, registration code, research area code, running number, employee, type of library item, period of time, due date, borrower, total fines, users, overdue lendables, articles.

After further brainstorming, the CRC team might propose librarian as a further candidate and recognise further items that a library might need to handle, like for example software and music.

8.2 Filter Candidates

Now we will discuss which candidates should be kept. It is important that the team agree on a candidate's description and purpose. The team member who proposed a candidate should therefore shortly explain his or her motives for doing so.

In the following we will discuss all candidates in our list.

- Application; the system itself. ⇒ Discard.
- Operations; irrelevant noise. ⇒ Discard.
- Technical library; the system itself? No. In this case, we refer to the stock or collection of things available for lending in the library. In our system, we would need a class responsible for keeping track of the stock ⇒ OK: Library.
- University department; outside the problem. ⇒ Discard.
- Searching and lending; verbs used as nouns. ⇒ responsibilities ⇒ Discard.
- Technical library materials; stock of the library. We have already chosen Library as responsible for that. ⇒ Discard.
- Books, videos, technical journals, software, and music; the objects in the library that represent books, videos, etc. Do they differ sufficiently to justify the definition of different classes? Our problem lists two differences; overdue fines and lending periods are different depending on the kind of object. But there are further (subtle) differences. Videos and music for example have playing times. Software has system requirements as an important property. ⇒ OK: Book, Video, Journal, Software, and Music.
- Library items; general term for books, videos, etc. Useful for handling books, videos, etc. when we do not want/need to know the actual kind of object. Here we can keep the responsibilities that are common for all library items irrespective of their kind. ⇒ OK: Library Item (superclass of Book, Video, etc.).
- Registration code, research area code, and running number; they are needed in our

system, but are there any specific responsibilities connected to them? Probably not, they are simple properties of Library Items. This might however be different, if our system should be responsible for assigning these codes in some automatic way. Currently, this is not the case. ⇒ Discard.

- Employee; longer down in our list of candidates we also have borrower and users. What is the difference between those three? Our system is only interested of users that actually borrow or are allowed to borrow books. It is not clear from the problem statement whether it is the responsibility of our system to check, if a borrower actually is an employee. For now, we assume that this is outside the scope of our library system and our system only needs information about actual borrowers. In this case, borrower would be the best and most specific term to name a class responsible for borrower information and activities. ⇒ OK: Borrower.
- Type of library item; irrelevant. This information is encoded in the different classes for the different types of library items available. ⇒ Discard.
- Period of time; the maximum time a certain type of library item can be borrowed. This is a simple property of library items. ⇒ Discard.
- Due date; same as period of time? No, hold on. There are actual responsibilities for due dates. Somehow, we need to check whether a due date has been passed. We do also need a class responsible for the generation of due dates, when library items are checked out from the library. ⇒ OK: Date.
- Borrower; see Employee.
- Fine; there are different kinds of fines. Each type of library item has a fine per day to compute the fine for late returns. Borrowers have a total fine that must not exceed a certain amount. This amount is another kind of fine. However, all these fines are simple properties in different classes and do not have any specific responsibilities on their own. ⇒ Discard.
- Users; see Employee.
- Overdue lendables; having or not having overdue lendables is a simple property of borrowers. ⇒ Discard.
- Articles; used here as a synonym for library items. ⇒ Discard.
- Librarian; this would be the class responsible for taking user requests, like checking in, checking out, and searching for library items. But hold on. Isn't that the same as Library? No. We said that Library is responsible for the stock of library items. But if Librarian is responsible for checking in, checking out, and searching, then there is nothing left for Library? On the other hand, if we merge Library and Librarian into one class this class will probably have too many responsibilities. ⇒ Let's see.

When reconsidering this list we find that Lendable is much better a class name than Library Item. We therefore replace Library Item by Lendable. We already have recognised the risk for a combined Library/Librarian class of becoming too large. We will therefore start with both of them, since it is usually much easier to merge classes as to split them. This gives us the following list of candidates: Lendable, Library, Librarian, Book, Video, Journal, Software, Music, Borrower, and Date.

8.3 Create CRC-cards and Allocate Responsibilities

The responsibilities for Lendable, Book, Video, Journal, Software, and Music are almost identical. For the role-play, it would not matter which one is actually used. We will therefore select only one representative to continue with (Book). This will simplify our further activities considerably. At the end of this subsection, we will shortly outline how the remaining CRC-cards would look like (see Figure 20-Figure 22).

Remember that an object does not need a collaborator for remembering information. Collaborators are only needed, if they actually provide help.

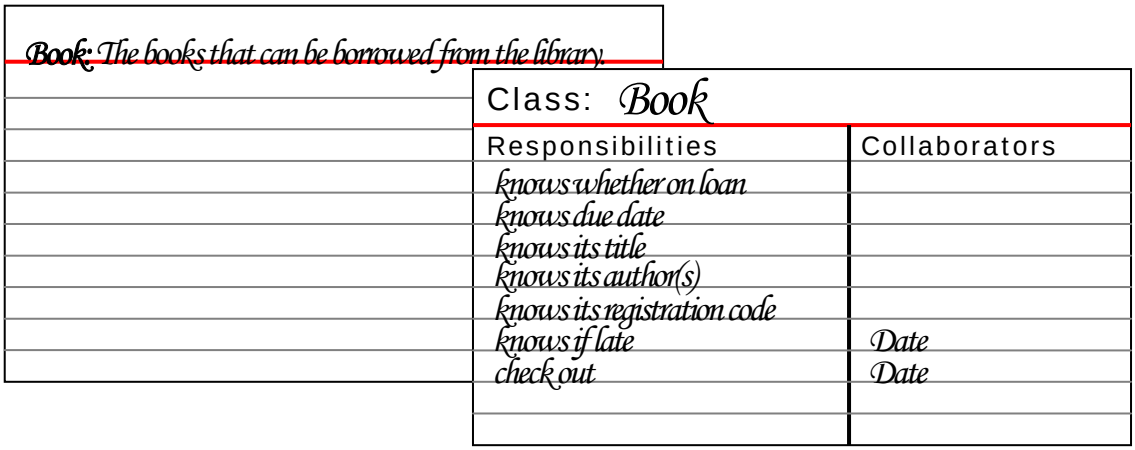


Figure 15: CRC-card for the `Book` class. Checking out a book needs `Date` as a collaborator to compute an appropriate return date.

