# Design Issues in a Component-based Software Product Line

**Paula M. Donegan*, Paulo C. Masiero**

Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo (USP)
Caixa Postal 668 – 13.560-970 – São Carlos – SP – Brazil
{donegan,masiero}@icmc.usp.br

***Abstract.*** *A software product line to support urban transport systems is briefly described and the design of two of its features is discussed. Different solutions based on components are shown for these two features and their variabilities. In particular, an analysis is made of how their design is influenced by the development process adopted, by the decision to use black-box (off-the-shelf) components or white-box components that may be created or adapted depending on application requirements, and by the decision of automating or not the composition process. Additionally, alternatives for deciding how to define iterative cycles and increments of the product line are discussed.*

## 1. Introduction

A software product line (SPL) consists of a group of software systems sharing common and managed features that satisfy the specific needs of a market segment or a particular objective and are developed in a predefined manner given a collection of core assets [Clements and Northrop, 2002]. The design of an SPL can use various software design techniques that facilitate reuse, such as object-oriented frameworks, components, code generators, design patterns, features diagrams and aspect-oriented languages. Several papers emphasize the difficulty of gathering, representing and implementing variabilities in the context of SPLs [Bachmann et al, 2004; Becker, 2003; Bosch et al, 2001, Junior et al, 2005]. Variability in an SPL differentiates products of the same family [Weiss and Lai, 1999].

This paper has two main objectives: to illustrate different solutions based on components to represent variabilities of an SPL and to discuss how these solutions are influenced by the adopted development process, by the decision to use black-box or off-the-shelf (COTS) components (without access to the source code) which are reused as they are or to use white-box components (with access to the source code) which may be created or adapted according to application requirements, and by the decision of automating the composition process. The solutions are presented in the context of an SPL (being developed as an academic project by the authors) to simulate support of urban transport systems.

The organization of the paper is as follows: Section 2 briefly describes the SPL; Section 3 presents some generic alternatives for the iterative and incremental development of an SPL; Section 4 summarizes the process used for the SPL; Section 5

* With financial support of FAPESP (*Fundação de Amparo à Pesquisa do Estado de São Paulo*)

discusses design of components for two features of the SPL; Section 6 presents some conclusions.

## 2. The Software Product Line to Control Electronic Transportation Cards

The SPL used as an example in this paper concerns management of electronic transport cards (ETC) named ETC-SPL. These systems aim to facilitate the use of city transport, mainly buses, offering various functionalities for passengers and bus companies, such as use of a plastic card to pay fares, automatic opening of barrier gates, unified payment of fares, integration of journeys and supply of on-line travel information to passengers.

The software allows the integration and automation of the transport network, with a centralized system that maintains the data of passengers, cards, routes, buses and journeys. The buses are equipped with a validator that reads a card and communicates with the central system (for example using RFID – Radio Frequency Identification) to debit the fare on the passenger's card. There may also be a bus integration system that permits the user to pay a single fare for multiple trips. In addition, passengers can go on-line and look up their completed trips and card credit.

The system domain was analysed and the ETC-SPL is being designed with the objective of generating at least three applications (or products) based on the analysis of three existing ETC systems in Brazilian cities: São Carlos (São Paulo), Fortaleza (Ceará) and Campo Grande (Mato Grosso do Sul).

## 3. Development Process of Software Product Lines

The literature describes various processes for the development of an SPL [Gomaa, 2004; Atkinson et al, 2000]. In general, they recommend that an organization wanting to develop an SPL has developed at least three similar applications belonging to the same domain [Roberts and Johnson, 1998; Weiss and Lai, 1999]. The evolution of an SPL may be proactive (ad hoc) or reactive (planned) [Sugumaran et all,2006]. An intermediate approach, called extractive, occurs when a second or third application is developed, parts of the code of one or more of the existing software products are generalized in such a way that they can be reused, until at a certain moment all the code is refactored so that new applications are capable of reusing a substantial part of the core assets.

