Component-Based Design: Alternatives During Partition Phase

Arantza Irastorza, Arturo Jaime, Oscar Díaz[1]
Department of Computer Languages and Systems
University of the Basque Country
P.O. box. 649. 20.080 San Sebastián (Spain)
e-mail: {jipirgoa, jipjaela, jipdigao}@si.ehu.es

Abstract

Several guidelines have been proposed for component-based design. A key step in these methodologies is component partition, whereby the system's functional requirements are assigned to the components. This work compares the different partition methods proposed so far. To this end, the same case study is applied to each method, and the resulting component architecture compared. As a result, we show the impact that the chosen method has on the resulting architecture, or more specifically, on the granularity of the components.

Keywords: component design, component identification, study of alternatives, granularity of components.

1. Introduction

Component-Based Design (CBD) is being fuelled by the promise of important gains in both productivity and quality. Productivity will be positively affected by using larger abstractions for analysis, design, and development. Quality will be positively affected by reusing known, proven solutions and the components that implement them [11]. It exists a graving demand for clear CBD guidelines as system complexity increases. The limited experience on using this approach makes architecture designers be doubtful about which is the most appropriate method. Although all the CBD methods share the notion of a system as an interconnected set of components, we are still far from a consensus on objectives, techniques and diagrams. This work focuses on the partition of the system into components, or the component-packaging phase [3].

Once the static and dynamic aspects of the domain have been captured (e.g. through use cases), the partition stage strives *to determine what functionality should be supported by which component*. Various approaches can be found in the literature. For example, guidelines can be based on use cases [10, 2], design patterns [11], business domains [1], the foreseen evolution [8] or the pre-existent components [12]. Each approach gives priority to a different aspect of the development. Thereby, guidelines based on "the pre-existent components" can be useful for an organization with a lot of legacy software, whereas an approach based on "the foreseen evolution" suits organizations that are willing to minimize the impact of a possible evolution on their requirements. Determining which method is the most suitable for the current design can be challenging for the designer. The chosen option determines the resulting architecture, specifically, the granularity of the components.

This work addresses the impact of the partition method on the resulting component architecture. To this end, five partition techniques are confronted with the same set of functional requirements -stated by means of use cases. The resulting component architectures are compared and conclusions draw. The partition has been conducted according with the guidelines found in the literature. We have to admit that these guidelines are not always as complete as required to apply to a case example in a step-by-step way. We will notice this as required throughout the paper.

The article is structured as follows. The case example is introduced in section 2. This example is used in section 3 to compare the distinct approaches to the partition phase. Finally, the conclusion section discusses the resulting architectures and the impact on component granularity.

2. Case study

Let us consider the information system of a bank. During the requirements analysis the fol-

[1] This research was supported by both the Secretaría de Estado de Política Científica y Tecnológica, under contract TIC 1999-1048-C02-02 and the Departamento de Educación, Universidades e Investigación of the Basque Government under contract UE2000-32.

lowing use cases have been identified (see figure 1):

- *Balance retrieval.* After verifying her identity and providing an account number, a bank client will be able to see her balance.
- *Refund.* After verifying her identity and providing an account number, a bank client will be able to request the amount of cash money that she wants. If the requested quantity does not go beyond the quantity available in the automatic telling machine (ATM), the account balance is checked. If the balance is greater than the requested quantity, it will be updated and the amount will be drawn out from the safe. If the amount of requested money is up to 20% greater than the balance due to an overdraft, a loan application process will be initiated. In both cases, a receipt for the operation will be handed to the client. If the amount exceeds 20% of the account balance, the refund operation will be cancelled and the client will be informed about it.