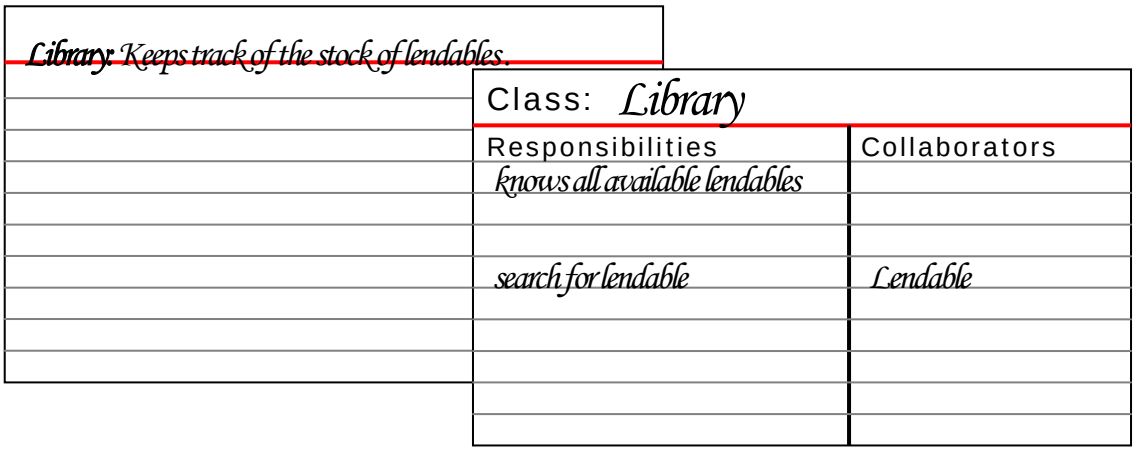


Figure 16: CRC-card for class `Library`. Searching needs `Lendable` as collaborator to compare the search information (for example a title) with information known by `Lendables`.

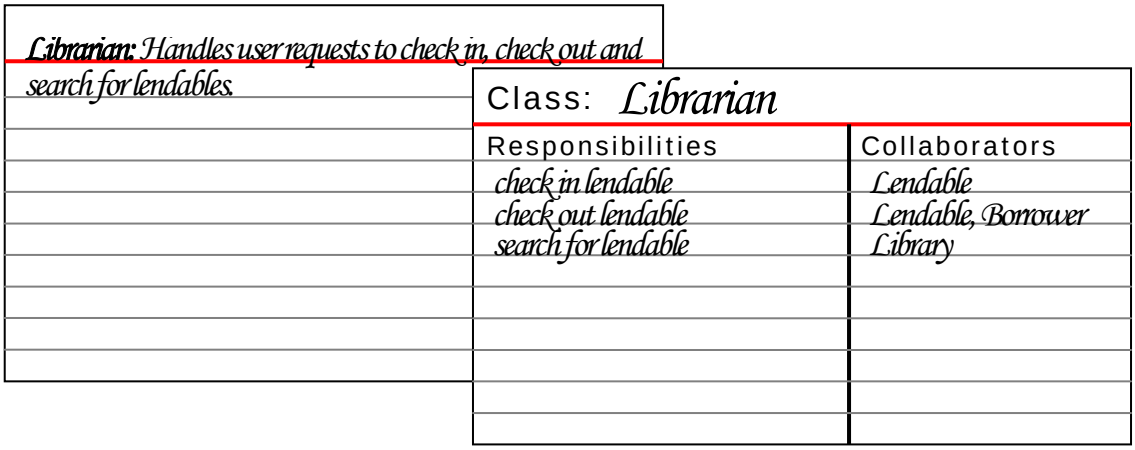


Figure 17: CRC-card for class `Librarian`.

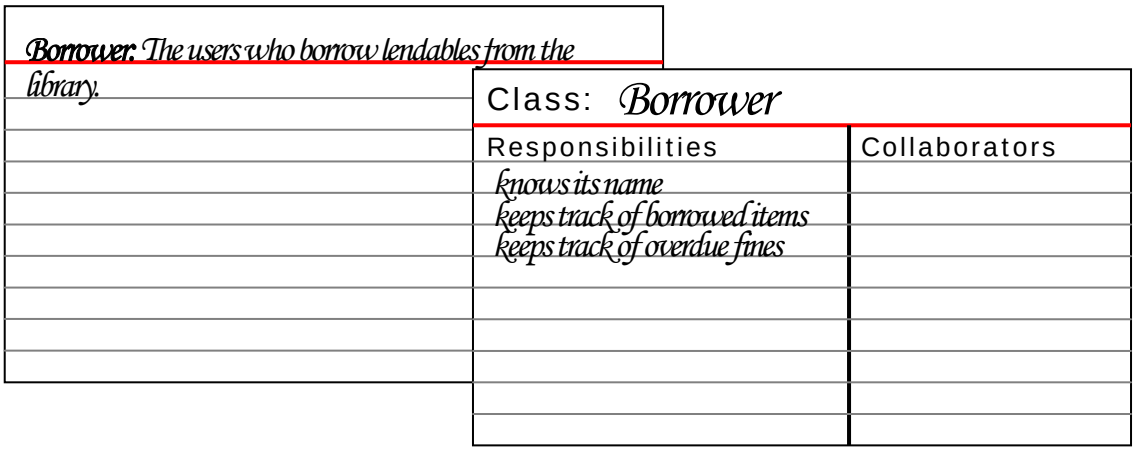


Figure 18: CRC-card for class `Borrower`.

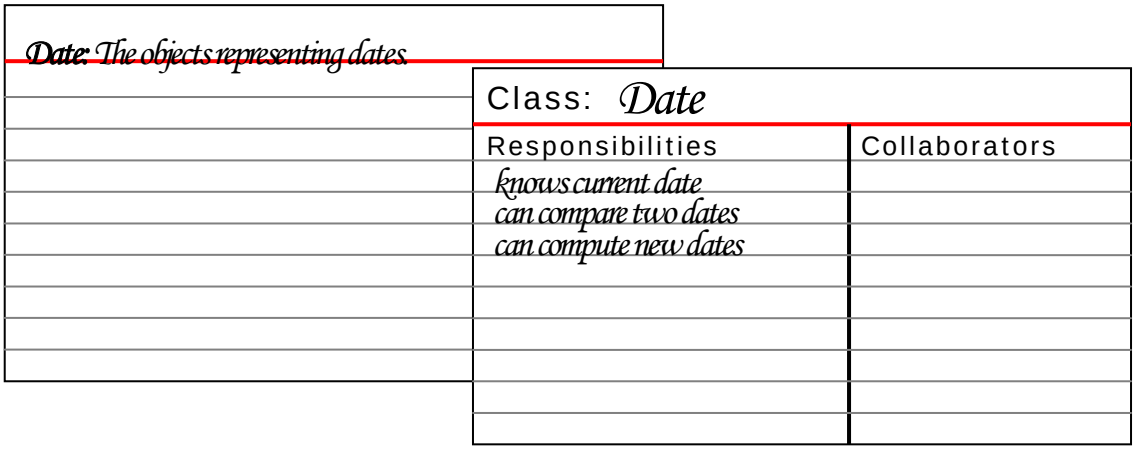


Figure 19: CRC-card for class `Date`.

