Bridging the gap between Use Case Analysis and Class Structure Design by Formal Concept Analysis

Stephan Düwel & Wolfgang Hesse

Abstract

The early stages of software development are increasingly supported by *object-oriented analysis and design (OOA/OOD)* techniques. Recent methodologies suggest to combine Jacobson's *use case analysis* with modelling techniques for *class/object structure, object behaviour* and *process interaction*. However, the choice and definition of objects and classes in the domain space is not supported by either of these techniques but left to the intuition of the analyst(s). *Formal Concept Analysis (FCA)* is a mathematical theory which offers support for analysing and visualising conceptual relationships by lattice diagrams. It is shown how FCA can successfully be applied to bridge the gap between the mentioned techniques: It starts with an analysis of the *formal context* given by use cases and the relevant "things" involved in these cases. It produces a lattice visualised by a *line diagram* which is used as a design and decision aid for building an appropriate class/object structure. This structure is a prerequisite for further modelling steps, e.g. modelling of processes by sequence diagrams.

The article comprises an outline of the covered OOA and OOD steps, a short introduction to the basic notions of FCA and a presentation of our approach including a demonstration example.

1 Introduction: Motivation and goals

During the last decade, a large variety of *object-oriented analysis and design (OOA/OOD)* techniques has been developed, supported by notations and tools and been applied in academic and industrial software development projects. Traditional techniques were typically based on two or three relatively independent models which represented different views of the system to be developed: a data view, a functional view and – if treated separately – a process or behavioural view. Designing and constructing a uniform system from such heterogeneous models often resulted in severe re-structuring problems and sometimes even in project failures.

Object-oriented software development techniques – and particularly the OOA techniques – promised a solution for this unsatisfying situation. They all share the paradigm of an encompassing, persistent *class structure model* which is built during the analysis stages and (with some modifications and enhancements) maintained through the whole system life cycle often including its operational stages. This model is intended to "map" or "mirror" the relevant ingredients of the application domain including their relationships and behaviour. Various methodologies following this paradigm have been published and experienced including those of Booch [Booc 94], Jacobson [JCJ+ 92], Shlaer and Mellor [ShMe 91], Coad and Yourdon [CoYo 90], Rumbaugh [RBP+ 91], Martin and Odell [MaOd 92]. At the moment, a certain convergence of these methods and techniques can be expected from the evolving Unified Modeling Language [BJR 97] marketed by Rational Company.

Integrating data and functional aspects in a joint class structure model serving both analysis and design purposes is a big step towards a fully-supported object-oriented development process. However, such a model is still incomplete with several respects:

- it does not cover important aspects like process modelling in-the-large or object behaviour over time,
- it leaves the fundamental question how to define and delimit classes and objects to the intuition of the analyst(s).

Most OO methodologies have answered the first open question by including additional notations and diagram types for modelling the uncovered aspects. Among these are *use cases* and *use case diagrams* introduced by Jacobson, *interaction, event trace* or *sequence diagrams* used by Jacobson, Rumbaugh and in UML and *state charts* adopted from D. Harel for UML. In fig. 1 we have listed the considered methodologies (including UML) and the diagram techniques they recommend.

	Use case diagram	Class diagram	Sequence diagram	Collaboration diagram	Statechart diagram	Activity diagram	Component diagram	Deployment diagram	Timing diagram	Data flow diagram	Object diagram	State transition graph	Fence diagram	Domain chart
UML	X	X	X	X	X	X	X	X						
Booch		X		X	X		X	X	X					
Coad/Yourdon		X			X					X				
Jacobson	X	X	X		X						X	X		
Martin/Odell		X		X	X	X							X	
Rumbaugh		X	X	X	X					X				
Shlaer/Mellor		X		X	X		X			X				X

Fig. 1: OO methodologies and their techniques

In particular, use cases are now becoming very popular as a means for analysing sample processes in the application domain. Of course, such an analysis normally yields valuable insights and results also concerning the second of the above mentioned questions (how to find the class and object structure). But it does not offer any systematic procedure or formalism to support this step which we consider crucial for the whole further development process. Objects and classes fall from heaven or “are there just for the picking” [Meye 88] to form the basis for most subsequent modelling steps.

We believe that this important step deserves more attention and more formal support which has to focus on a *conceptual analysis of the application domain*. We agree with authors like Booch, Martin/Odell or Rumbaugh who start with ‘concepts’ when writing about object oriented software development. This is a strong argument for using *Formal Concept Analysis (FCA)* in the class and object definition steps of object-oriented analysis. FCA is a mathematical theory which offers support for analysing and visualising conceptual relationships by lattice diagrams.

We propose to use FCA in order to bridge the gap between use case analysis and the following steps which imply the determination of a class/object structure and the production of state charts or sequence diagrams. Our technique starts with an analysis of the *formal context* given by use cases and the relevant "things" involved in these cases. It produces a lattice visualised by a *line diagram* which is to be used as a design and decision aid for building an appropriate class/object structure. This structure can then be used as a stable basis for the further indicated analysis and modelling steps.

In the following sections, we present our approach in more detail. First, we closer examine the covered OOA and OOD steps and the contributions of some well-known methodologies to support these steps (section 2). Then the basic notions of FCA are introduced and exemplified (section 3). In the following section (no. 4) the application of FCA to an example is demonstrated. It is then shown how the insights and results of such an analysis can be used to build and refine a class structure model for the given domain. The article ends with an outlook and general conclusions (section 5).

2 Object-oriented system analysis: From use cases to class and object structures

In the early nineties, I. Jacobson has introduced *use case analysis* as a starting step primarily to be performed in the early stages of system or domain analysis. According to Jacobson, a *use case* is a "special, behaviorally related sequence of transactions ... a user will perform .. in a dialogue with the system" [JCJ+ 92, p. 127]. The resulting *use case model* is a part of the *requirement model* and forms the basis for most subsequent models including the *analysis model*, the *design model* and the *implementation model* [JCJ+ 92, p. 123]. Later Jacobson states that "this transformation from use cases into objects forms one of the most important parts" of his methodology [JCJ+ 92, p. 142]. It is done by a "partition of functionalities". But there is no systematic procedure or guideline how to derive the objects or classes from use cases. "... Most entity objects are found early and are obvious. These 'obvious' entity objects are often identified in the problem domain object model. Others can be harder to find. Entities usually correspond to some concept in real life, outside the system, although this is not always the case. ..." [JCJ+ 92, p. 184]. We agree that the indicated transformation is very important but we do not believe that the given support is adequate.

