A Software Modelling Exercise Using FCA

Thomas Tilley[1], Wolfgang Hesse[2], and Roger Duke[1]

[1] The University of Queensland
Brisbane, Australia
{tilley, rduke}@itee.uq.edu.au
[2] University of Marburg
Department of Mathematics/Computer Science
Hans Meerwein-Str.
D-35032 Marburg, Germany
hesse@mathematik.uni-marburg.de

Abstract. This paper describes an exercise in object-oriented modelling where Formal Concept Analysis is applied to a formal specification case study using Object-Z. In particular, the informal description from the case study is treated as a set of use-cases from which candidate classes and objects are derived. The resulting class structure is contrasted with the existing Object-Z design and the two approaches are discussed.

1 Introduction: Identifying class candidates via use-case analysis

Formal Concept Analysis (FCA) is a lattice based ordering and visualisation technique that has already been applied in Software (Re-) Engineering [5, 6] and for identification of classes in Object-oriented modelling [2, 3]. This paper describes a software modelling exercise where FCA is applied to a formal specification case study using Object-Z. In particular, the informal description from the case study is treated as a set of use-cases from which candidate classes and objects are derived via FCA. The resulting class structure is contrasted with the existing Object-Z design and the two approaches are discussed.

Our aim was to perform a comparison between a class hierarchy derived via the application of FCA and an existing class diagram produced as part of an Object-Z case study [1]—in particular applying FCA in connection with use-case analysis to discover class candidates [3]. Moreover, the FCA class decomposition was performed sight unseen, that is, only the use-cases were presented to the class designers—they did not have access to an existing class diagram for the system being modelled. The informal description of the system was considered as a use-case source and five use-cases were identified. In particular a number of questions were asked:

– What are the differences between the two class hierarchies and are there valid reasons for the differences?
– What support does FCA offer the class designer and to what extent is it automated?

– How does the FCA approach influence the quality of the resulting class structure?
– Is FCA a useful mechanism for constructing Object-Z classes?

Currently, the "Object-Z engineer" works in a bottom-up manner, using mainly inheritance and association to create the system. The process is largely based on native experience, and a great deal of Object-Z "training" tries to cultivate this experience. Can FCA help by providing a method that relies less on training and previously acquired knowledge but results in the same or at least a similar class structure?

Our FCA-based methodology for identifying class candidates from a use case-like problem description works as follows (for details cf. [2] and [3]):

– (Re-)Structure the problem description and formulate use-cases.
– Mark all relevant "things" occurring in the use-case descriptions.
– Build a *formal context* (in the sense of FCA) by taking the marked "things" as *objects* and the use-cases as *attributes*.
– Generate the formal concept lattice for discussion. Check the concept nodes of the resulting lattice for being good class candidates.
– Discuss, rework and modify the use-case descriptions and the marking of objects.
– Iterate the preceding steps until a satisfactory class structure has evolved.

For the FCA terminology and details on formal contexts, concept lattices, etc. cf. [4].

The next section of the paper introduces this approach in more detail using a mass transit railway ticketing system as an example. Section 3 describes the progress from the initial informal description to the final concept lattice representing a possible class structure for our example case. Section 4 contrasts the results of the two approaches before Section 5 summarises the paper.

2 The mass transit case: From an informal description to a first concept lattice

Our starting point was an informal description taken from a case study modelling a mass transit railway ticketing system in Object-Z [1]. The main purpose of the case study was to capture the functionality of the different ticket types. The functionality was specified as perceived by an observer of the railway system. The informal description of the system from [1] reads as follows:

– The mass transit railway network consists of a set of stations. For simplicity, it will be assumed this set is fixed, i.e. stations are neither added to nor removed from the network.
– The fare for a trip depends only upon the stations where the passenger joins and leaves the network, i.e. the actual route taken by the passenger when in the network is irrelevant. The fare structure can be **updated** from time to time.

- Three types of tickets can be **purchased**:

  Single-trip tickets permit only a single trip, and only on the day the ticket is purchased. The ticket has a value in the range $1 to $10, and the passenger is permitted to leave the network if and only if fare for the trip just completed does not exceed the ticket's value.