In the case of a proactive evolution, the organization can use a process based on reverse engineering or forward engineering that differ basically in their first phase, as proposed by Gomaa (2004). In the process based on reverse engineering, artifacts of analyses, such as use cases and conceptual models, are recreated from existing systems. In the process based on forward engineering the same artifacts are derived from various sources, such as existing requirements documents and processes for requirements capture. From this point on, both processes are similar and domain analysis considers the use cases which are common to all applications of the domain, constituting the kernel of the SPL, and those which are optional (existing only for some of the SPL products) or alternative (choosing from a set of possibilities). A general conceptual model is created representing the common and variable parts. Afterwards a features diagram can be developed to synthesize the common and variable parts of the SPL.

There are several models of processes for SPL development, all beginning with the domain analysis phase described superficially in the previous paragraph. One alternative is then to elaborate the design for the entire modeled domain. The implementation can be done afterwards, in one version only or in various partial increments. This alternative seems to be uneconomic and complex [Gomaa, 2004; Atkinson and Muthig, 2002].

Another option is to follow a more agile iterative and incremental process, in which the SPL is first designed and implemented in a version that contains only kernel features, and then incremented by the design and implementation of subgroups of optional and alternative variabilities, as proposed by Gomaa (2004). The SPL is based on components and variabilities of several different mechanisms such as inheritance, extensions (e.g. the strategy design pattern), configuration, template instantiation and generation can be implemented [Bosch, 2000].

The choice of increments to be produced in each iterative cycle can be done horizontally or vertically and this has a great influence on the design of the SPL architecture and on the components that implement variabilities, as is shown in Section 5. The horizontal increments are planned by including a subgroup of features that attend to a specific application but do not necessarily contain all possible variabilities of each feature included in the increment. The vertical increments implement, in a general and complete way, all the variabilities of a subgroup of chosen features, but do not necessarily produce a specifically desired application. Using the ETC-SPL as an example, a horizontal version could be one that would generate the ETC system for the city of São Carlos. A vertical version for the ETC-SPL would be an SPL containing all the possible forms of journey integration specified during the domain analysis. These possibilities are shown schematically in Figure 1.

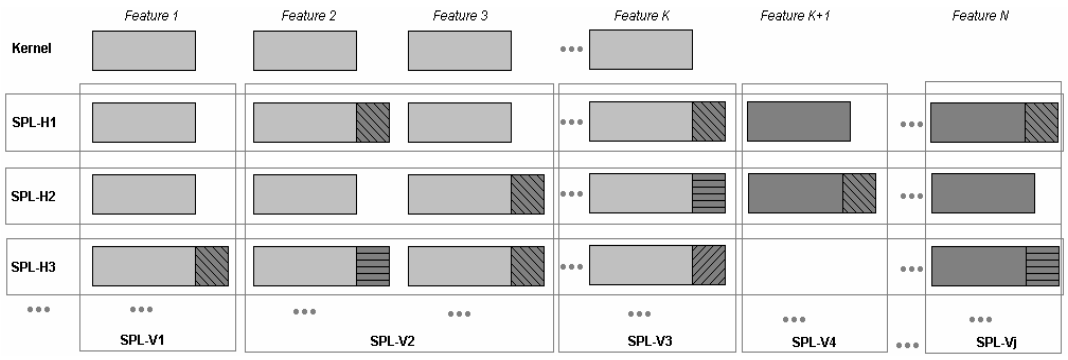


**Fig. 1 – Vertical and horizontal increments**

The behavior of variabilities in horizontal versions is shown in Figure 1 by the different shadings of variabilities extending a basic feature contained in the kernel. The figure illustrates, for example, features that do not appear in the kernel and do appear in a later version, features that appear in the kernel and are extended in one way for one version and in a different way for another version, etc. With the adoption of an evolving process such as that shown in Figure 1, each variability needs a careful design, because it may require refactoring in later versions. In other words, a design that is adequate for one version may not be so for a later version.

Horizontal increments are more realistic economically, in the sense that the SPL evolves as new applications need to be incorporated to the line, even though they can require more rework as the line evolves. On the other hand, vertical increments, even when not producing a previously foreseen application after the first iterations, have the advantage of allowing each chosen feature to be analysed and designed globally, including all its variabilities for the domain.