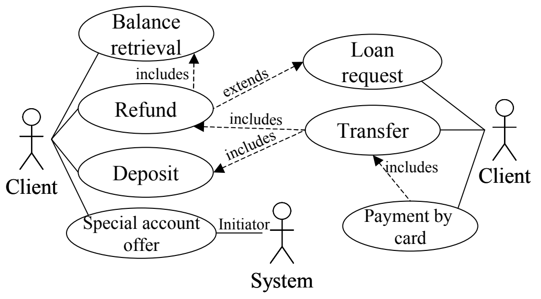

Fig. 1. A use case diagram

- *Deposit.* A bank client provides an account number, registers the amount of money that she wants to deposit and hands over the amount to the machine. After checking that the specified amount and the handed over amount are the same, the account balance will be updated. The client will receive a receipt that the operation has been carried out.
- *Transfer.* A bank client has verified her identity and provides two account numbers and the amount of money to be transferred. After verifying the existence of both accounts the system checks that the first account from which the money is going to be transferred has sufficient funds. If both processes have been successful, the balances will be updated subtracting the quantity from the first account and adding it to the second one. At the end a receipt for the operation will be handed to the client. If the amount to be transferred is up to 20% greater than the first balance, both balances will be updated and a receipt for the operation will be handed to the client. Moreover, the difference between the requested and the available quantities will be registered as a loan on the first account. However if this amount exceeds 20% of the account balance, the transfer operation will be cancelled and the client will be informed about it.
- *Payment by card.* After identifying herself to the system and providing the access key of the electronic card to one of her accounts, a bank client indicates the amount of money she wants to pay and the account from which the money is going to be taken. The system verifies the existence of the account and checks whether the account balance associated with the credit card is greater than the amount stored. If this is true, both account balances, the credit card account and the account the money is paid into, will be updated by subtracting the quantity to be paid from the former and adding it to the latter. At the end a receipt for the operation will be handed to the client. If the amount of requested money is up to 20% greater than the balance, both balances will be updated and a receipt for the operation will be handed to the client. Moreover, the difference between the requested and the available quantity in the account will be registered as a loan associated to the account which the card belongs to. When the quantity to pay exceeds 20% of the account balance, the payment operation will be cancelled and the client will be informed about it.
- *Special account offer.* When an account balance increases beyond a certain threshold, information about other bank products (e.g. pension plans or investments on the stock market) will be sent to the client automatically.
- *Loan request.* Loan requests are managed in an automatic way. Loans can be of three types: 'mortgage', 'personal' or 'by overdraft' (when someone makes a payment that exceeds the overdraft allowance). When a client applies for a loan, her identification and the state of her account are checked. The concession of the loan will depend on that state, that is, whether already another loan is associated with the account and which type of account it is. Moreover for some types of loans the client's salary or the account average balance during last year might be checked as well.

3. Partition in components

After gathering the system requirements, the designer faces the problem of how to distribute these requirements among the components. The literature offers different guidelines to carry out this partition process. Some of these proposals can be complementary while others are opposed. The aim of this work is to check these guidelines, applying them to the same group of requirements (as described in section 2).

3.1. Partition by Use Cases

They constitute a division of the system, based on its expected functionality. In [10, 2] it is suggested that the functionality expressed in each use case of the system can be defined in a separate component.

From a use case, a class diagram that satisfies its needs is extracted. These classes will group together around a component. At first sight, it appears that putting this option into practice is simple, like the example in figure 2[2].

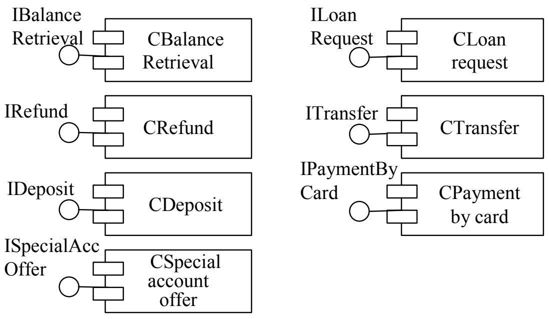

Fig. 2. Result of the partition by use cases

However, we need to consider that the same class could appear in several use cases. For example, the *Account* class and the *Client* class appear in both, the 'Balance retrieval' and the 'Refund' use case. Moreover the *Account* and *Transaction* class are repeated in the 'Refund' and the 'Deposit' use case. Thus, if a class is in two different use cases, it should appear in both components describing these use cases. In our example we would have repeated the *Account* class in the *CBalance Retrieval*, *CDeposit* and *CRefund* components.

The reviewed references [10, 2] do not clarify what should be done in this case. We suggest the following possibilities:

- **Distributing.** The class is distributed among different components [4]. For example, the *Account* class would be distributed among the *CBalance Retrieval* and the *CRefund* components (see figure 3). Each component includes those parts of the *Account* class that it needs although, this means that state information or functionalities will have to be repeated for both components (this can also be seen in figure 3). Thus, in our model a real world account would be shown as two instances: one in the *CBalance Retrieval* component, and another one in *CRefund*. Therefore the attribute *number* must contain the same value in both instances. In this way, two instances that have the same *number* would describe two views of the same account: the account view when someone retrieves its balance, and the account view when someone makes a withdrawal.