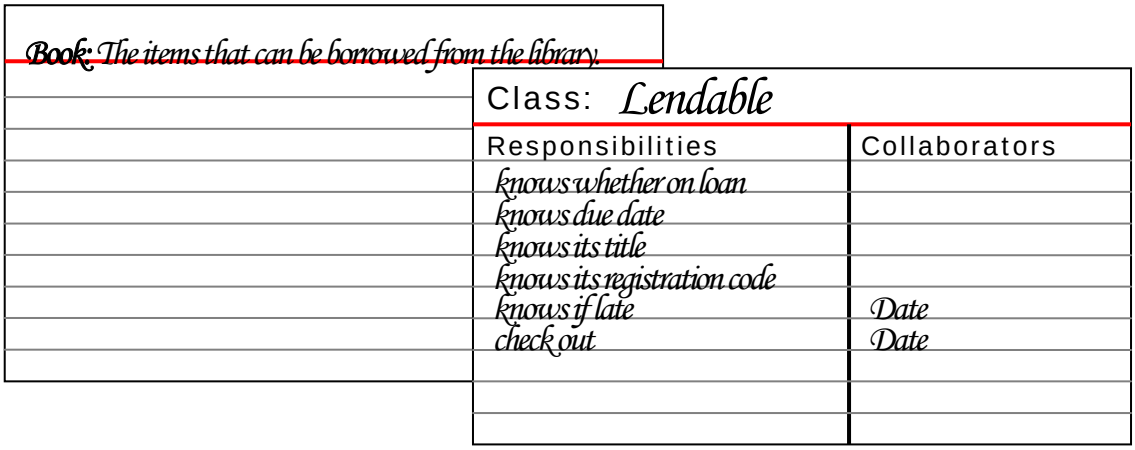


Figure 20: CRC-card for the general (super-)class `Lendable`. Specific types of library items would inherit from `Lendable` (see the following figures for examples).

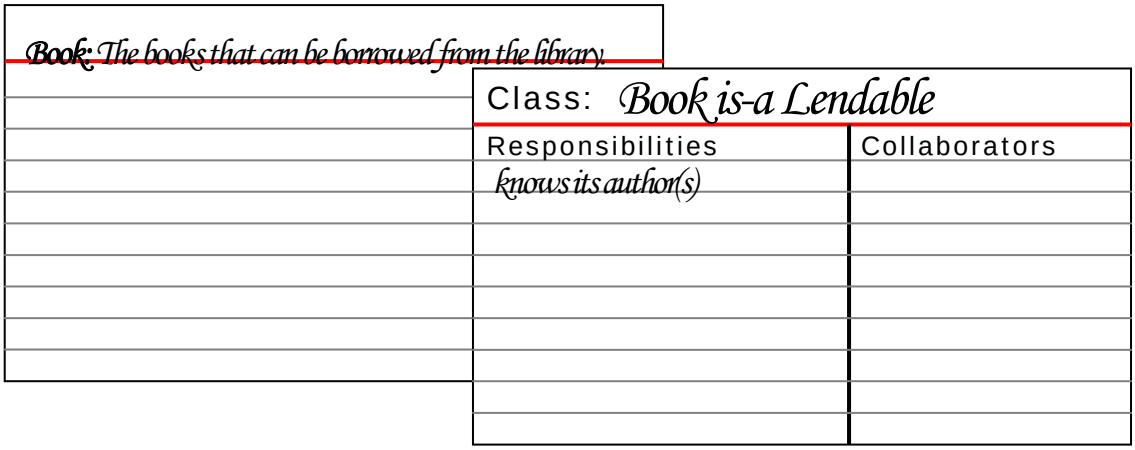


Figure 21: CRC-card for class `Book` as derived from superclass `Lendable`. The only thing a book differs from a general library item (of type `Lendable`) is that books have author(s).

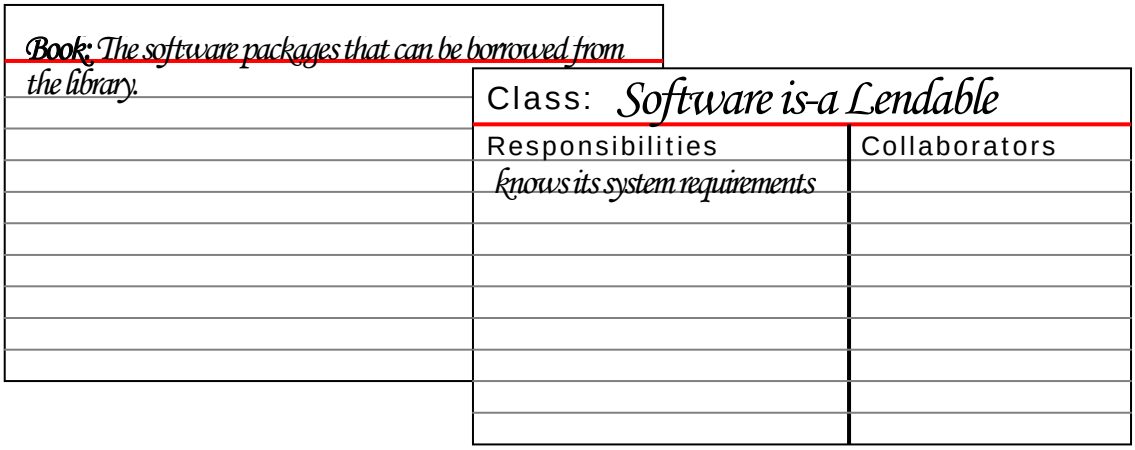


Figure 22: CRC-card for class `Software` as derived from superclass `Lendable`. The only thing software differs from a general library item (of type `Lendable`) is that software has descriptions of system requirements. The CRC-cards for `Video`, `Journal`, and `Music` would look similar.

8.4 Define Scenarios

The scenarios are the test cases of our system. As with test cases in general, we need a

situation we want to test plus specific test data to perform the actual test. We cannot start with too general a scenario, like "what happens when an employee will borrow a book". We would easily get lost in making decisions about irrelevant details. What would for example happen if the book is not available in the library, if it is on loan, if the employee already has borrowed 10 items, if the employee has overdue items, if …?