Another important decision to be made is how to develop the applications during the phase of application engineering, either using a manual process of generation of components that implement the SPL (in this case they correspond to the software assets available) or using an automated process, for instance a code generator. A solution that is adequate for a manual increment before automation may not be the best for a generative process. Thus it seems best to analyse all specified variabilities of a certain feature, before committing to a definite automated solution. However, if the SPL is composed mainly of black-box (off-the-shelf) components they impose an important restriction that leads to a solution using component composition and automatic generation of glue code.

Specifically for the ETC-SPL, we started with the specifications of three known products, the ETC systems of Fortaleza, Campo Grande and São Carlos, which were the applications to be generated initially. We considered it important to have a complete application early on, therefore opting to use horizontal iterative cycles in which each increment allows generation of one of these applications.

## 4. Development of the ETC-SPL: Iterative Cycles and Kernel Architecture

The points observed in the previous section were taken into consideration and PLUS (Product Line UML-based Software Engineering) [Gomaa, 2004] was used for the development of the ETC-SPL. The SPL Engineering was divided into two phases: in the Inception phase domain analysis yielded initial use cases, the feature diagram and a conceptual model, among other artifacts; for the Elaboration phase five iterations have been planed, each one producing a version of the ETC-SPL:

- Iteration 1: Comprising only features of the kernel.
- Iteration 2: Version 1 + features and variabilities of the application of Fortaleza.
- Iteration 3: Version 2 + features and variabilities of the application of Campo Grande.
- Iteration 4: Version 3 + features and variabilities of the application of São Carlos.
- Iteration 5: Version 4 with all variabilities but automatically generated with an Application Generator.

The features diagram for the kernel of the ETC-SPL (common features) is presented in Figure 2 using the notation of Gomaa (2004) and Figure 3 shows the architecture of the ETC-SPL kernel. The architecture is composed of three distributed modules, of which there is one occurrence of the server module (*ETCServer*) and various occurrences of each of two client modules (*Bus* and *WebAccess*). Internal to these modules are the components of the SPL, derived following the processes

suggested by Gomaa (2004) and by Cheesman and Daniels (2001), the latter used specifically for the identification of business and system components. The design based on components of some variabilities of the ETC-SPL is presented and discussed in Section 5. The additional features of the ETC systems of each city are the following:

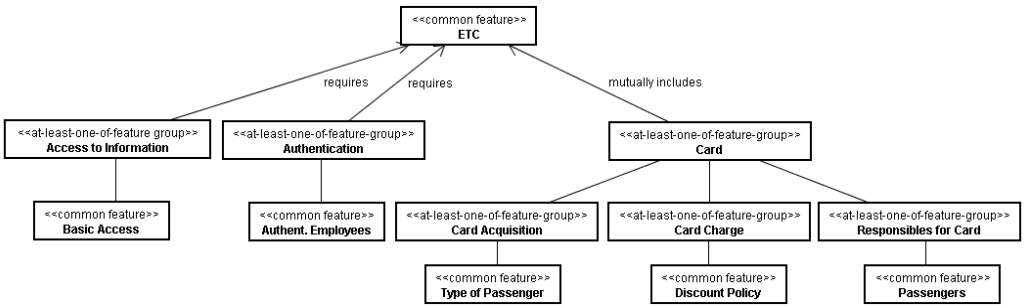


**Fig. 2 – Features diagram for the kernel ETC-SPL**

- **Fortaleza**: Form of Integration (Transport Hub), Card Payment, User Companies.
- **Campo Grande**: Additional Access, Form of Integration (Integration (Time, Integration Route, Number of Integration Trips), Transport Hub), Card Restriction (Number of Cards), User Companies.
- **São Carlos**: Additional Access, Passenger Authentication, Form of Integration (Integration (Time, Integration Route)), Card Restriction (Card Combination), Trip Limit.