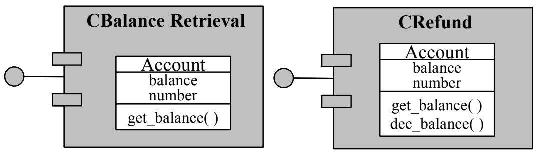

Fig. 3. The *Account* class distributed in several components

However, it is not always that easy to distribute the functionalities of a class among several components. For example, the *balance* attribute is accessed by the *CBalance Retrieval* component to retrieve a balance and by the *CRefund* component to decrease a balance. Therefore the *balance* attribute should be included in both component instances and contain the same value. This example outlines the problem involved in assessing consistency among copies, similar to the consistency problems encountered in database design.

- **Concentrating.** Putting the class in question in a single component can alleviate the problems mentioned above. The other components will then access the functionalities of the class via the services provided by that component. Thus, we would define the *CBalance Retrieval*, *CDeposit* and *CRefund* components for the corresponding use cases. The *CBalance Retrieval* component, for example, would contain the *Account* class and would offer services to the other two components (by means of the *IAccountDeposit* and the *IAccountRefund* interfaces). There would result in *CDeposit* and *CRefund* being dependent on *CBalance Retrieval*, which has been indicated by a dotted arrow in figure 4. This idea is the same than the partition by design patterns approach [11], which will be discussed in the next subsection[3].
- **Separating.** A variant of the previous alternative would be to define for the *Account* class a

[2] Along the article, identifiers with prefix C denote the name of a component, while the prefix I corresponds to the name of a service, to the interface of a component. Class, component and interface names appear in italic and use case names in single quotation marks.

[3] An extreme case will occur when all classes of the component move to other components. In this case it might be unnecessary to maintain a component without classes.

separate component that provides services to all other components that need to access its functionalities (see figure 5). This case would result in components, such as *CAccount*, which do not fit directly to a use case.

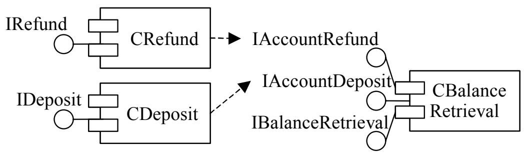

Fig. 4. Partition by use cases (concentrating)

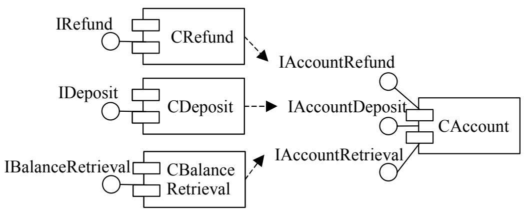

Fig. 5. Partition by use cases (separating)

- **Gathering.** In contrast with the previous alternatives, this proposal groups those use case functionalities in the same component that share the same class[4]. In our example, the 'Refund', the 'Deposit' and the 'Balance retrieval' use cases would be supported by the same *CAccount Transaction* component which would include all identified services, as shown in figure 6.

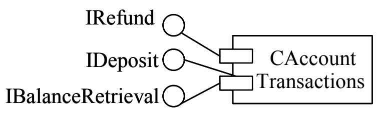

Fig. 6. Partition by use cases (gathering)

3.2. Partition by Design Patterns

Use cases determine the problems that need to be solved. Design patterns are proven design solutions. The correspondence between use cases and design patterns is more a question of intuition and experience than methodology. Starting from use cases, Larsen proposes to use design patterns to extract the key components [11]. This process consists of (1) defining the use cases, (2) identifying the needed design patterns and (3), in the partition phase, defining the components derived from the patterns. These stages will be applied in an iterative way until the whole system has been designed.

Each pattern gives rise to an initial component. However, several patterns could include the same classes. In this case, the classes could be distributed among several components, similar to the use case based approach.