It would furthermore be impossible to define the results we expect from our scenario and it would therefore be difficult to decide whether the role-play actually was successful.

We will start with scenarios that cover the most basic and important functionality of the system, like checking out books. Furthermore, we will take the non-problematic cases first to establish a sufficiently stable working model, before going into the details. For our library system, we start with the following concrete scenario:

> *John Doe will borrow the book 1984; John Doe is a registered borrower in the system; currently he has not borrowed any items and has no outstanding fines. The book 1984 is available and not on loan. After borrowing, John Doe will be registered as borrower of 1984 and 1984 will have a valid return date.*

As a next step, we could take some variations of this scenario for example where 1984 is on loan or John Doe has borrowed 10 books already. Another important basic scenario is checking in books. Here we could use the following scenario:

> *John Doe returns the book 1984 on time; John Doe is a registered borrower in the system; currently he has no outstanding fines. After borrowing, John Doe will no longer be registered as borrower of 1984. 1984 will no longer be on loan.*

Even here, we would first take a few variations before continuing with scenarios for other functionality.

8.5 Prepare Group Session and Role-Play

We select a scribe from the team and distribute our five main CRC-cards (`Library`, `Librarian`, `Borrower`, `Book`, and `Date`, as described in section 8.3) among the other team members. We start with the scenario John Doe borrows 1984 as described in section 8.4. and evaluate the starting situation for the scenario. As a result we develop a corresponding role-play diagram (RPD) as described in section 6.7.

According to the definitions of our CRC-cards, we get the following starting situation when the library system starts up (see Figure 23): A GUI object, a librarian object, and the actual library comprising some library items. But where are the borrowers? So far, no class is responsible for keeping track of the borrowers. We have two classes that seem suitable for taking this responsibility; `Library` or `Librarian`. `Librarian` has so far no bookkeeping responsibilities. `Library` is responsible for keeping track of the books in the library. It therefore seems reasonable to add the responsibility to keep track of the borrowers. We add responsibilities `know all borrowers` and `search for borrower` (with collaborator `Borrower`) to the `Library` CRC-card.

For our role-play, it is sufficient to assume there are two books and two borrowers in the library. This gets us the starting RPD below (Figure 23). Please note that the `Book` objects do not know anything about the `Borrower` objects.

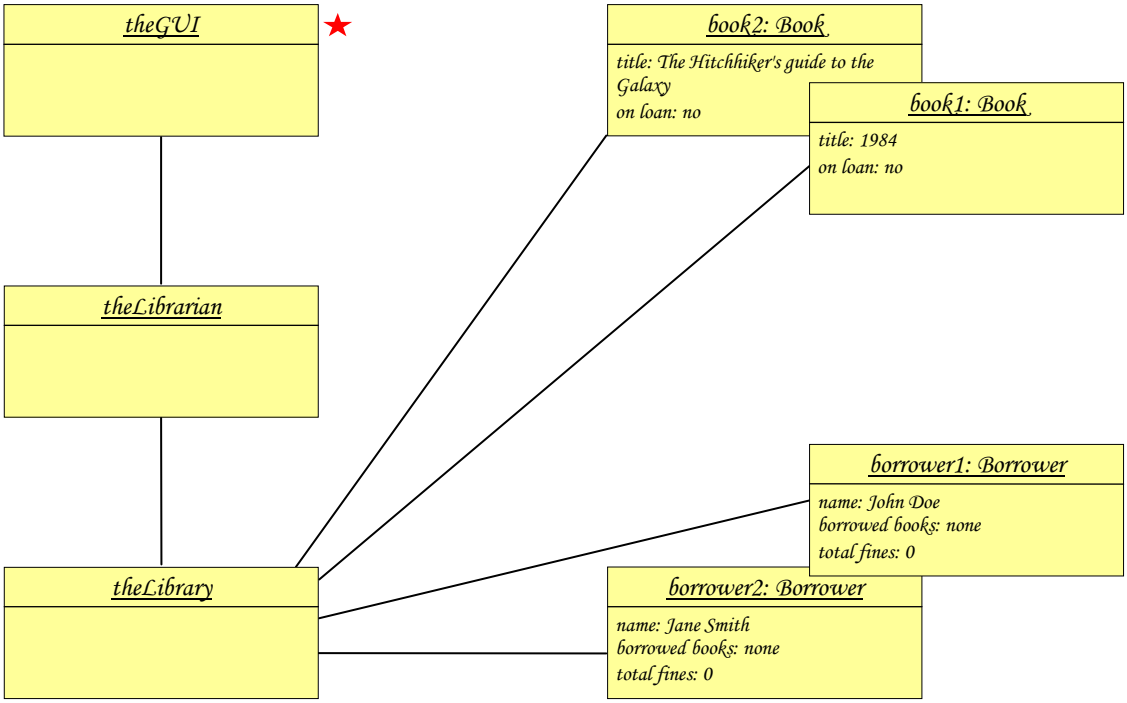


Figure 23: Starting RPD for scenario *John Doe will borrow the book 1984*. The currently active object is marked by a star. At scenario start, this is usually the GUI object.

To role-play a scenario means that we for each step in the scenario do the following:

- identify the currently active object and its current task or problem;
- identify the next (sub-) task or (sub-) problem;
- identify the object in the current RPD that has the responsibility to accomplish this (sub-) task or (sub-) problem;

  If there is no such object, check whether there is a CRC-card with a matching responsibility. If so, evaluate why a corresponding object card is missing and create it eventually. If not, add the required responsibility to a suitable CRC-card and update corresponding objects cards accordingly. Missing objects can be a sign of severe problems with your model and/or scenario. Sometimes these problems cannot be solved easily and it might not be possible to proceed in the role-play. In these cases you have to go back and revise your model(s) before continuing the role-play.
- check whether the object in question is known to the currently active object; if not announce it somehow, so that they can communicate;
- the currently active object requests the service (corresponding to the identified (sub-) task or (sub-) problem) and control is transferred to the object that provides the requested service;
- the team member responsible for the currently active object describes aloud (in first person) how the object provides this service;
- when the service is completed, control is transferred back to the requesting object.

8.6 Record the Actual Role-Play

The scenario *John Doe will borrow the book 1984* starts at `theGUI`, where someone presumably has input the name of a user and a book title this user wants to borrow. We

"translate" that into a request for checking out *1984* to *John Doe*.