  Multi-trip tickets are valid for any number of trips provided the current value of the ticket remains greater than zero. A ticket's initial value is either $50 or $100. Each time the passenger leaves the network the value of the ticket is reduced by the fare for the trip just completed. If this fare exceeds the value remaining on the ticket, the passenger is still permitted to leave the network and the value of the ticket is set to zero. A multi-trip ticket expires after two years even if it has some remaining value.

  Season tickets are valid for either a week, a month, or a year. Within that period no restrictions whatsoever are placed upon the trips that can be undertaken.
- As tickets are expensive to produce, they can be **reissued**, i.e. tickets can have their expiry date and value reset. (The type of ticket cannot be changed.) Although tickets are issued to passengers, the essential interaction is between tickets and stations; thus passengers are not modelled.

From the informal description five use-cases were identified: *update fare structure*, *buy single ticket*, *buy multi-trip ticket*, *buy season ticket*, and *reissue ticket*. In a first step, the text was cut into five pieces according to bold keywords representing the five "use-cases". All nouns showing a certain relevance were taken as "things", i.e. objects in the FCA sense. The choice of the nouns was deliberately done in a syntactical, "quasi-automated" way, i.e. without further semantic considerations whether this choice makes much sense. For every "thing", we can mark whether it is contained in a use-case description or not. The result is represented in Table 1. The column names represent the use-cases and the rows represent the nouns identifying objects in the use-cases [1]. A "×" at the intersection of a use-case and a noun indicates that the noun was identified in this use-case description. The corresponding formal concept lattice is shown in Figure 1.

A first correction concerns the cut of the text into use-cases where the headline introducing the three types of tickets was mistaken as a part of the *update fare structure* use-case. In fact, the noun *type of ticket* is not addressed in *update fare structure* but it is part of the introductory headline and thus applies to the following three use-cases describing the purchase of the three ticket types.

The initial changes between the context in Table 1 and Table 2 result from *type of ticket* being removed from the *update fare structure* use-case and added to

[1] In the standard FCA terminology the term "attribute" refers to the keyword identifying a column in a formal context and "object" labels identify a row. To avoid confusion the term "item" (referring to the nouns) is used here instead of "attribute".

	reissue ticket	update fare structure	buy single	buy multi	buy season
type of ticket	X	X			
expiry date	X				
value	X		X	X	
fare struct		X			
station		X			
fare		X	X		
network		X	X	X	
passenger		X	X	X	
ticket	X		X	X	X
trip		X	X	X	X
day			X		
single trip			X		
initial value				X	
multi-trip				X	
number				X	
value remaining				X	
month					X
period					X
week					X
year				X	X

Table 1. First Formal context created from the five use-cases. The corresponding concept lattice is shown in Figure 1.

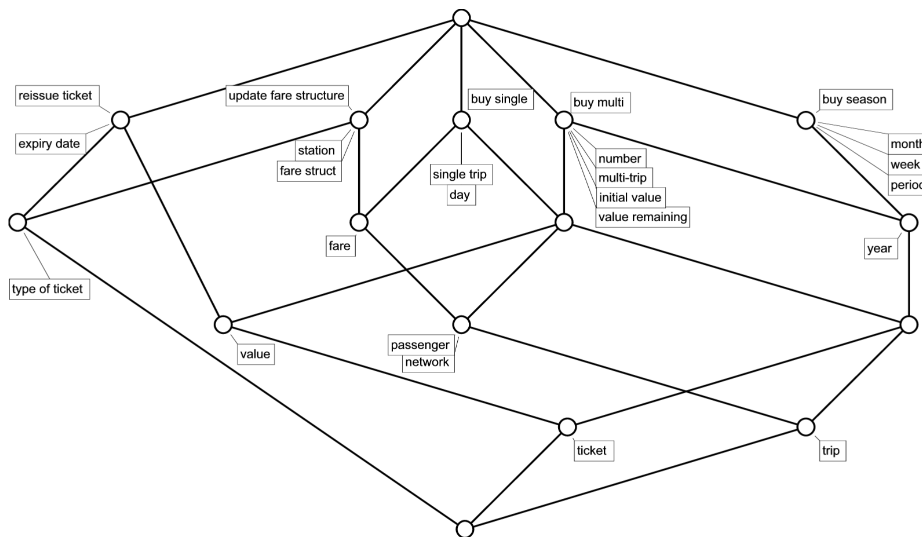

Fig. 1. The Formal Concept lattice for the context represented in Table 1.