In an earlier paper ([DüHe 98]), we have reported on a case study which aimed to determine class and object candidates from use cases by means of Formal Concept Analysis. In principle, it turned out that all resulting concepts are candidates for classes or class components (attributes or operations) from which the analyst has to choose his or her classes and to decide which ones to treat as classes or as class components. In this paper, we are revisiting the same example but we want to shift the focus to the different perspectives the analyst has to adopt during the early analysis stages and what he or she can learn from reading FCA line diagrams from different sides to support his or her choices and decisions.

We find it quite an interesting observation that just a "non-OO" technique like use case analysis has become so popular as a starting point to OO analysis. It shows that different perspectives are vital for a thorough analysis and that practicable methods have to involve "perspective changes", for example, from a functional view (represented by use case diagrams) to a data view (represented by class structure diagrams).

As a first conclusion, we can sum up some observations on use case-based methodologies:

- Use cases are a natural entry point to system analysis since they help to concentrate on the goals and the functional aspects of the application system.

- The step from a use case to a class structure model implies a change of perspective from a functional to a data view (not unlike the one implied in some earlier "structured" techniques).

We propose to postpone the decision about the class candidates until a better insight – based on both the functional and the data view – is achieved. In a first step we examine the full system functionality by use cases and look for involved "things". These "things" are candidates for entity objects or their attributes. This data-centred view will be complemented by functionally decomposing the uses cases down to the level of operations which then can be assigned to classes. These steps are supported by FCA and in particular by the resulting line diagrams which illuminate the obtained analysis results.

3 Formal Concept Analysis: a short introduction

Formal Concept Analysis (FCA) starts with a set G of *formal things* (German: Gegenstände) and a set M of *formal features* (German: Merkmale). In most original papers on FCA usually formal things and formal features are called (formal) objects and (formal) attributes. We have chosen the above terms to avoid any mix-up with OO terminology. Sometimes we will use the term "thing" (without prefix "formal") for denoting "real world things". Formal things and formal features are connected by a binary relation $I \subseteq G \times M$, called incidence relation. This relation indicates whether a formal thing has a formal feature. The triple (G,M,I) is called a *formal context*. It can be visualised in a table. Table 1 gives an example of a formal context. In this example the authors and UML are the formal things, their proposed notation techniques are the formal features and the relation is given by the marks in the table.

From a formal context, formal concepts are built: For every set $A \subseteq G$ of formal things $A^I = \{ m \in M \mid (a,m) \in I \ \ \forall a \in A \}$is the set of their common formal features and for $B \subseteq M$ we have $B^I = \{ g \in G \mid (g,b) \in I \ \ \forall b \in B \}$ as the set of common formal things. A pair (A,B) with $A \subseteq G$, $B \subseteq M$ is called a *formal concept* if $A = B^I$ and $B = A^I$.

If (A,B) is a formal concept, then A is called the *extent* of (A,B) and B the *intent* of (A,B). The extent comprises all formal things that belong to the formal concept and the intent consists of all formal features that the formal things of the formal concept share. This is just a formalisation of how a concept is viewed in philosophy and how it is understood – among others – by Martin/Odell (cf. [MaOd 92]).

On the set of all formal concepts of a formal context we consider the *sub-/super-concept relation* ≤: A formal concept (A,B) is called a sub-concept of another formal concept (C,D) – denoted $(A,B) \leq (C,D)$ – if the condition $A \subseteq C$ holds (the condition $D \subseteq B$ is equivalent). With this ordering relation, the set of all formal concepts of a formal context forms a complete lattice called the *concept lattice* B(G,M,I) (cf. [GaWi 98]).

As a finite ordered set, a finite concept lattice can be visualised by a *line diagram*. Fig. 2 shows the line diagram corresponding to the formal context in fig. 1. The nodes in the line diagram represent the formal concepts. The sub-/super-concept ordering relation is directly visualised by the edges: Each node is connected to its lower neighbours (with respect to the sub-/super-concept order). Thus the paths descending from a node of a formal concept lead to all nodes representing sub-concepts. For this reason the line diagram is a directed acyclic graph, where the direction of the edges is represented by the vertical arrangement of the start and end vertices.

For every formal thing g∈G there is a formal concept $\gamma g := (\{g\}^{II},\{g\}^{I})$ and dually for every formal feature m∈M $\mu m := (\{m\}^{I},\{m\}^{II})$ is a formal concept. γg is the minimal formal concept containing g in its extent and dually μm is the greatest formal concept comprising m in its intent. This fact allows an abbreviated notation for line diagram. A formal thing g is only annotated beneath the node representing γg. A formal feature m is only marked above the node representing μm. The whole information of the formal context is preserved in the line diagram. The formal features belonging to a formal thing g are found by following all ascending paths beginning at the node marked "g". To find all formal things characterised by a given formal feature, one has to proceed analogously on all descending paths.