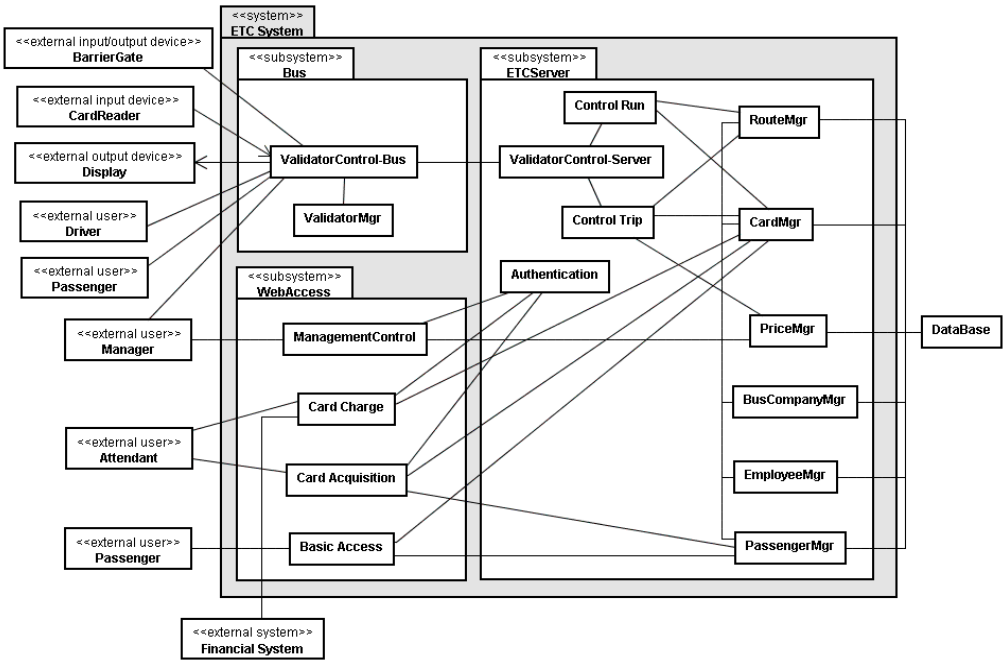


**Fig. 3 – Kernel architecture of the ETC-SPL**

## 5. Design Decisions for the Features of Forms of Integration and Card Payment

Two features of the ETC-SPL are discussed with the objective of illustrating how design decisions are influenced by the decisions taken related to the SPL development process adopted, to the type of component, and to the manner of composition (manual or automated). One feature (*Form of Integration*) uses new classes to model and implement its feature and another feature (*Card Payment*) uses subclasses (with new attributes and methods) to do so. For simplicity and space reasons, the models of classes that are illustrated show only the attributes.

### 5.1 Design of features related to "Forms of Integration"

We initially will consider the optional feature *Transport Hub* exclusive to the cities of Fortaleza and Campo Grande, which are considered in version 2 of the ETC-SPL. Figure 4 shows part of the features diagram related to this feature. The ETC system of Fortaleza has only bus transport hubs as a form of trips integration. The transport hubs work as a special terminus where passengers can change buses without paying another fare. Other more sophisticated ways of integration occur in the ETC systems of the other two cities, corresponding to other variabilities of the *Form of Integration* feature group.

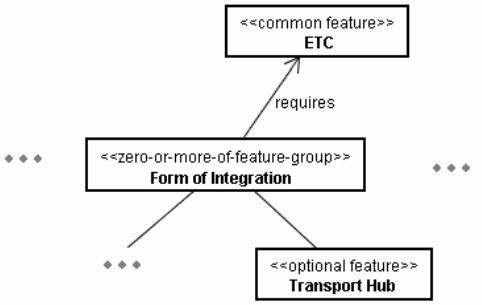


**Fig. 4 – Part of the features diagram related to the *Transport Hub* feature**

Figure 5 shows the model of classes used to implement the operations related to the routes of the bus company. The classes *Route*, *Run*, *TransportMode* and *Bus* (represented with the stereotype[*] <<*kernel*>> in the figure) are wrapped in a kernel component called *RouteMgr*, represented previously in Figure 3. The design of the feature *Transport Hub* requires the addition of a *TransportHub* class to the model. Generally, the inclusion of new features to the SPL design implies adding and/or modifying classes, operations and attributes. In the same way, the components may need adaptations or compositions such that variabilities reflect on the components' architecture. There are many ways of treating these changes, each having advantages and disadvantages that reflect on the decisions taken for the SPL's design.

One way of treating the inclusion of operations and attributes inside existing classes and the inclusion of new classes is to add them directly inside their components and change operations according to new needs. For the given example, there should be two components, the kernel component *RouteMgr* and the alternative equivalent

[*] Gomaa (2004) sometimes uses more than one stereotype to classify more specifically elements of a SPL.

component, which could be called *RouteTransportHubMgr*. They would be used respectively for the kernel application and the application of Fortaleza.