	reissue ticket	update fare structure	buy single	buy multi	buy season
type of ticket	X		X	X	X
expiry date	X				
value	X		X	X	
fare struct		X			
station		X			
fare		X	X		
network		X	X	X	
passenger		X	X	X	
ticket	X		X	X	X
trip		X	X	X	X
day			X		
single trip			X		
initial value				X	
multi-trip				X	
number				X	
value remaining				X	
month					X
period					X
week					X
year				X	X
time		X			X

Table 2. Changes to the formal context shown in Table 1 are shown in grey. The corresponding concept lattice is shown in Figure 2.

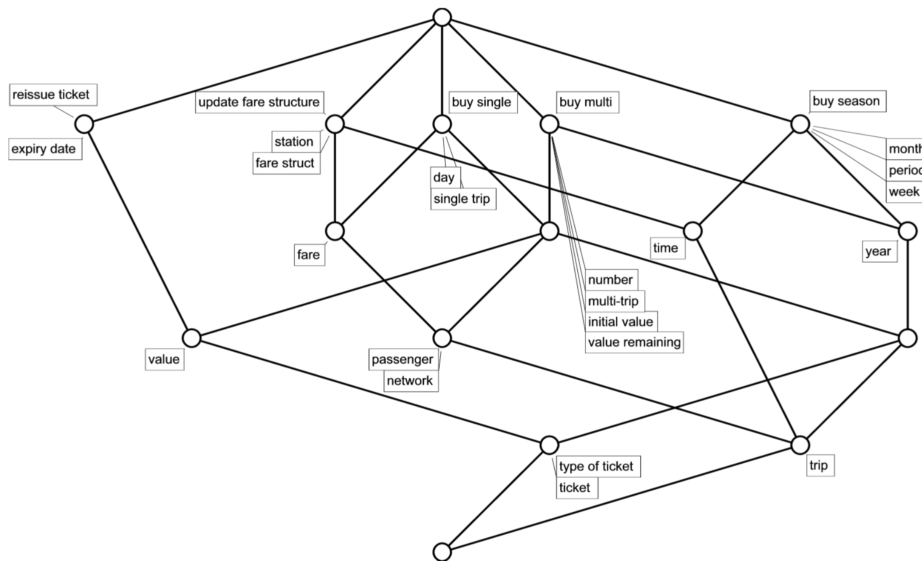

Fig. 2. The Formal Concept lattice for the context represented in Table 2.

the three "buy" use-cases: *buy single*, *buy multi* and *buy season*. The differences between the two contexts are shown in grey. Furthermore the item *time* has been identified and included in both the *update fare structure* and *buy season* use cases. Time is explicitly mentioned in *update fare structure* but not in *buy season ticket*. However, we have extended the text "Within that period . . . " by the implicit assumption "Within that period of *time*. . . ".

3 Iterating the FCA steps

The initial steps (identification of use-cases and contained things, corrections of the incidence matrix) resulted in the formal context and lattice diagram of Table 2/Figure 2. A first analysis of the lattice shows the use-case $\leftrightarrow$ thing dependencies as far as they can be derived from the pure syntactical formulation of the use-cases:

1. If we select a certain node marked by a use-case identifier, then we find all things occurring in this use-case somewhere among its successor nodes.
2. If we select a certain node marked by a "thing" identifier, then we find all use-cases containing this thing somewhere among its predecessor nodes.