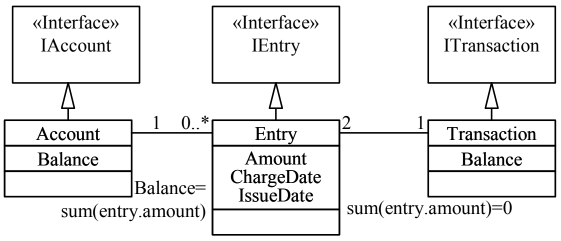

Fig. 7. *Account* Design Pattern

Finding the pattern that fits to a specific use case is based on the designer's ability and on a systematic search. Thus, the designer, using her previous experiences, would see that the structural patterns *Account* [5] and *Party* [7] bring a solution to all use cases, since they include the account and the account holder notions, respectively. Figure 7 shows the class diagram of the *Account* pattern. The attributes and operations that define the interfaces of these classes (i.e. *IAccount*, *IEntry* and *ITransaction*) will be used to support the use cases. The *IParty*, *IPerson* and *IOrganization* interfaces of the *Party* pattern classes (in figure 8) will be used in the same way.

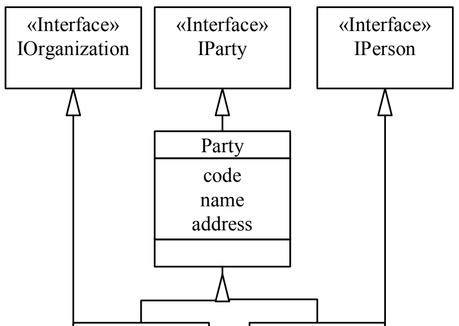

Fig. 8. *Party* Design Pattern

Once the pattern has been identified, a component that includes the functionality associated to this pattern will be defined. Thus, the two previous patterns will give rise to the *CAccount* and *CParty* components, respectively. The former will offer the *IAccount*, *IEntry* and *ITransaction* services, and the latter the *IParty*, *IPerson* and *IOrganization* services. Therefore, the interfaces of the classes included in a component shape the interfaces of that component.

[4] The problem with this technique is that too many use cases might be grouped in the same component. In such case, it might be preferable to apply the idea to a subset of the affected use cases and to use different alternative for the rest.

At a later stage of this iterative process, the *Contract* structural pattern [7] of figure 9 could be used to design the 'Loan request' use case. A loan can be understood as a contract type, where the object is the loan, and the contracting parties are the client and the bank. From this pattern the *CLoan* component will be defined.

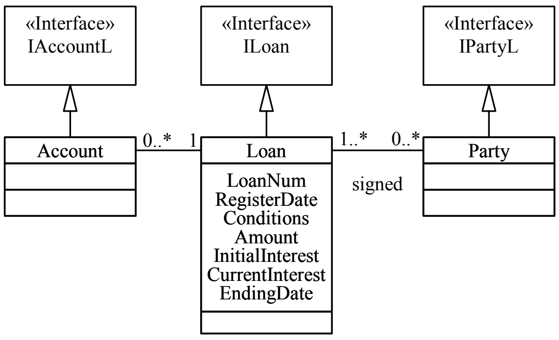

Fig. 9. *Contract* Design Pattern

Unlike in the previous cases, this new component will not include all classes from that pattern. The *Contract* pattern has three classes: *Party*, *Account* and *Loan.* Both *Party* and *Account* are already present in the other previously identified components, *CParty* and *CAccount*. Therefore, to give support to the 'Loan request' use case, it will only be necessary that *CParty* and *CAccount* offer in their interfaces the corresponding services, which are *IPartyL* and *IAccountL*. This possibility is shown in figure 12, where the *CLoan* component only offers the *ILoan* interface and uses the *IAccountL* and *IPartyL* interfaces. In this example, only the *Loan* class will be part of the *CLoan* component.

Since the 'Loan request' use case regulates the loan granting, depending on the state of the account, it would be possible to use the *State* design pattern [6] for this design, too (see figure 10). From this pattern a *StateAccount* class is defined. It has some subclasses, one for each significant account state determining the characteristics of the loan, and 1:1 relationship with the *Account* class. The *CStateAccount* component will be defined in such a way that it draws these services together.

In the same way than the *CLoan* component, this new component will not include all pattern classes. Since the *Account* class is already integrated in the *CAccount* component, a new interface, *IAccountSt*, will be added to the component in order to provide a service to the *CStateAccount* component as shown in figure 12.

On the other hand, the analysis of the 'Special account offer' use case provides a possibility for using the *Observer* behavioural pattern [6] in figure 11. The 'Special account offer' use case makes offers to those accounts that comply with certain conditions. In this case the account would be the observed object, and the observation function, specified in the *Observer* class, would watch the level of change of the account balance.