In the remainder of this subsection, we present a transcript of a hypothetical but nevertheless realistic role-play for this scenario.

`theGUI`: I request from `theLibrarian` that the book with title "1984" is checked out to borrower "John Doe". I ask `theLibrarian`, since this is the only object I know.

> *Control is transferred*[16] *to `theLibrarian` object and the request is recorded in the RPD (see Figure 24).*

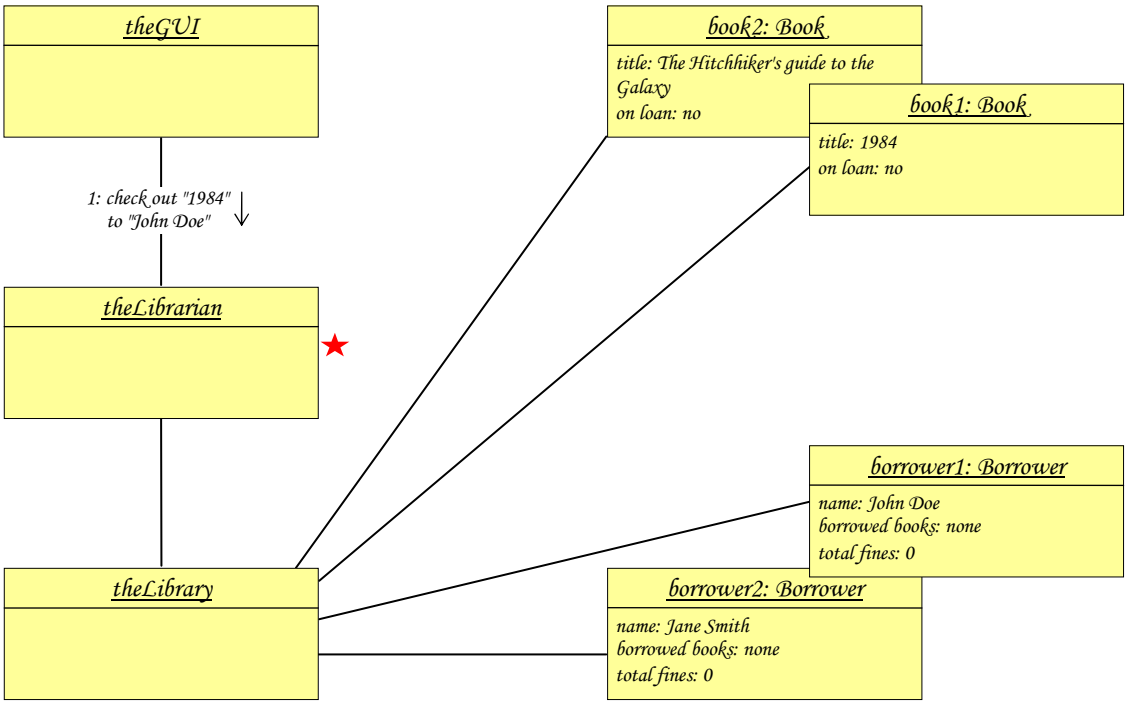


Figure 24: RPD from Figure 23 after the first request.

`theLibrarian`: That's fine with me. I have the responsibility `check out`. To check out a book to a borrower, I have to check whether the book is available and whether the borrower is allowed to borrow. First, I ask the book whether it is on loan.

> *Hold on … How do you ask the book? You do not even know whether the book is in the library stock. Furthermore, you can only ask the book if you "know it" (i.e. are connected to it in the actual RPD).*

`theLibrarian` continues: OK. However, I do not know anything about books in stock. My `check out` responsibility has collaborators `Lendable` and `Borrower`. However, I cannot ask them, since I do not know the actual objects yet. I should probably ask `theLibrary`, since `Library` is the class that keeps track of the stock.

> *The group discusses the problem. The `Library` CRC-card already has a suitable responsibility (`search for lendable`) and it is therefore reasonable to add `Library` as a collaborator to the `check out` responsibility of the `Librarian` CRC-card.*
>
> *The `Librarian` CRC-card is updated and the role-play is resumed.*

`theLibrarian` continues: I request from *`theLibrary`* object to `search for lendable` with the title "1984".

---

[16] Transferring control to another object means that the team member responsible for the CRC-card corresponding to that object continues the role-play (see section 6.7).

Control is transferred to `theLibrary` object and the request is recorded in the RPD. The team member responsible for the `Library` class/CRC-card continues the role-play.

`theLibrary`: I can do that. I have the responsibilities to `know all available lendables` and `search for lendable`. I check all lendables in stock whether they have the title "1984". I can do that, since `Lendable` is a collaborator for `search for lendable`. `Book1` is the object you look for.

Now the library has "introduced" `book1` to `theLibrarian` and `theLibrarian` can now communicate with `book1` directly. The scribe draws a line between object cards `book1` and `theLibrarian` in the RPD (see Figure 25). Control goes back to the requester (`theLibrarian`).

Please note that we did not discuss how the search is actually carried out. Otherwise, the role-play would become too complex. It is sufficient to note that it could easily be done by comparing titles of all known books (lendables).

`theLibrarian`: Now I can ask `book1` whether it is available.

The scribe records the request in the RPD and control is transferred to `book1`.

`book1`: I have a responsibility `knows whether on loan` and my object card currently says `no`.

The scribe records this communication in the RPD. Control goes back to the requester (`theLibrarian`).

`theLibrarian`: Now I can continue with the `check out`. I also need to check whether "John Doe" is allowed to borrow. As with the book, we first have to get the actual borrower object. I request from *`theLibrary`* object to `search for borrower` with the name "John Doe".