An immediate consequence is: the higher things occur in the lattice diagram the more specialised they are—i.e. the lower-most things are the most general ones. A dual argument would apply to the use-cases if these formed a hierarchy (which is not the case in this example).

Further refinement of the structures now calls upon the "contextual knowledge" of the modeller/reviewer. From this point of view we start a first "semantic" analysis of the lattice: *passenger* and *network* seem to be too high in the diagram—we would expect them to be more general than the diagram shows. This inspires us to review the buy season ticket use-case: Although not mentioned in the corresponding use-case description, a season ticket implicitly involves both a *passenger* and the mass transit *network*. A more explicit version of the use-case description would read:

> **Season tickets** are valid *on the whole network* for either a week, a month, or a year. Within that period no restrictions whatsoever are placed upon the trips that can be undertaken *by the passenger*.

This modification is reflected in Table 3 and the corresponding lattice in Figure 3.

The context in Table 4 represents the recognition that the items *day* and *year* in *buy single ticket* and *buy multi-ticket* respectively also imply *time*. The resulting concept lattice is depicted in Figure 4 and at this point in the exercise the modellers were shown the existing class diagram of the system for the first time. An initial informal comparison was made and these observations are presented in Section 4 of the paper.

Looking at Figure 4 we detect that while the item *time* has now moved into an appropriate position, *fare* seems still too high in the diagram. This leads us

	reissue ticket	update fare structure	buy single	buy multi	buy season
type of ticket	X		X	X	X
expiry date	X				
value	X		X	X	
fare struct		X			
station		X			
fare		X	X		
network		X	X	X	X
passenger		X	X	X	X
ticket	X		X	X	X
trip		X	X	X	X
day			X		
single trip			X		
initial value				X	
multi-trip				X	
number				X	
value remaining				X	
month					X
period					X
week					X
year				X	X
time		X			X

Table 3. Changes to the formal context shown in Table 2 are shown in grey. The corresponding concept lattice is shown in Figure 3.

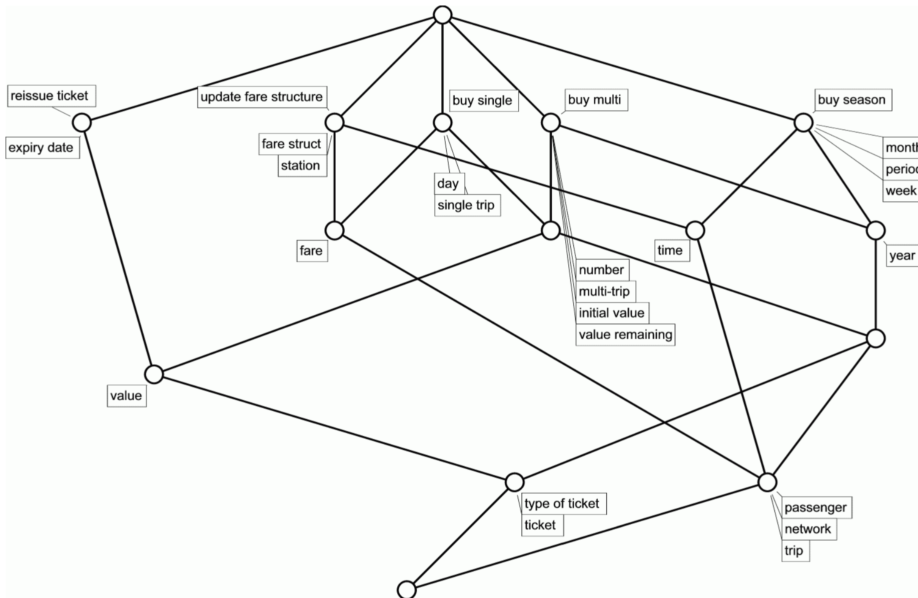

Fig. 3. The Formal Concept lattice for the context represented in Table 3.

	reissue ticket	update fare structure	buy single	buy multi	buy season
type of ticket	×		×	×	×
expiry date	×				
value	×		×	×	
fare struct		×			
station		×			
fare		×	×		
network		×	×	×	×
passenger		×	×	×	×
ticket	×		×	×	×
trip		×	×	×	×
day			×		
single trip			×		
initial value				×	
multi-trip				×	
number				×	
value remaining				×	
month					×
period					×
week					×
year				×	×
time		×	×	×	×

Table 4. Changes to the formal context shown in Table 3 are shown in grey. The corresponding concept lattice is shown in Figure 4.