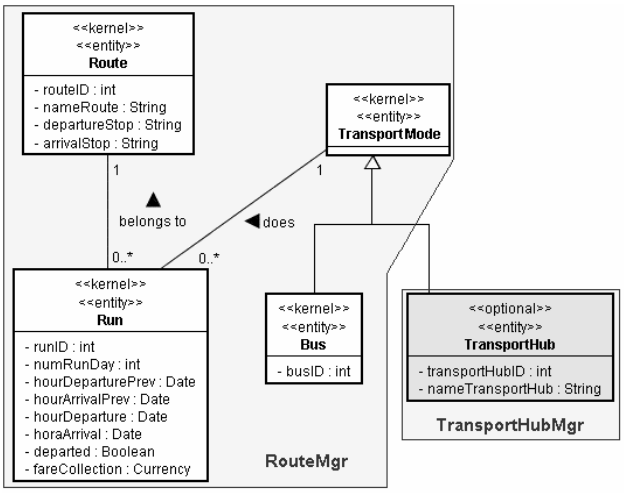


**Fig. 5 – Fragment of the class model related to the *Transport Hub* feature**

This solution's advantage is its facility of implementation and composition. There are, however, many disadvantages. There can be future problems with maintainability because the solution tends to duplicate code and a future modification can demand the update of both alternative components. Besides, the original component, *RouteMgr*, has to be a white-box because its adaptation requires the knowledge and access of its internal structure.

To include classes, operations and attributes without having internal access to the implementation of previously developed components, corresponding to the SPL's assets, it is necessary to design black-boxes. Therefore, to design the ETC-SPL, we preferred to use these kind of components and, in this way, operations and attributes that would be added to existing classes are separated into new classes so that new variability specific components are created. These components can then be assembled with components already designed to create new ones satisfying the new requirements. The components differ because they implement distinct variabilities, however they can be formed by kernel components that are reused. The way in which they are connected and the required implementation to join these different components can also be changed. The disadvantage of this solution is a greater communication between components, which can decrease efficiency.

Consequently, instead of including the class *TransportHub* inside the component *RouteMgr*, a new component is created for the class with its attributes and operations, called *TransportHubMgr* and the component *RouteMgr* is reused without any alteration. The use of these components is managed by another component (*Controller RouteTransportHubMgr*) and the three components are then wrapped in a composed component called *RouteTransportHubMgr*. These components' architecture details are shown in Figure 6. The interfaces are not altered and the components requiring them do not need any modifications. The interface *IRouteMgt* is required by the components

*Control Trip*, *Control Run* and *ManagementControl*, the interface *IUpdateRun* is required by *Control Run* and *ICollectFare* is required by *Control Trip*.

In the Campo Grande version, besides the *Transport Hub* feature, there is also the feature *Integration*. The integration can be done using defined integration routes, provided that the trip remains inside a specified time limit. There is also a maximum number of integration trips within the time interval allowed. These features can be seen in Figure 7.