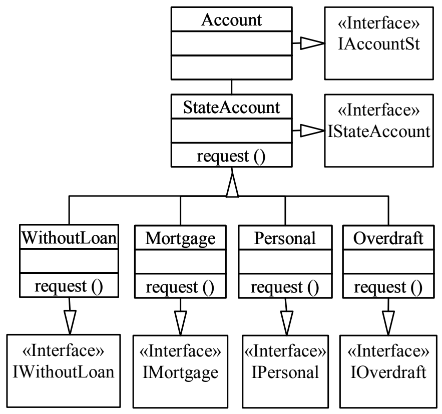

Fig. 10. *State* Design Pattern

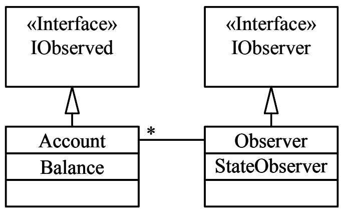

Fig. 11. *Observer* Design Pattern

During the partition phase, this pattern would give rise to another component, called *CObserver*. Since the *Account* class is already integrated in the *CAccount* component, the *IObserved* interface will be added to the *CAccount* component in order to provide a service to the new *CObserver* component (see figure 12). This new component will only include the *Observer* class.

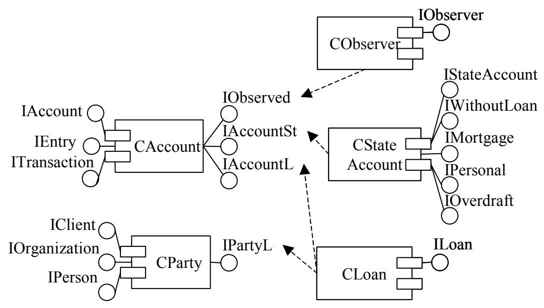

Fig. 12. Partition by design patterns

Figure 12 shows the resulting component diagram after a partition process based on design patterns.

3.3. Partition by Business Domains

A business domain models a part of the organization infrastructure, including generic services and business rules. Each business domain creates a component[5]. Allen and Frost suggest to use this notion like a guideline during the partition phase. Their method is described in [1].

In order to determine the domains of our example, we are going to use the group of guidelines provided by the method and the system requirements. Three business domains can be distinguished: *Account*, *Party* and *Loan*. These domains are only useful to delimit the three initial components of our system (*CAccount*, *CParty* and *CLoan*). The latter stages of the method will be used to fill these components with content, identifying their services and refining their internal design. First an initial class diagram is described for each component. Figure 13 shows the class diagrams for our three domains. The method recommends to take into account the design patterns in order to define the class diagrams, since many problems are reoccurring.

Next, a state diagram is defined for each class and the use cases are again analysed by means of their sequence and collaboration diagrams. Figure 14 shows the collaboration diagram of the 'Loan request' use case.

From this collaboration diagram, the *checkIdentifier* and *getSalary* services in the *CParty* component and the *getState* and *getAverageBalance* services in the *CAccount* component are identified. These services will be added to the list of methods of the respective classes in the class diagram of figure 13.

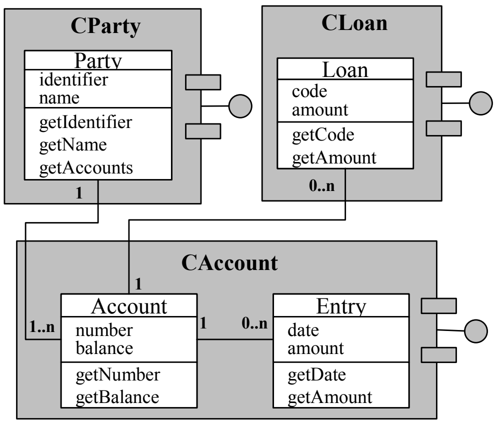

Fig. 13. Class diagrams of the initial components

The result component diagram for the 'Loan request' use case, is shown in figure 15. The process would be completed when all use cases had been analysed.

In spite of what has been said above, the initial identification of business domains has a certain amount of intuition. This can lead to certain parts of the system, that would be good candidates to form separate components, not being considered. Because of this problem, the method includes the possibility of identifying during the design process additional components. For example, such an additional component would be identified when detecting a group of classes that offer a cohesive group of services. Apart from identifying new component candidates, the designer can also study the utility of pre-existent components or subsystems.