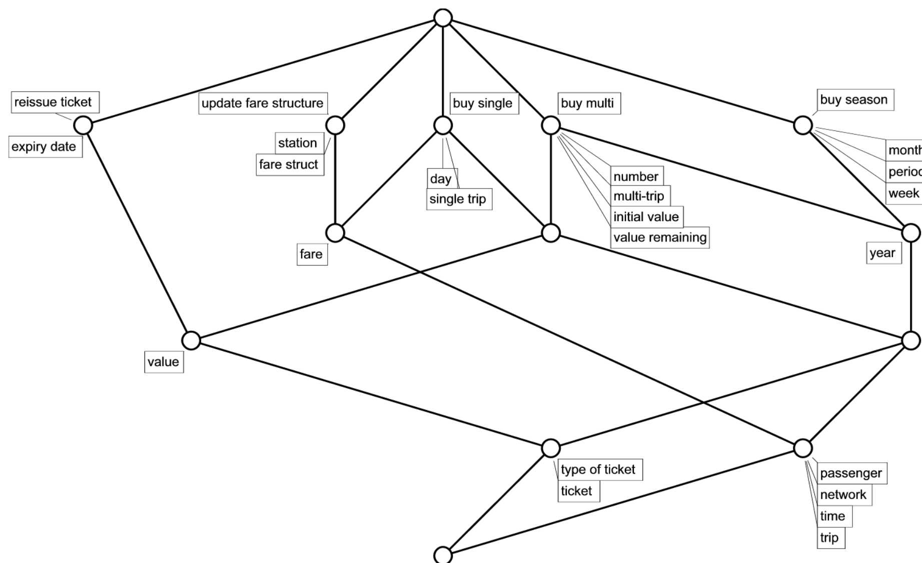

Fig. 4. The Formal Concept lattice for the context represented in Table 4.

to correct a mistake from our noun analysis of the use-cases: the item *fare* is mentioned in the *buy multi-ticket* use-case but was missed in the creation of the earlier contexts. This change is reflected in Table 5.

	reissue ticket	update fare structure	buy single	buy multi	buy season
type of ticket	X		X	X	X
expiry date	X				
value	X		X	X	
fare struct		X			
station		X	X	X	
fare		X	X	X	
network		X	X	X	X
passenger		X	X	X	X
ticket	X		X	X	X
trip		X	X	X	X
day			X		
single trip			X		
initial value				X	
multi-trip				X	
number				X	
value remaining				X	
month					X
period					X
week					X
year				X	X
time		X	X	X	X

Table 5. Changes to the formal context shown in Table 4 are shown in grey. The corresponding concept lattice is shown in Figure 5.

Furthermore, we consider the *station* item: the calculation of a *fare* implies knowledge of the stations by which a passenger enters and exits the mass transit railway network. Both the *buy single ticket* and *buy multi-ticket* use-cases include the *fare* item so in Table 5 the *station* item has been included for these use-cases as well. The concept lattice resulting from these "semantic implications" is depicted in Figure 5.

4 Comparing the two approaches

The aim of this modelling exercise was to perform a comparison between a class hierarchy derived via the application of FCA and an existing class diagram produced as part of an Object-Z case study. Having derived the lattice depicted in Figure 4 the modellers were shown the existing class diagram for the first time. One further refinement was made resulting in Figure 5. This section compares and contrasts the "final" lattice with the existing class diagram shown in Figure 6.