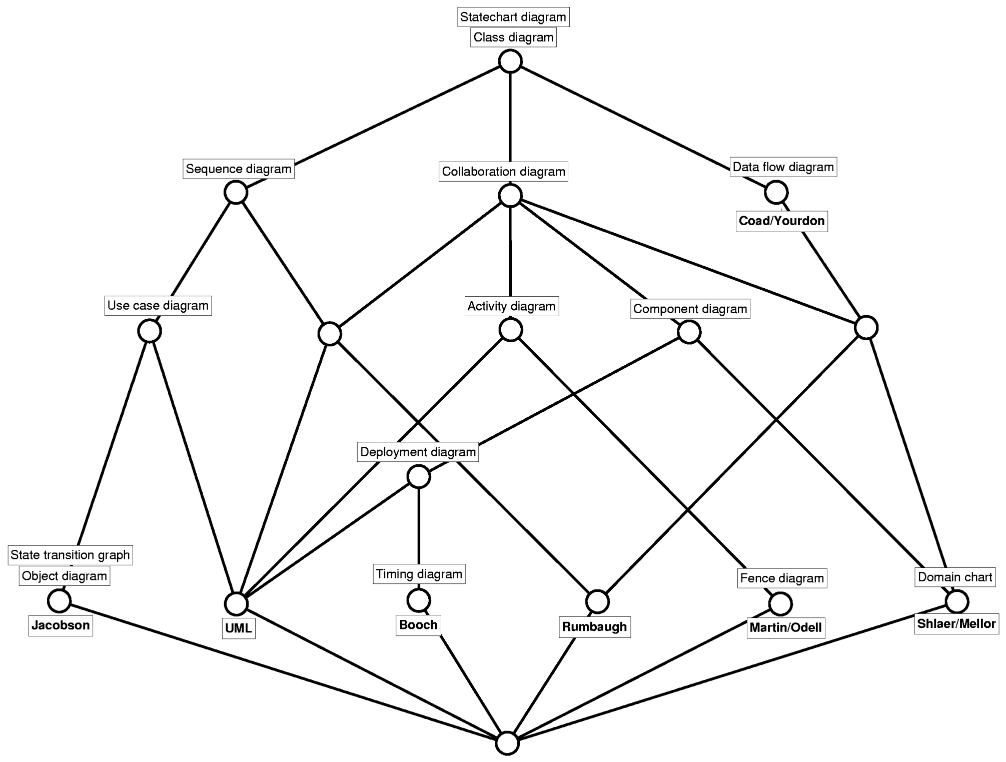


Fig. 2: Line diagram corresponding to fig. 1

For example, it can be read from fig. 2 that the node labelled *Use case diagram* represents the formal concept ({*UML, Jacobson*},{*Use case diagram*, *Sequence diagram*, *Class diagram*, *Statechart diagram*}) and that this is a super-concept of ({*UML*},{*Use case diagram*, *Sequence diagram*, *Class diagram*, *Statechart diagram*, *Activity diagram*, *Collaboration diagram*, *Deployment diagram*, *Component diagram*}) (cf. the node labelled *UML*) (μ(*Use case diagram*) ≥ γ(*UML*)). Indeed the second concept is a lower neighbour of the first.

But the structure of the line diagram even contains more information. In our example all considered methods use class diagrams to model the static system structure and statecharts to model dynamic aspects. Except for Coad/Yourdon all authors use sequence diagrams or collaboration diagrams to model dynamics. All three authors of the UML gave up a little of their ideas: Jacobson lost his object diagram and his state transition graph, Rumbaugh did not maintain the data flow diagrams and Booch abandoned his timing diagrams. Apart from these dropped concepts the UML is a union of all modelling concepts of the three authors where activity diagrams were added.

The formal context and the concept lattice represent two different views on the same information. Usually a line diagram of the concept lattice is computed from the formal context and further investigation of the context data is done with the help of the diagram. The computation of a correct diagram can be done automatically but normally the resulting diagram needs some manual re-arrangement in order to get a nice shape. The vertical position of a node relative to its neighbours cannot be changed without changing the semantics of the diagram (Tools keep track of this.). But the horizontal arrangement and the absolute distances can be arbitrarily chosen.

Sometimes another view turns out to be helpful: implications. An *implication* $A \rightarrow B$ with two sets $A,B \subseteq M$ of formal features states that every formal thing having all the formal features of A also has all formal features of B. Such an implication can be read from the line diagram this way: For every $b \in B$ the formal concept μb has a descending path to the infimum $\wedge\{\mu a \mid a \in A\}$. For example in fig. 2 we can see that every author who proposes sequence diagrams and data flow diagrams also suggests the use of collaboration diagrams. Thus e.g. {*Sequence diagram*, *Data flow diagram*} $\rightarrow$ {*Collaboration diagram*} holds. Correspondingly, an implication $C \rightarrow D$ with two sets $C,D \subseteq G$ of formal things holds if and only if every formal feature which belongs to all formal things of C also belongs to all formal things of D.

This small example already shows how ‘knowledge’ can be represented in a structured way by means of FCA.

4 Example: The business of a wine distribution centre

By using the following example we will demonstrate how Formal Concept Analysis (FCA) can be used to extract suitable class candidates from use case descriptions. These build the basis for a class model and the sequence diagrams of the use cases. The chosen example was presented in the FRISCO report ([FHL+ 98]). In one of our student projects it served for demonstrating the application of UML techniques. The results elaborated by the students during this project (cf. [FGT 98]) were the basis for this article's example. We applied FCA to the students' results.

The case study describes the development of a software system for 'Japan Wines, Inc.' (JWI). JWI is a wine distribution centre which orders wine from wineries and delivers it to retail shops. Originally the business of JWI was described in 28 informal statements. From these the students extracted eight use cases: *Receive order*, *Process order, Order missing products, Create delivery instructions, Process delivery results, Process incoming deliveries, Determine inventory stock, Define maximal and minimal stock quantity*.

To illuminate the style of the use case descriptions we rephrase the first two of them. The others are formulated in analogous manner (cf. [DüHe 98]).

„Receive order“

- The centre receives *orders* from *customers* by phone from 9:00 a.m. to 5:00 p.m..
- A received order is recorded on a form.
- An order may consist of many *detailed items*. Detailed items refer to single *products*. Each detailed item is recorded in a line of the form.

„Create delivery instructions“

- The centre produces a delivery instruction ticket for each delivery truck by gathering the ordered items in the 'assigned ordered items' file, considering the destinations and the total amount of the orders for each item.

In order to treat the use case descriptions with FCA we perform an indexing. In professional projects this would be done with the help of a domain expert. For each use case we list all the involved "things". These "things" are candidates for objects, classes and attributes in the later class model. In terms of FCA we treat the use cases as formal features and the real "things" as formal things. The incidence relation points out whether a "thing" is involved in a use case. If one use case U *uses* or *extends* another use case V, all "things" occurring in V are considered to be involved in U as well. Correspondingly, in the formal context an implication V $\rightarrow$ U between the formal features V and U is introduced.

Performing the indexing for our example we got the formal context in fig. 3. This formal context reflects the indexing of the use cases. Of course, such a result highly depends on the choices of "things" by the developer.