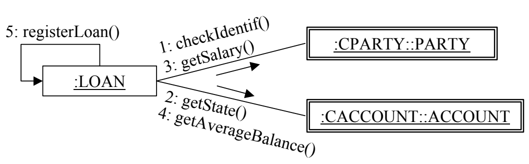

Fig. 14. Collaboration diagram of the 'Loan request' use case (The boxes with double line symbolize external domains. The notation :A::B represents the B class of the A component, and :B represents the B class)

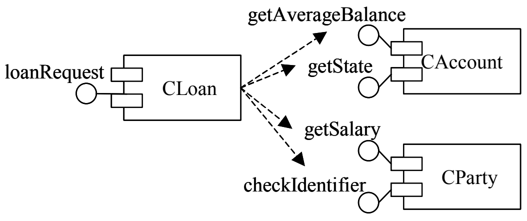

Fig. 15. (partial) Partition by business domains

3.4. Partition by the foreseen evolution of the system

This approach deals with both the current requirements of the system and its possible future needs. The aim is to ease the evolution of the system by anticipating possible changes so that a later amendment to the system only affects one component. Hoover *et al.* are exponents of this approach [8].

For our example we foresee the following changes:

- Change of currency from peseta to euro.
- Management of the commission on accounts whose balance is less than a certain quantity.
- Introduction of purse cards.
- Management of loans by overdraft.
- Electronic banking.

5 Jacobson et al. propose to make the division being based on the domain, but their perspective is nearer to the class or abstract data type identification [10].

Once the requirements have been identified: (1) the operations/attributes must be identified, (2) the way the possible changes can affect these operations/attributes is determined, (3) a component is defined for each group of operations/attributes affected by the same change and (4) a first list of components is refined taking into account those operations/attributes affected by more than one change.

Thus in our example, once the operations and attributes have been identified and it has been determined how they could be affected by the changes, five components, one for each possible change, can be identified. However, the 'introduction of purse cards' and 'electronic banking' changes affect the *client identification* operation. This results in grouping all those operations/attributes affected by both changes into one component. A possible outcome of this process is shown in figure 16. The *CAccount* component includes the operations/attributes affected by the first two changes, while *CMoneyTransfer* includes those affected by the other three changes.

The proposal of Hoover *et al.* does not indicate how those operations/attributes not affected by any change are distributed into components. Furthermore, these authors understand a component as a group of attributes and operations, without determining if the component organizes these aspects using one or more classes [4, 13]. In this last case, a possible variation to the work of Hoover *et al.* is to consider the class (and not the operation) as the impact unit of the change. Obviously, in order to detect whether a class is affected by a change its operations or attributes will have to be analysed. However it is the class, as unit, which will be located in one component or another as a result of the analysis.

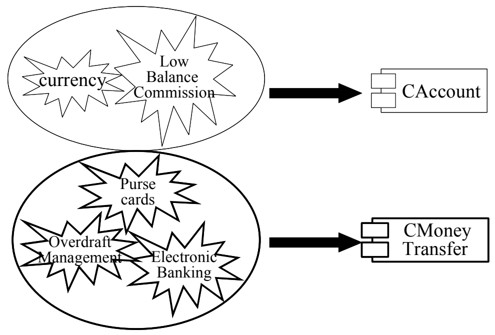

Fig. 16. (partial) Partition by foreseen evolution

For example, when starting the process from the use cases, a first class diagram is defined. Next the way the changes might affect the operations or attributes of each class is anticipated. For example, the 'electronic banking' change affects the *getBalance* and *getAmount* operations of the *Account* and the *Loan* class. In the next step, those classes affected by the same change are assimilated in the same component. Thus, the *CElectronicBanking* component will include the *Account* and the *Loan* class. If the same class is affected by several changes, their components will be merged. For instance, the *Loan* class is affected by both the 'electronic banking' change and the 'overdraft management' change. As result, the *CElectronicBanking* and the *COverdraftManagement* components are combined in a single component, called *CComponent1*. The result is shown in figure 17.