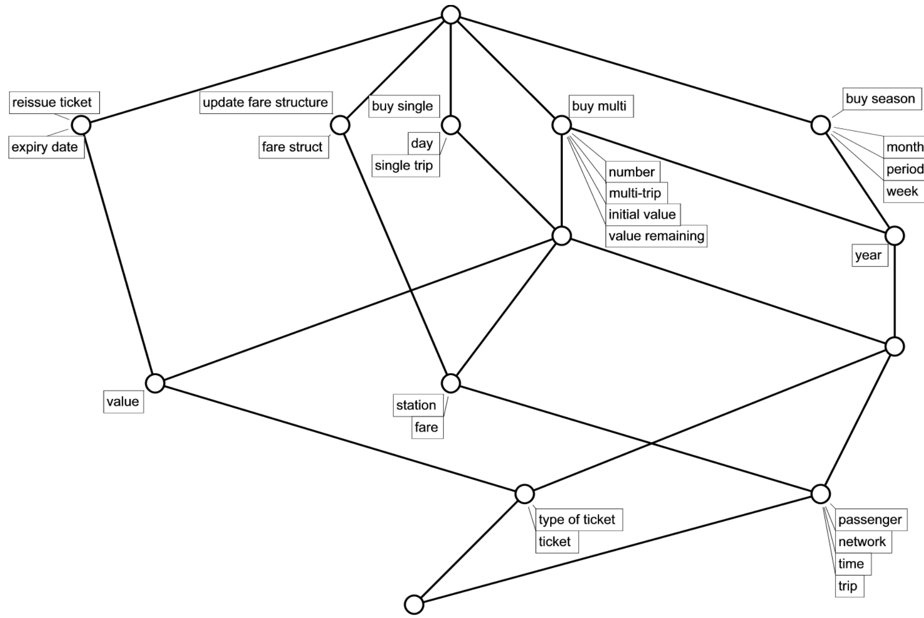

Fig. 5. The Formal Concept lattice for the context represented in Table 5.

If the node below the label "year" in Figure 5 is considered as a filter, that is, a point representing a common lower bound for any two nodes above it in the lattice, then the similarity between the line and class diagrams becomes more readily apparent. In Figure 7 this filter and the corresponding ideal (common upper bound) are shown in bold. The nodes labelled *buy single*, *buy multi*, and *buy season* in Figure 7 represent the class candidates corresponding to the classes *SingleTripTicket*, *MultiTripTicket*, and *SeasonTicket*. The unlabelled counterparts of the *TripTicket* and *Ticket* class unions in Figure 6 have also been labelled. In this case, the structure in bold represents the encapsulation of "ticket buying" functionality. An alternative interpretation that considers the re-issuing of tickets would move the corresponding *Ticket* label down to the *type of ticket* and *ticket* node.

An obvious difference between the two structures is the presence of attributes and possible methods (e.g. *update fare structure*) in the line diagram as compared with Object-Z functions in the class diagram. However, the relationship between the two structures can still be inferred by checking if the attributes required for a particular function are in the "correct" place. For example, the Object-Z representation makes use of *EnterStation* and *ExitStation* functions so that the appropriate fare can be calculated and checked for SingleTrip and MultiTrip tickets. The action of entering and exiting stations is assumed domain knowledge and is therefore not present in the use-cases. While the actions themselves do not appear in the line diagram the lattice mirrors the required structure because