	Receive order	Process order	Order missing products	Determine inventory stock	Create delivery instructions	Define maximal and minimal stock quantity	Process incoming deliveries	Process delivery results
Customer order	×	×						
Customer	×							
Detailed ordered item	×	×			×		×	×
Product	×	×	×	×		×	×	×
Stock quantity		×	×	×			×	
Assigned ordered items file		×			×		×	×
Waiting ordered items file		×					×	
Missing quantity			×					
Order to supplier			×					
Supplier			×				×	
Minimal stock quantity			×			×		
Maximal stock quantity			×			×		
Delivery instruction					×			×
Delivery truck					×			
Destination					×			
Delivery result								×

Fig. 3: Formal context of use cases

"Things" that show up in a low position in the resulting line diagram are relevant for many use cases. These are "first class" class candidates. Now we consider the potential attributes of these class candidates. If they have already been mentioned among the examined "things", they are usually in a position above their class. This is at least the case if we assume that every use case which deals with an attribute has to access it via its class. In the formal context, an implication from the class attributes to the class should hold. This observation may be used to associate the right attributes to a class the developer has chosen. However, if different use cases use different attributes of a class, these may be wide spread over the line diagram.

In our case the preferred class candidates of the JWI system *Detailed ordered item* and *Product* are in the lowest positions of the diagram. All use cases use at least one of them. The attribute candidates *Stock quantity*, *Minimal* and *Maximal stock quantity* of the class candidate *Product* show up just above *Product*.

It is not sufficient to examine just the lowest level of the line diagram. For example, class candidates that are only interesting for single use cases show up at the same node as the corresponding use case.

The line diagram shows the dependencies between the data and functional view of our domain. Looking at the diagram from the supremum (top) side we can follow the refinement of the use cases representing the system functionality. Considering the diagram from the infimum (bottom) side yields the data view.

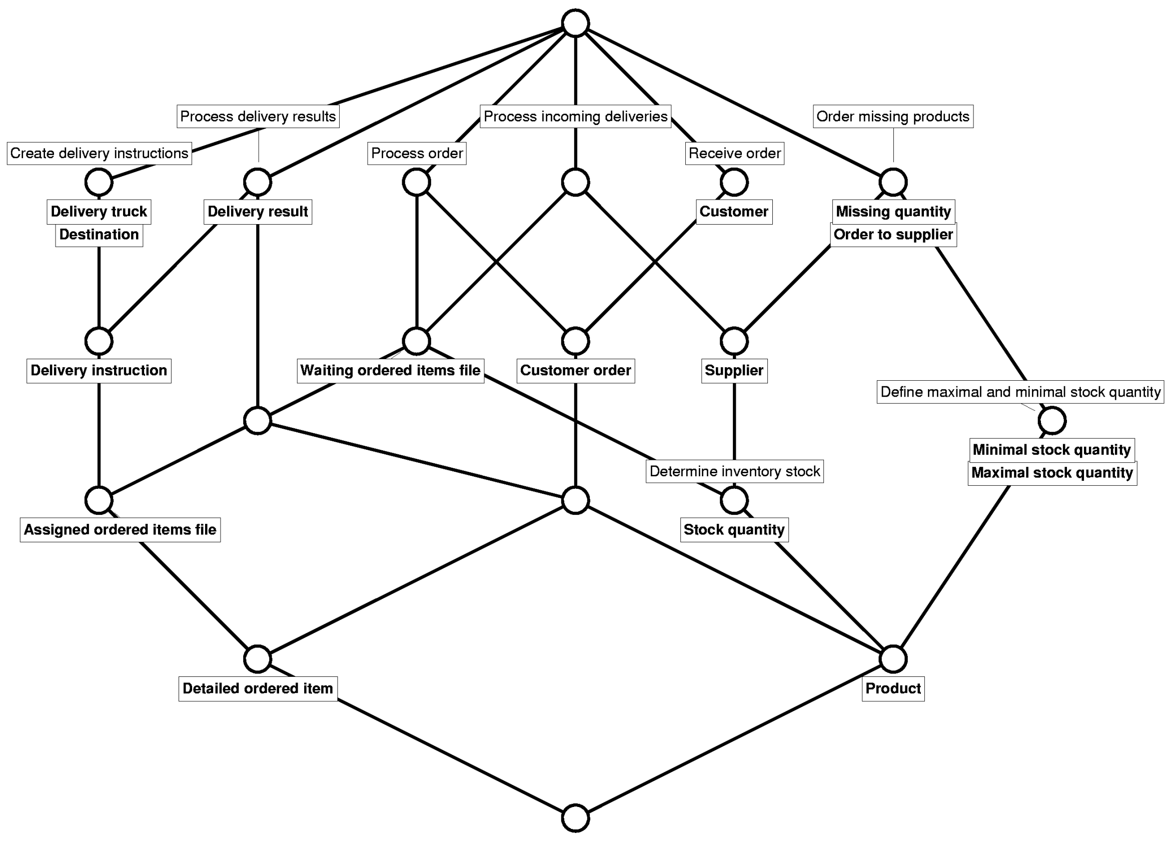


Fig. 4: Line diagram of JWI system

This type of line diagram gives valuable hints for finding classes among the "things" that have been chosen as relevant because it gives a ranking between them and a way to associate attributes to classes. Furthermore, if two nodes in sub-/super-concept relationship are considered as class candidates, this relationship may be interpreted as an inheritance or uses relation between the resulting classes.

But beyond this point an examination of class operations is missing. The use cases can be functionally decomposed step by step until the level of (class) operations is reached. The resulting "sub-use cases" are included into the analysis as new formal features. The formal context has to be extended correspondingly by marking the involved "things". Furthermore, the detailed functional perspective may yield new relevant "things" associated to the sub-use cases. To keep the formal context consistent, an implication between the sub-use case and the original use case has to be added (cf. treatment of uses relationships between use cases). From the new context a new line diagram is computed. Once the level of class operations is reached the new line diagram shows the data dependencies of the resulting operations.