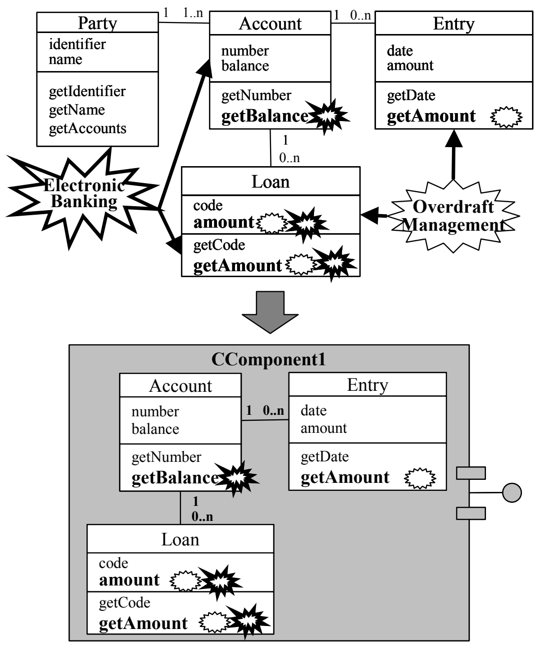

Fig. 17. Grouping of classes in components based on the foreseen evolution

As mentioned before, some classes might not be affected by any change. If this is the case the alternatives described in the previous sections (use cases, design patterns or business domains) can be applied.

A possible example for the use case alternative might be a use case whose class diagram is made up exclusively of classes unaffected by any change. All classes of the use case would make up a component, such as described in section 3.1.

We can also find that one class of the use case has been included in a component that might be subject to changes. This is a similar situation to the one described in section 3.1 where the same class took part in two different use cases. The proposed solutions (*distributing*, *concentrating*, *separating* and *gathering*) can also be used for this situation, with one restriction: a class assigned to a component that might be subject to changes cannot be separated from it[6], because a change affects one component.

The same applies to situations where a design patterns or business domains approach has been chosen.

3.5. Partition by pre-existent components

The aim of the previous methods was reuse (i.e. *design for reuse*), but they did not start by reusing components (i.e. *design with reuse*). This means, the process of component identification did not take into account, at least explicitly, already existing components. Sametinger promotes this latter approach [12] which is a bottom-up development alternative, where the construction of the new system is based on obtaining the appropriate "materials" that are already available.

Jacobson *et al* [10] have also supported the design approach of reusing components. They pointed out that it is more convenient to base a design on pre-existent components, instead of searching *a posteriori* to the design process for components that might fit into the design. The phase of the design life cycle in which a component can be created depends on the available information about it. For example, if its use cases are known, it can be integrated in the requirements analysis. However, if only its specification is known, it can only be considered in the design phase.

An effective reuse of components involves finding, understanding and adapting existing components. The search requires a repository that contains and manages information about the available components. Although this information is commonly limited to the component interfaces (IDL), some authors suggest that semantic information should be included in order to facilitate the search for reusable components [9]. The repository also gathers different documentation that helps to understand the functionality offered by the component and its context of use.

In order for a component to be included in the repository it should be reliable, easy accessible, appropriately documented, self-content and independent of the application for which it had been created, it should also use interfaces and standard architectures and should be parametrical and sufficiently proven. To guarantee these conditions additional effort during the development of the component is necessary, but this effort will be rewarded in later developments when that component is reused.

It has to be pointed out that it is not always possible to adapt a component. For example, the supplier company of the component might not have been able to produce a component that can be adapted by the client or it might even be possible that the company does no longer exist [14]. However it might be worthwhile reusing a component when it is easier to find, understand and adapt an existent component than to develop a completely new component from scratch.

Sametinger proposes an eclectic focus with a spiral model that includes the following phases [12]:

1. Identifying the objectives and finding the components that might be useful for the new design (*design with reuse*). Considering the convenience of each component and organizing one or several solution outlines that are based on the functionality of the suitable components.
2. Deciding how to implement the system by taking into account which components can be reused and what adaptations have to be made. This decision is based on a study of the alternatives that have been identified in the previous phase. The required effort to carry out the adaptations and the risks that the use of each component involves are evaluated. In order to minimize risks, the designer studies the available information on the component and experiments with it.
3. Implementing the system by carrying out the adaptations identified in the previous phase and incorporating the modified components and the new components to the system.
4. Planning the next phases. The adapted components and the new components are evaluated in order to include them in the repository.