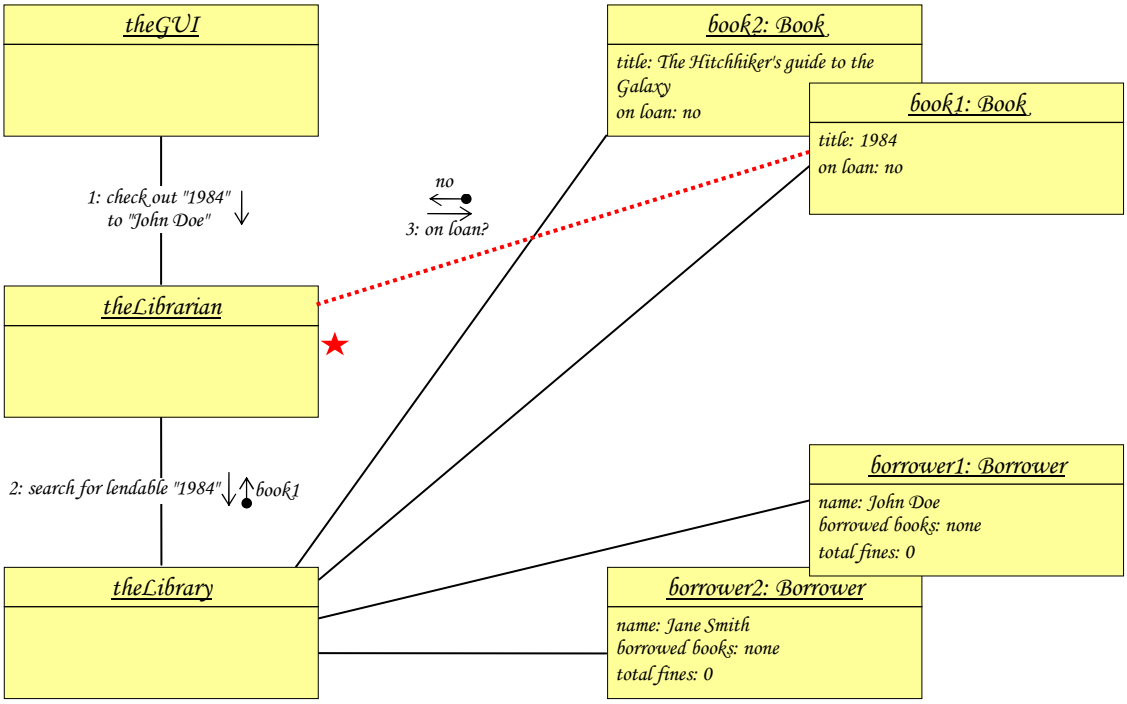


Figure 25: RPD after `book1` has answered `no` on `theLibrarian`'s request `on loan`.

Control is transferred to `theLibrary` object and the request is recorded in the RPD. Please remember that we updated CRC-card `Library` with responsibilities `know all borrowers` and `search for borrower` (see section 8.5).

theLibrary: Borrower1 is the object you look for.

The scribe draws a line between object cards borrower1 and theLibrarian in the RPD (see Figure 26). Control goes back to the requester (theLibrarian).

theLibrarian: Now I can check whether borrower1 is allowed to borrow. However, I don't know how. I do not know anything about the rules for borrowing. I ask borrower1 directly; are you allowed to borrow?

Control is transferred to the borrower1 object and the request is recorded in the RPD.

borrower1: I don't know. I know my borrowed items and my overdue fines, but I don't know the rules for borrowing.

The CRC team interrupts the role-play to discuss that problem. Who should have the responsibility to know or check whether a borrower is allowed to borrow? In the real world, this would be the librarian. However, in the software world the borrower itself keeps track of its borrowed items and overdue fines[17]. A borrower could therefore easily check whether it is allowed to borrow without any need to collaborate with other objects.

We add the responsibility checks whether allowed to borrow to the Borrower CRC-card.

The role-play is resumed.

borrower1 continues: I still do not know the rules for borrowing.

OK. OK. We add another responsibility knows rules for borrowing. How this is actually realised is an implementation detail that will be discussed in the design phase.

The role-play is resumed.

borrower1 continues: OK. My object card says that I have not borrowed any books and that I do not have any overdue fines. Therefore, I can answer yes. I am allowed to borrow.

Control is transferred back to the requester (theLibrarian) and the answer is recorded in the RPD.

theLibrarian: Now we can finally check out book1 to borrower1. First, I request book1 to check out.

Control is transferred to book1. The scribe records the request in the RPD.

book1: I actually have a responsibility check out. When I check out a book, I must also take care of my responsibility knows due date. However, I do not know how to compute dates. I do not even know how long I can be borrowed.

Hold on. The rules for computing return dates are specific for the type of Lendable. It is therefore reasonable to let the different types of Lendable be responsible for that.

We add a responsibility knows maximum borrowing time to Lendable (Book).

The role-play is resumed.

book1 continues: I request Date to create a new return date 6 weeks from now.

To create new objects it is sufficient to have the corresponding class as a collaborator. We can therefore create a new Date object and connect it to book1 (see step 7 in Figure 26).

Date: I can do that. I have responsibilities knows current date and compute new dates.

[17] Please note that in the software world a borrower cannot cheat. In the real world the librarian keeps own records about all borrowers' status. In the software world, these records are the actual borrower objects. There is no need for double bookkeeping.

The scribe adds a new object card `returnDate` to the RPD and connects it to `book1`.
The role-play is resumed with `book1`.

`book1` continues: Now I know that I am on loan and I know my due date.

The scribe changes `on loan:` on `book1`'s object card to `yes`.
Control returns to the requester (`theLibrarian`).

`theLibrarian`: as a final step, I request `borrower1` to remember that it has borrowed `book1`.

Control is transferred to `borrower1` and the request is recorded in the RPD. Please note that we also can draw a line between `borrower1` and `book1`, since theLibrarian has "announced" `book1` to `borrower1` (see Figure 26).

`borrower1`: I have the responsibility keeps track of borrowed items. Do I need a specific responsibility to actually add the book to my borrowed books?

We add responsibilities for adding and deleting items, respectively and resume the role-play.

`borrower1` continues: Now I `add item book1` to my borrowed books.

The scribe updates the RPD. Control is transferred back to the requester (`theLibrarian`).