One general problem in refinement processes is to decide where to stop the refinement. Formulating criteria for that is particularly difficult in our situation because we still do not have the final classes needed to decide whether a given sub-use case can be treated as a class operation or not. But FCA gives us support to define such criteria. In order to find potential operations to be associated to the class candidates, we examine the implications between formal things. In our case an implication $A \rightarrow B$ with two sets

A,B of formal things suggests that all operations which use all "things" in A also use all "things" in B. A basis of such implications can be computed. The implications of this basis are presented to the analysts in the form of questions and they have to decide if this implication really holds within the application domain. If not, there has to be an operation which is not yet included in the refinement and which separates the "things" of A and B. This operation has to be considered within the refinement of that use case.

For example in fig. 4 the implication *Delivery instruction* → *Assigned ordered items file* holds. Examination by the analysts shows that this implication does not correctly reflect the "real world" situation. Therefore, the use case *Create delivery instructions* is refined leading to three sub-use cases *Create delivery instruction*, *Insert detailed ordered item*, and *Attach delivery instructions*.

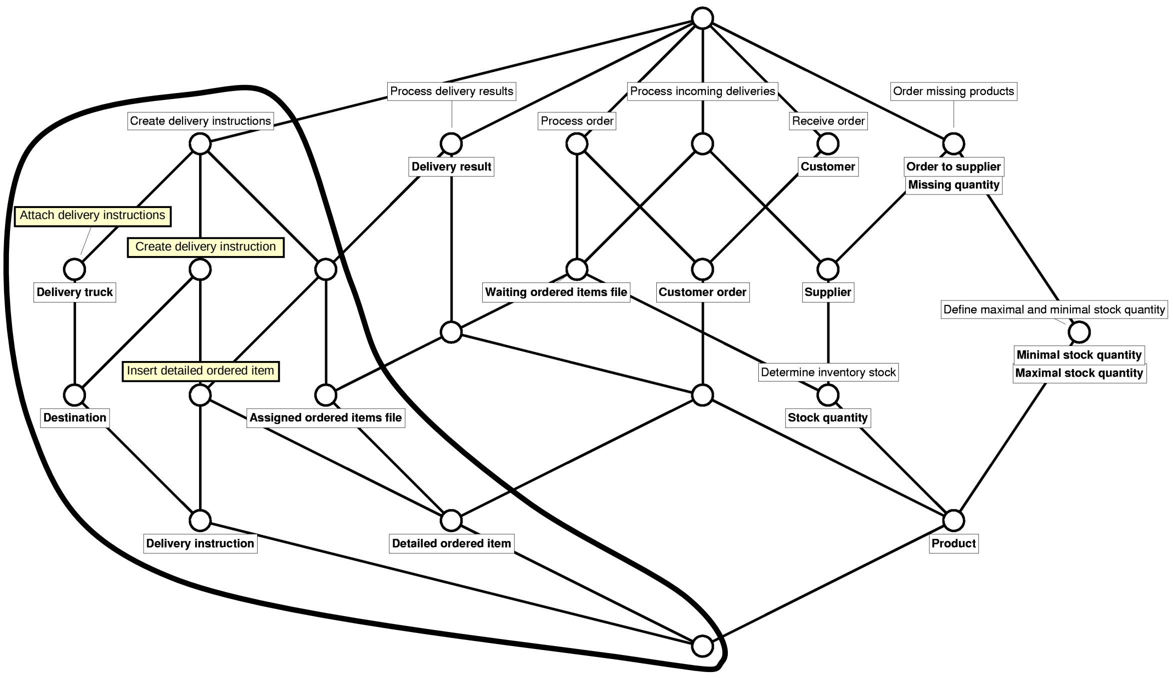


Fig. 5: Use case "Create delivery instructions" refined

By this refinement the "unnatural" implication is removed and three potential class operations have been found.

The line diagram contains data as formal things and operations as formal features. The incidence relation reflects which data is used in which operation. Lindig and Snelting already examined this situation while looking for a modular structure of existing systems (cf. [LiSn 97]). As module candidates they considered not only formal concepts, but also so-called "*blocks*" (cf. [GaWi 98]). In FCA a *block relation* of a formal context (G,M,I) is defined as a binary relation $J \subseteq G \times M$, satisfying:

1) $I \subseteq J$
2) $\forall\, g \in G : g^J$ is an intent of (G,M,I)
   $\forall\, m \in M : m^J$ is an extent of (G,M,I)

(G,M,J) forms a new formal context the formal concepts of which correspond to whole intervals (named "blocks") of the original concept lattice. All block relations to a given formal context can be automatically computed.

Analogously to Lindig/Snelting we aim to find additional class candidates by studying block relations. In object-oriented analysis attributes and operations are grouped to classes. In FCA this corresponds to grouping formal things and formal features to formal concepts. But within a formal concept each formal feature belongs to every formal thing and vice versa. In our case this is too restrictive because we cannot expect each class attribute to be involved in every class operation. Block relations can help us to generalise this grouping. In the formal context table the formal concepts are visible as maximal filled rectangles. According to a block relation incompletely filled rectangles are completed by additional marks (see condition 1 above). But this "filling" must be done carefully: The grouping of formal things to concept extents and of formal features to concept intents must not be changed (condition 2).