It has to be pointed out that the previous approach merges reused, adapted and new components. For the development of latter ones any of the methods discussed in the previous sections can be used. In this case, the possibility of components overlapping each other has to be taken into account.

For example, supposing that in the use case based approach, described in section 3.1, one of the created components includes a class that forms part of a reused component. If the adaptation of the reused component is not possible, the *separating* and *gathering* alternatives cannot be used. If the component in question provides all services required by the use case, the *concentrating* alternative can be chosen. Otherwise, only the *distributing* alternative with the inconveniences

[6] More precisely, what cannot be detached from the component are the operations and data associated with the changes that have determined the grouping of the classes in that component.

pointed out in section 3.1 can be used. If, on the other hand, the adaptation of the reused component is possible, the partition proposes different constraints. For example, if the adaptation admits the implementation of the whole use case to be included in the component, the *gathering* option could be chosen. In general any of the partition alternatives can be chosen as long as the foreseen adaptation mechanism allows it.

4. Conclusions

This work reviews the different partition proposals found in the literature. A common case study has been used as comparison framework. It must be emphasized that some proposals do not deal with all cases. For example, the partition by use cases does not outline how the classes taking part in more than one use case are distributed among the created components. This work sets out to fill these gaps in order to allow a more complete evaluation of the final result.

It would be interesting to have a standard to which all proposals could be compared because each proposal gives priority to a different aspect. Thus, the approach based on the foreseen evolution improves the maintenance of the resulting system. On the other hand, the partition by preexistent components improves, *a priori*, the productivity.

However, many aspects of the approaches discussed here can be subjective, and difficult to measure. The only observable parameter is the resulting architecture, or to be precise, the granularity of the obtained components. This standard can serve as a base to measure other more subtle aspects such as reuse, maintenance or productivity of the approach. Table 1 shows a comparison of the studied methods, with regards to the granularity of the resulting components.

Use cases	**Design patterns**	**Business domains**	**Foreseen evolution**
++	+	+++	+++

Table 1. Granularity of the resulting component in each approach

The component created for a use case might need more than one design pattern. Therefore, its granularity will be generally bigger than the granularity of a component that has been created for a design pattern. Similarly, for a business domain that has several use cases, the resulting component will be bigger than if it has only one use case. The components created by the foreseen evolution approach cannot be compared with the other approaches in a direct way. However, merging components that include the same change can give rise to potentially very big components.

References

[1] P. Allen and S. Frost. *Component-Based Development for Enterprise Systems. Applying the SELECT Perspective^{TM}*. Cambridge University Press, 1998.

[2] B. Berkem. “Traceability management from business processes to use cases with UML. A proposal for extensions to the UML’s activity diagram through goal-oriented objects”. *Journal of Object-Oriented Programming,* pp. 29-34, Sept. 1999.

[3] A.W Brown. “Moving from Components to CBD. Supporting distributed computing paradigms”. *Component Strategies*, pp. 23-28, Apr. 1999.

[4] D. D’Souza and A. Wills. *Objects, Components and Frameworks with UML. The Catalysis approach*. Addison-Wesley, 1999.

[5] M. Fowler. *Analysis Patterns. Reusable Object Models*. Addison-Wesley, 1997.

[6] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] D.C. Hay. *Data Model Patterns. Conventions of Thought*. Dorsert House, 1996.

[8] C.L. Hoover and P.K. Khosla. “Analytical partition of software components for evolvable and reliable mems design tools”. *Int. Journal of software Engineering and Knowledge Engineering,* vol. 9, no. 2, pp. 153-171, 1999.

[9] I. Iribarne and A. Vallecillo. “Una metodología para el desarrollo de software basado en COTS”. *Actas del primer taller de trabajo en ingeniería del software basada en componentes distribuidos* 2000. Technical report of Univ. of Extremadura TR 12/2000.

[10] I. Jacobson, M. Griss and P. Jonsson. *Software Reuse. Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[11] G. Larsen. “Designing component-based frameworks using patterns in the UML”. *Communications of the ACM*, vol. 42, no. 10, pp. 38-45, Oct. 1999.

[12] J. Sametinger. *Software Engineering with Reusable Components*. Springer, 1997

[13] C. Szyperski. *Component Software. Beyond Object-Oriented Software*. Addison-Wesley, 1998.

[14] J. Voas. “Maintaining Component-Based Systems”. *IEEE Software*, pp. 22-27, July/August, 1998.