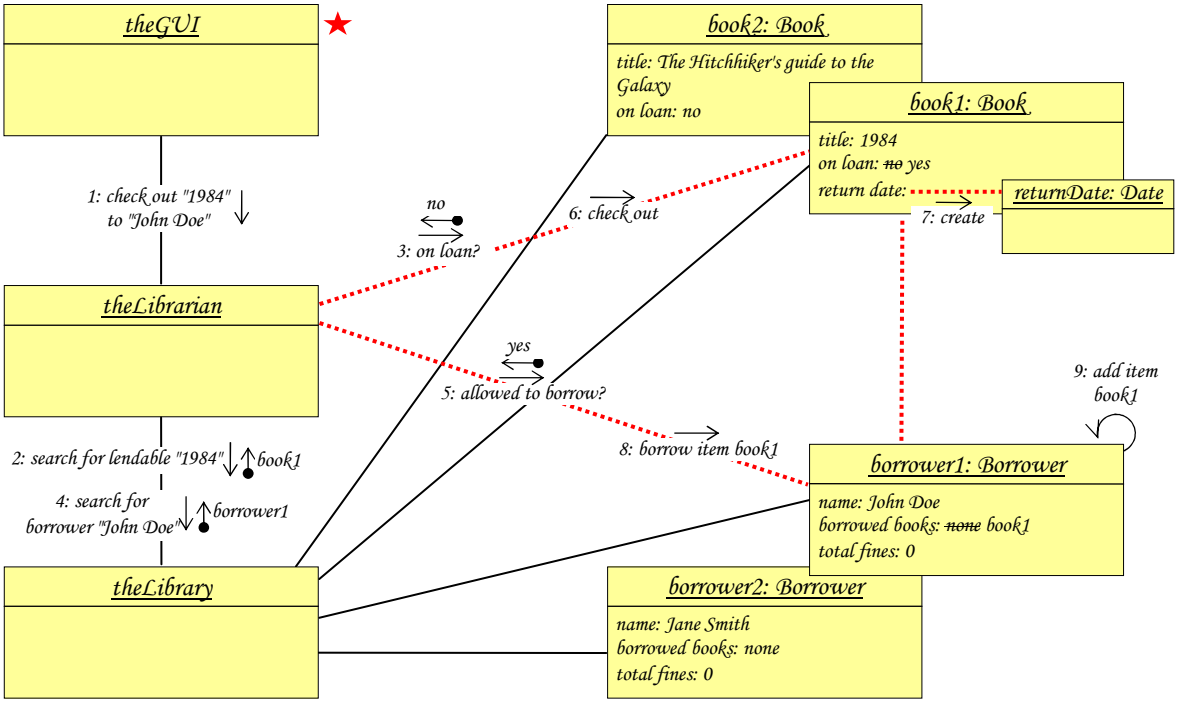


Figure 26: Final RPD for scenario *John Doe borrows book 1984*.

`theLibrarian`: OK. I'm done.

Control is transferred back to the requester (`theGUI`).

`theGUI`: Done.

We now have to check if the results are as expected. If so, the scenario has been successfully executed.

After completion of the role-play, we check whether the scenario has been correctly recorded and everything has been updated correctly. Before we continue with the next scenario, we should carefully document the scenario.

As pointed out earlier, there is no single correct way to model a system. In the role-play

above, `theLibrarian` is the most active part, since it "executes" most of its `check out` responsibility on its own. However, this could have been done in many different ways. For example, `theLibrarian` could have delegated most of its own `check out` communication with `book1` to `borrower1`. The only necessary change to our CRC-model would be to add `Book` (`Lendable`) as a collaborator for `Borrower`'s new `borrow item` responsibility. This alternative way to model, the system would have several advantages. The model would be less `Librarian`-centric than now and responsibilities would be distributed more evenly throughout the system. On the other hand, might this alternative be seen as less "natural" and therefore more difficult to understand.

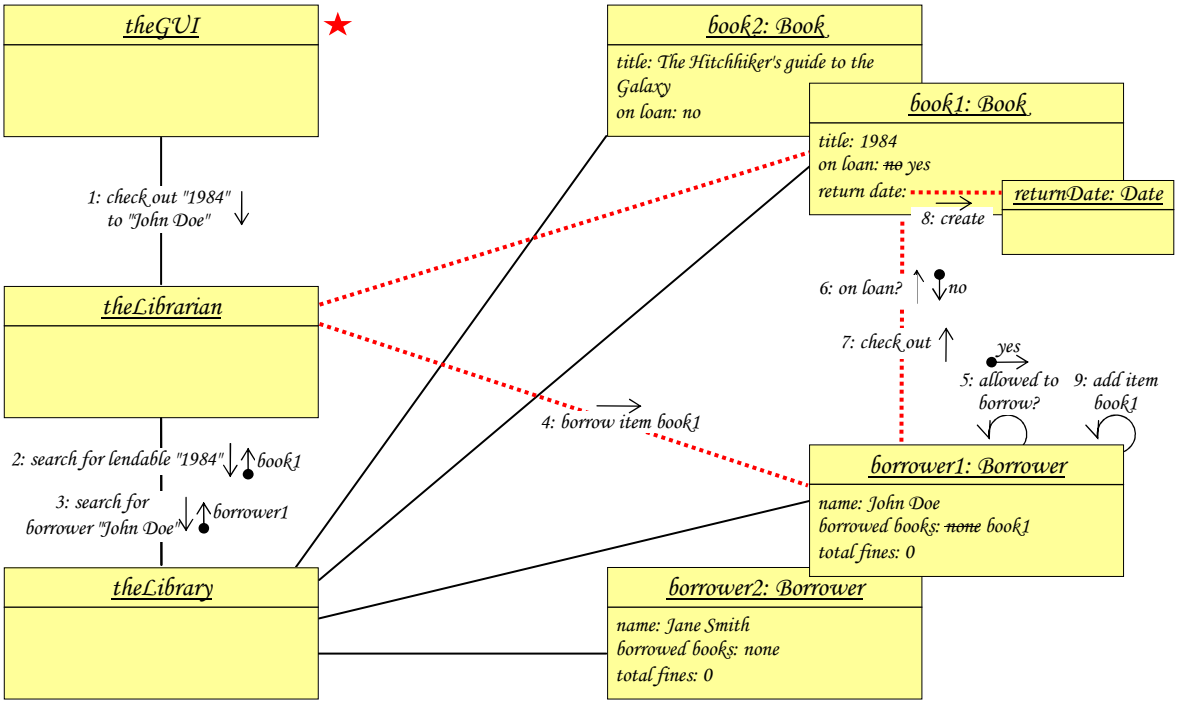


Figure 27: Final RPD for scenario *John Doe borrows book 1984* with alternative responsibility distribution as described above.

8.7 Resulting UML Diagrams

Our updated CRC-cards and the result from our scenario role-play can be described by the following UML diagrams. The sequence diagram in Figure 29 is consistent with the class diagram in Figure 28, except some method parameters.