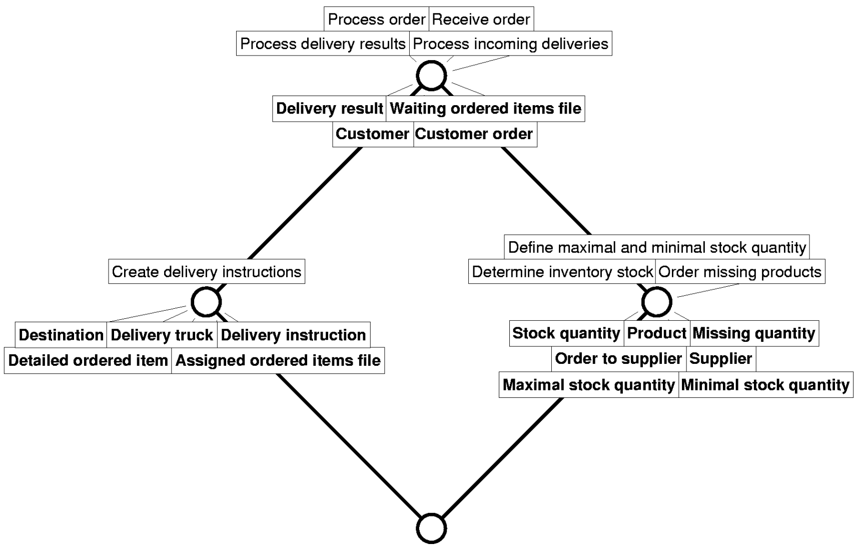


Fig. 6: Line diagram of a block relation context

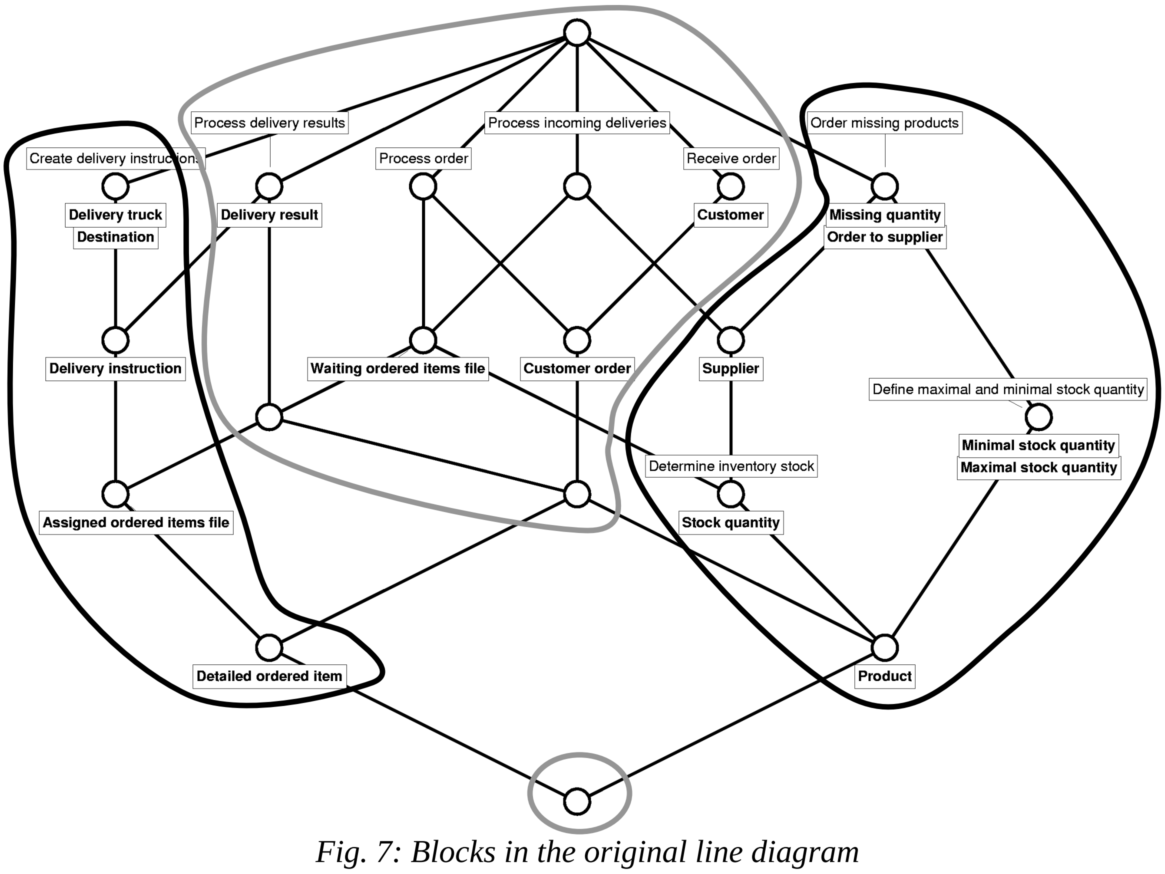


Fig. 7: Blocks in the original line diagram

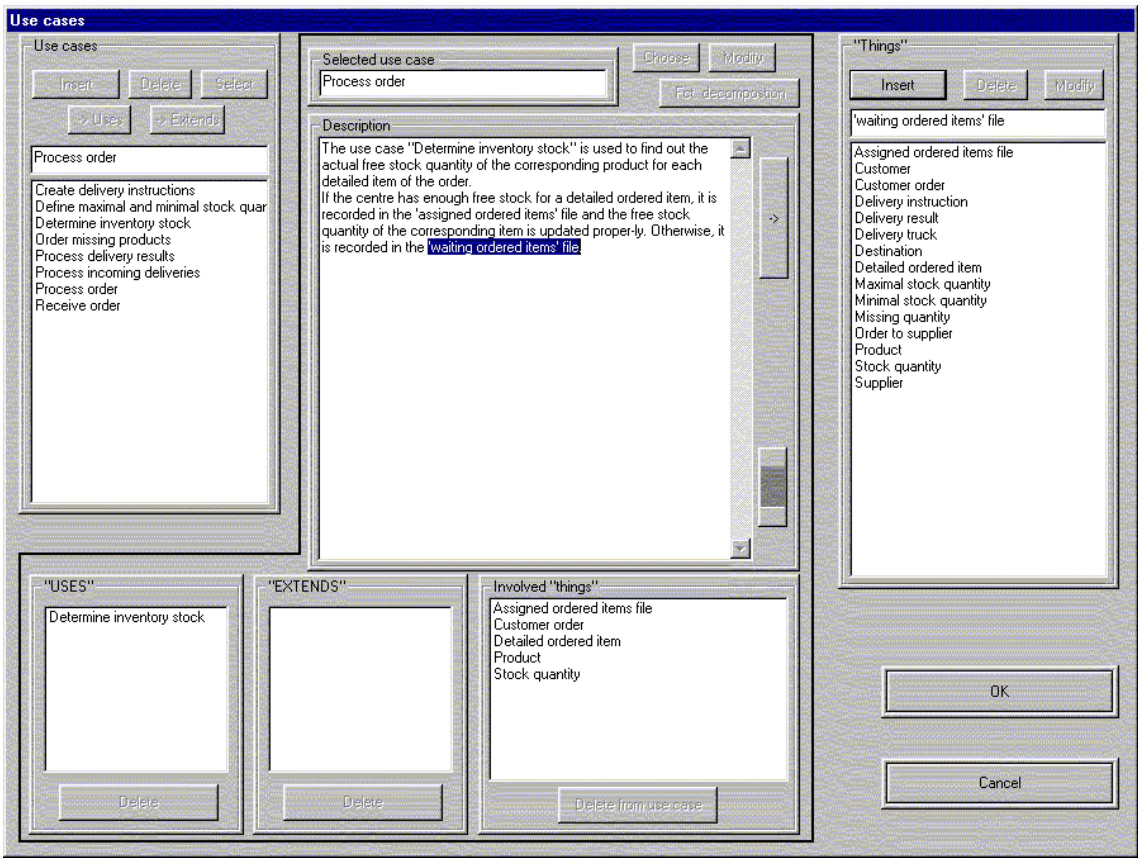
In our sample case we did not yet find interesting class candidates this way. But block relations turned out to be helpful in finding a coarse structure, i.e. packages or components of the system. In the example a block relation of the formal context in fig. 3 yielded the following line diagram in fig. 6. The corresponding blocks of the original concept lattice are marked in fig. 7.