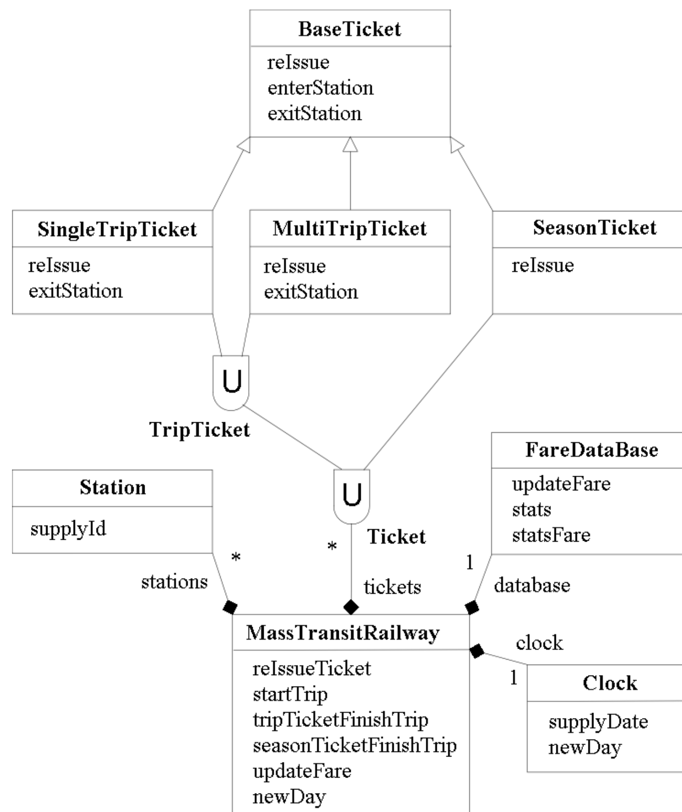

Fig. 6. Class union diagram for the mass transit system taken from [1].

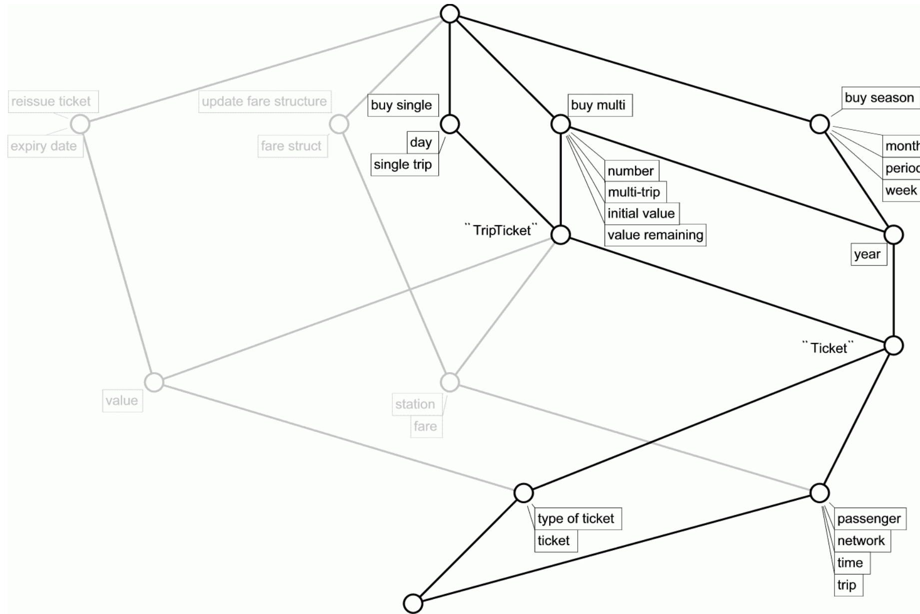

Fig. 7. The Formal Concept lattice from Figure 5 with the ticket class hierarchy shown in bold. The nodes labelled *TripTicket* and *Ticket* correspond to the class unions in Figure 6.

the *station*, *fare*, and *value* attributes are only available to these ticket types. This information is not required for a SeasonTicket.

Other differences between the two structures include the absence of obvious *Station*, *Clock* and *FareDataBase* classes in Figure 5. In addition, a comparison shows that the *stats* and *statsFare* functions in Figure 6 are quite artificial. These differences are largely due to functional artefacts or abstractions required for the Object-Z specification of the mass transit railway as described in [1]:

> The main purpose of this case study is to capture the functionality of the different ticket types. The approach taken is to specify ticket functionality from the point of view of the passenger, i.e. as perceived by an observer of the railway system. In order to do this, however, it is necessary to conceptualise and abstract various other objects in the system, such as the stations, a database to record the fare structure, and a clock to keep track of the days.