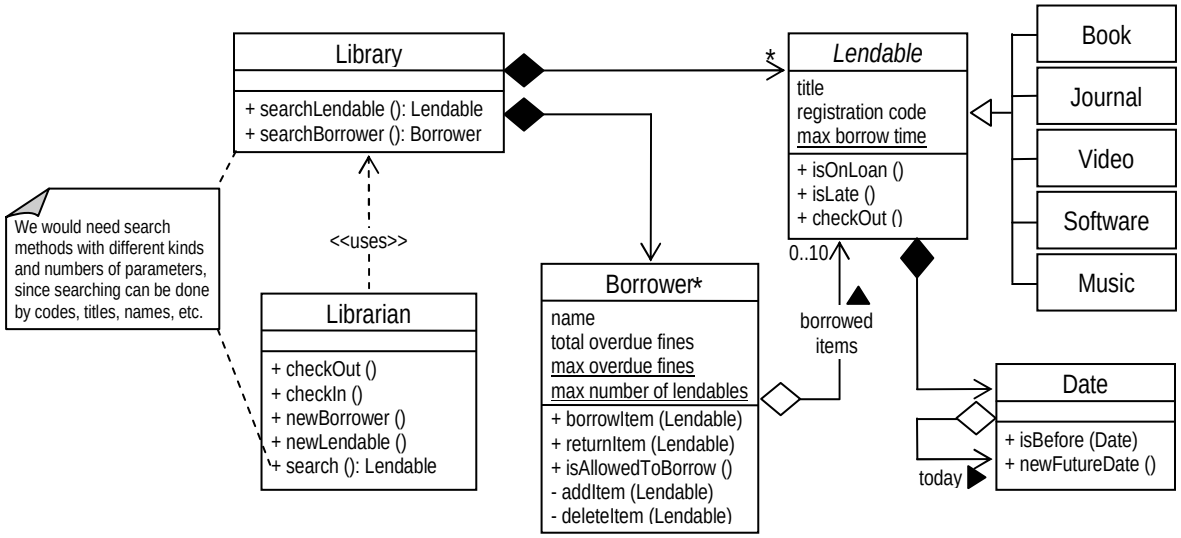


Figure 28: A UML class diagram for the example library system.

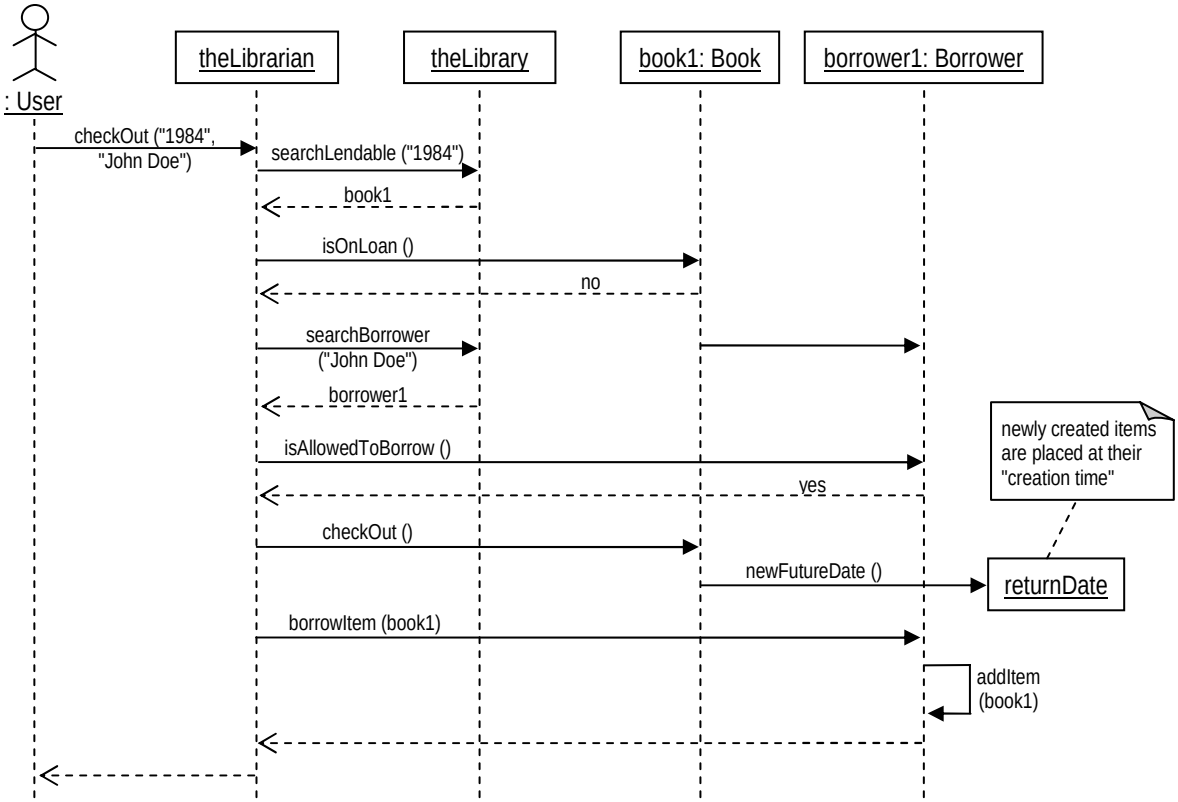


Figure 29: A UML sequence diagram corresponding to the RPD in Figure 26.

Acknowledgements

I thank Johan Eliasson, Lena Kallin Westin and particularly Marie Nordström for their feedback on RPDs and earlier versions of this report.

References

Beck, K., Cunningham, W. (1989). A Laboratory for Teaching Object-Oriented Thinking. *Proceedings OOPSLA'89*. 1-6.

Bellin D., Suchman Simeone, S. (1997). *The CRC Card Book*. Reading, MA: Addison-Wesley.

Biddle, R., Noble, J., Tempero, E. (2002). Reflections on CRC Cards and OO Design. *Proceedings TOOLS Pacific 2002*.

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications, 2nd Edition*. Redwood City, CA, USA: Benjamin/Cummings.

Booch, G., Rumbaugh, J., Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley.

Fowler, M. (2004). UML Distilled, Third Edition. Boston, MA: Addison-Wesley.

Lieberherr, K.J., Holland, I.M. (1989). Assuring Good Style for Object-Oriented Programs. *IEEE Software* **6** (5), 38-48.

Meyer, B. (1997). *Object-Oriented Software Construction*. Englewood Cliffs, NJ: Prentice Hall.

OMG (2003). UML 2.0 Superstructure Specification. OMG Adopted Specification, ptc/03-08-02. Object Management Group, available from http://www.omg.org/uml.

Wilkinson, N. (1995). *Using CRC Cards, An Informal Approach to Object-Oriented Development*. New York, NY: SIGS Publication.

Wirfs-Brock, R., Wilkerson, B., Wiener, L. (1990). *Designing Object-Oriented Software*. Upper Saddle River, NJ: Prentice Hall.

Wirfs-Brock, R., McKean, A. (2003). *Object Design--Roles, Responsibilities, and Collaborations*. Upper Saddle River, NJ: Prentice Hall.

Zamir, S. (editor) (1999). *Handbook of Object Technology*. Boca Raton, FL: CRC Press.