The left block comprises everything about deliveries whereas the right block is concerned with the internal stock management. The central (higher) block describes a component with coordinates the activities of JWI in response to orders from retail shops.

5 Tool support

Currently we are developing a tool supporting the sketched analysis method. In this tool all mathematical background is hidden from its user. The only structure presented to him is the line diagram which he may interpret without knowing its exact mathematical semantics.

The user only has to name and describe his use cases. The tool allows him to pick words from the entered description and mark them as involved "things". Alternatively he may type in these "things" independently or choose them from the list of already entered "things". The tool stores these data in a formal context invisible to the user, computes a line diagram of the corresponding concept lattice and displays it to the user.



Use cases

Process order

Create delivery instructions
Define maximal and minimal stock quar
Determine inventory stock
Order missing products
Process delivery results
Process incoming deliveries
Process order
Receive order

Selected use case: Process order

Description: The use case "Determine inventory stock" is used to find out the actual free stock quantity of the corresponding product for each detailed item of the order.
If the centre has enough free stock for a detailed ordered item, it is recorded in the 'assigned ordered items' file and the free stock quantity of the corresponding item is updated proper-ly. Otherwise, it is recorded in the 'waiting ordered items' file.

"Things"

'waiting ordered items' file

Assigned ordered items file
Customer
Customer order
Delivery instruction
Delivery result
Delivery truck
Destination
Detailed ordered item
Maximal stock quantity
Minimal stock quantity
Missing quantity
Order to supplier
Product
Stock quantity
Supplier

"USES"

Determine inventory stock

Involved "things"

Assigned ordered items file
Customer order
Detailed ordered item
Product
Stock quantity

Fig. 8: Use case dialog

Fig. 8 shows the dialog to enter use case data. The upper left list box shows all use cases that were entered by using the edit box above. From this list a use case may be selected for entering more details. The selected use case is shown in the middle boxes.

The dialog elements referring to this use case are surrounded by a black line. There is an edit field for the description and there are list boxes showing the *uses* or *extends* relationships together with the "things" involved in the use case. On the right hand side there are the list box for all entered "things" and the corresponding edit box. In fig. 8 the user has selected the term *'waiting ordered items' file* from the use case description. He now may edit it in the right edit box and insert the edited name into the list of all examined "things" and at the same time attach it to the selected use case. *Used* or *extended* use cases may simply be picked from the list of use cases.

This way all identified use cases are specified as described in section 4. In our example, the tool computed the following line diagram from the use case data:

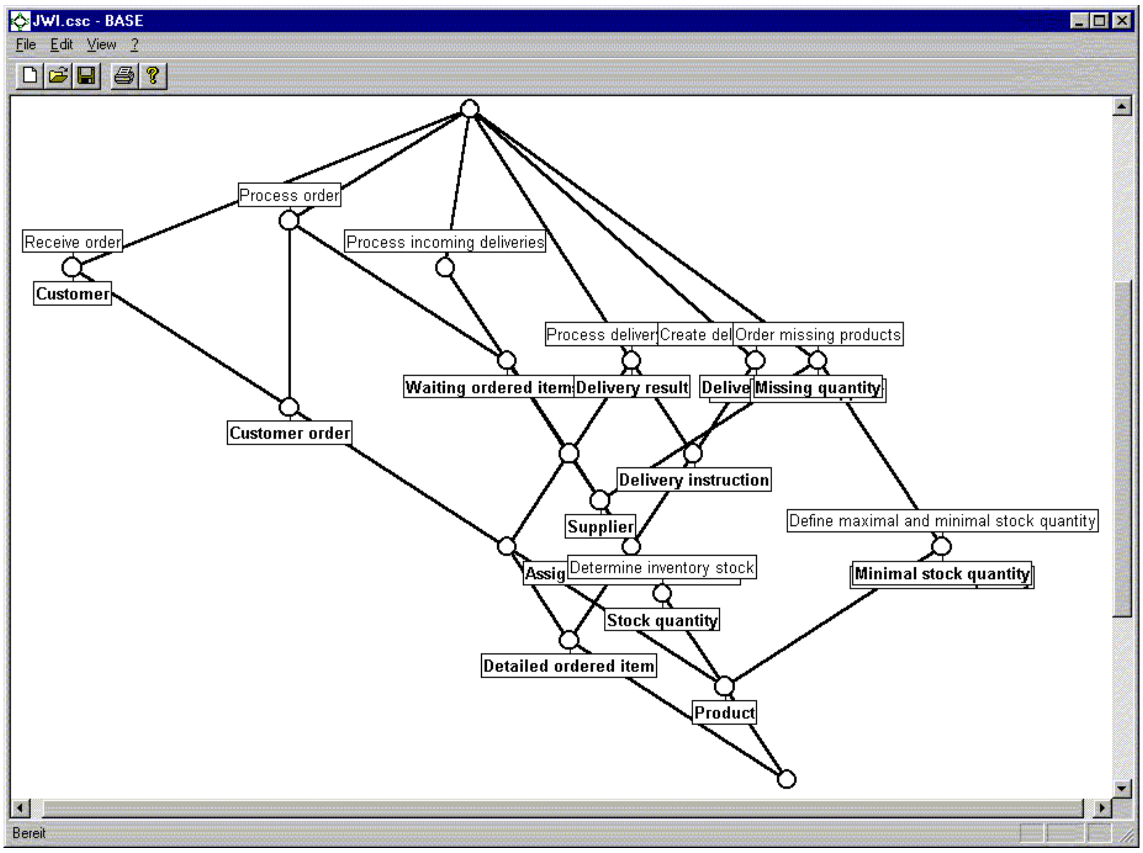


Fig. 9: Automatically computed line diagram

The shown concept lattice is just the one of fig. 4. For the computation of the diagram, our tool uses "The Formal Concept Analysis Library" developed by Frank Vogt in Darmstadt (described in [Vogt 96]) with some modifications. The automatic drawing can only guarantee the correctness of the diagram but not its beauty. For the second the user is responsible. He may drag the diagram's vertices (a single one, all above or beneath one) and the text and drop them in the position he likes. But the tool does not allow him to destroy the vertical order of connected nodes.

6 Conclusions

In this article we have demonstrated how Formal Concept Analysis can be used to guide the transformation steps from a use case model to the class structure model during the early stages of system analysis. For this transformation we profited from the line diagram of FCA which enabled us to study the given application domain simultaneously from two perspectives – combining a data and a functional view. In our experience, this approach to "play" with class candidates and to try out different solutions for associating classes and class components with each other is particularly promising.

However we are aware that this important and difficult task requires much detailed work which cannot completely be automated, including negotiations and a thorough discourse between domain experts and software developers – at least if certain quality standards are to be achieved. As we have shown, Formal Concept Analysis, the resulting line diagrams and the methods to refine and examine such diagrams can successfully be applied to visualise complex situations and thereby facilitate the communication between the people involved.

The tool we have developed allows the user to enter uses cases and their descriptions. From the use case descriptions he may select words as involved "things". These words can be modified to conform to system wide conventions. Furthermore "things" can be entered independently from the use case descriptions and *uses* or *extends* relationships between use cases may be defined. From this a line diagram is automatically computed as described in section 4. It is the only mathematical structure shown to the user. Thus he (or she) does not have to be familiar with the mathematical background because the line diagram is readable without knowing anything about lattices. Using this tool the additional effort spent for indexing of the use cases and for formatting the corresponding line diagrams is soon compensated by the advantages of a transparent visualisation and an eased communication. Currently we are working on a method to support the comparison of different line diagrams to integrate views of different analysts.

In this paper we have focussed one particular aspect of OO modelling and one particular use of FCA for this task. However other tasks within the OO analysis and modelling field might be considered as well, for example, process modelling, clustering of classes and objects or modelling of states and transitions. Moreover there is a broad application field of FCA for other related areas such as software design or re-engineering. For example, successful applications of FCA have been reported for the structuring of class models (cf. [GMM+ 98], [SnTi 98]) and for modularization of already existing systems (cf. [LiSn 97]). In the future, we envision a broad, complementary support of many activities in the early stages of Software Engineering by Formal Concept Analysis methods – in a field which traditionally had been reserved to the bare intuition of the involved experts.

7 References

[BJR 97] G. Booch, I. Jacobson, J. Rumbaugh: UML 1.1 Notation Guide, Rational Software Corporation, 1997, http://www.rational.com/uml /resources/documentation/notation/index.jtmpl

[Booc 94] G. Booch: Object-Oriented Analysis an Design with Applications, Benjamin/Cummings 1994

[CoYo 90] P. Coad, E. Yourdon: Object-Oriented Analysis, Prentice Hall 1990

[DüHe 98] S. Düwel, W. Hesse: Identifying Candidate Objects During System Analysis, Proc. CAiSE'98/IFIP 8.1 3rd Int. Workshop on Evaluation of Modeling Methods in System Analysis and Design (EMMSAD'98), Pisa 1998

[FGT 98] P. Franke, T. Graf, M. Trouvain: Fortgeschrittenenpraktikum Informatik, "Einarbeitung in UML 1.1 anhand eines Fallbeispiels", Philipps-Universität Marburg, http://www.mathematik.uni-marburg.de /~hesse/uml/Welcome.html, 1998

[FHL+ 97] E. Falkenberg, W. Hesse, P. Lindgreen, B.E. Nilsson, J.L.H. Oei, C. Rolland, R.K. Stamper, F.J.M. Van Assche, A.A. Verrijn-Stuart, K. Voss: FRISCO - A Framework of Information System Concepts - The FRISCO Report, ftp://ftp.leidenuniv.nl/pub/rul/fri-full.zip, 1997

[GaWi 98] B. Ganter, R. Wille: Formal Concept Analysis, Mathematical Foundation, Springer 1998

[GMM+ 98] R. Godin, H. Mili, G.W. Mineau, R. Missaoui, A. Arfi, T.-T. Chau: Design of Class Hierarchies based on Concept (Galois) Lattices, Theory and Application of Object Systems (TAPOS), 4(2), pp. 117-134, 1998

[JCJ+ 92] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard: Object-Oriented Software Engineering - A Use Case Driven Approach, ACM-Press, Addison-Wesley 1992

[LiSn 97] C. Lindig, G. Snelting: Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. Proc. International Conference on Software Engineering (ICSE 97), Boston, USA, pp. 349-359; 1997

[MaOd 92] J. Martin, J. Odell: Object-Oriented Analysis and Design, Prentice Hall 1992

[Meye 88] B. Meyer: Object oriented software construction, Prentice Hall 1988

[RBP+ 91] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W. : Object oriented modelling and design, Prentice Hall, Englewood Cliffs 1991

[ShMe 91] S. Shlaer, S.J. Mellor: Object Lifecycles, Modeling the world in states, Yourdon Press 1991

[SnTi 98] G. Snelting, F. Tip: Reengineering Class Hierarchies Using Concept Analysis, Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 99-110, 1998

[Vogt 96] F. Vogt: Formale Begriffsanalyse mit C++, Datenstrukturen und Algorithmen, Springer 1996