A further important question concerns the modularisation of the system, i.e. its decomposition into smaller units typically called *components*, *packages* or *modules*. Lindig and Snelting have shown that FCA can support this decomposition by forming so-called *block relations* [5]. Block relations result from filling up a formal context table with additional marks (not contained in the original context) in order to coarsen the lattice structure and get more compact concepts. In our case, the attempt to find an appropriate decomposition for the lattice of Figure 4 resulted in the initial package structure depicted in Figure 8. Three possible packages deal with the purchase of (various kinds of) tickets, the fare structure and its updates, and the re-issuing of tickets.

5 Conclusion

This paper has presented a modelling exercise to identify class candidates using use-case analysis and formal concept analysis and then contrast it against a known existing structure. A small, well understood example was chosen and a comparison of the resulting structures demonstrates that they are quite similar. Obvious differences between the two structures rely on information that is not made explicit in the use-cases or they represent artificial constructions related to the specification in Object-Z.

Although it may be possible to automate the initial noun identification within use-cases later refinements rely on the insight and judgement of the modeller. The value of this approach then is in the process itself—the construction and discussion of the line diagrams, and in the kinds of questions it forces the designer to ask about the structure. The process and resulting diagrams also promote discussion as modellers consider and question the position of attributes in the line diagram and try to adjust the formal context accordingly.

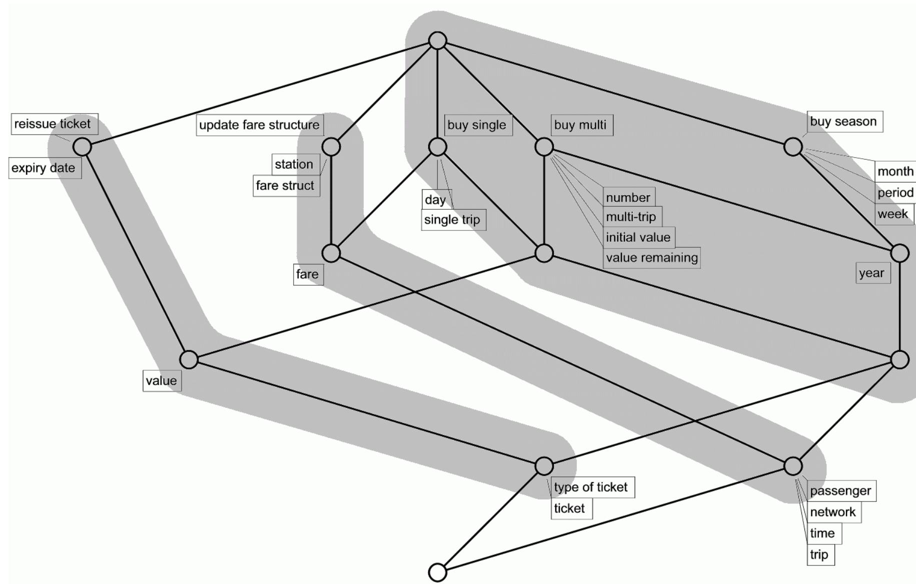

Fig. 8. Initial package structure based on Figure 4.

References

1. R. Duke and G. Rose. *Formal Object-Oriented Specification Using Object-Z*. MacMillan Press, 2000.
2. S. Düwel. *BASE - ein begriffsbasiertes Analyseverfahren für die Software-Entwicklung*. PhD thesis, Philipps-Universität, Marburg, 2000. Available through University library Marburg: `http://www.ub.uni-marburg.de/digibib/ediss/welcome.html`.
3. S. Düwel and W. Hesse. Bridging the gap between use case analysis and class structure design by formal concept analysis. In J. Ebert and U. Frank, editors, *Proceedings of Modellierung 2000*, pages 27–40. Fölbach-Verlag, 2000.
4. B. Ganter and R. Wille. *Formal Concept Analysis - Mathematical Foundations*. Springer-Verlag, Berlin, 1999.
5. C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the International Conference on Software Engineering (ICSE 97)*, pages 349–359, Boston, 1997.
6. G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of ACMSIGSOFT Symposium on the Foundations of Software Engineering*, pages 99–110, November 1998.