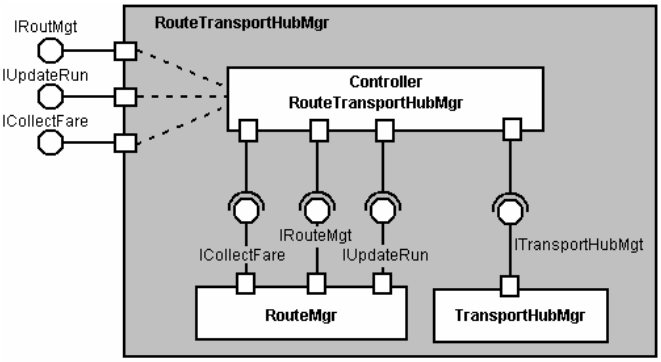


**Fig. 6 – Composed component *RouteTransportHubMgr***

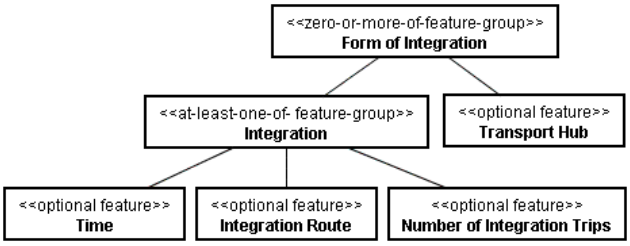


**Fig. 7 – Part of the features diagram related to the "*Form of Integration*" feature**

In the model of classes, the feature *Integration Route* is represented by an optional class called *IntegratedRoute* and is associated to the *Route* class, shown in Figure 8. Only this specific feature will be shown as an example, because the other features are found in different parts of the model and are not related to the *RouteMgr* component. Since Campo Grande also uses transport hubs, the component *RouteTransportHubMgr* can be reused for this version and, so that the components' integrity is maintained and to keep them as black-boxes, the optional *IntegratedRoute* class is integrated in a new component called *IntegratedRouteMgr*. Similarly as in the previous example, a controller component is created and the three components are then internal parts of the composed component *RouteTransportHubIntegrationMgr* seen in Figure 9.

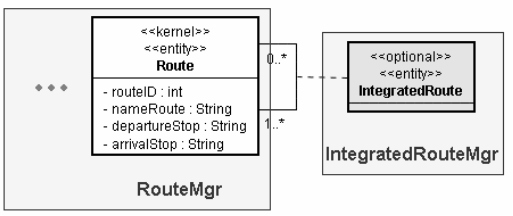


**Fig. 8 – The feature "*Integration Route*" in the class model**

This component has another interface as the corresponding components of the earlier versions do not have the *IVerifyIntegratedRoute* interface. With integration this interface is required by the *Control Trip* component which consequently also has to be altered to treat the provided result according to the business rules of the integration feature. The solution can be to use composition, designing a new component or to separate the additional interest in an aspect [Kiczales, 1996; Suvée et al, 2006] so that the component does not need to be replaced and there can be an enhancement in the process of the variabilities' composition of the SPL [Heo and Choi, 2006]. The other interfaces remain the same as in the Fortaleza version.

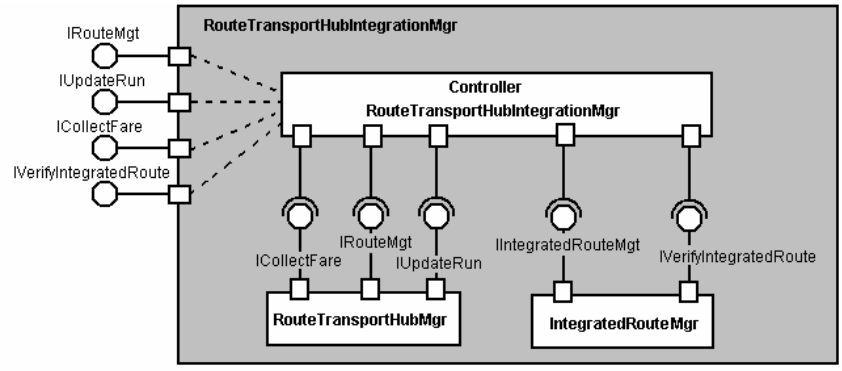


**Fig. 9 – Composed component *RouteTransportHubIntegrationMgr***

In the São Carlos version, the feature *Integration Route* also exists but does not have the *Transport Hub* feature. Therefore the components related to the transport hub cannot be reused and a new composition is needed. For this version the components that can be reused are *RouteMgr*, developed in version 1 (kernel), and *IntegratedRouteMgr*, developed in version 3 (Campo Grande). A new controller is necessary to compose these components and form the composed component *RouteIntegrationMgr*, whose architecture can be seen in Figure 10.

## 5.2 Design of features related to "Card Payment"

In the ETC system of Fortaleza, some passenger types have to purchase the bus card. This feature does not exist in the other two cities and is designed in the iteration of version 2, not reflecting in other iterations. This feature is shown in the partial features diagram of Figure 11. The cards can be of different categories according to the type of passenger and information about passenger trips may be stored. A card can also have various associated payments related to charges made for the card. When the feature

*Card Payment* is present, the payment can also refer to the purchase of a card. These requirements lead to the classes' model presented in Figure 12.

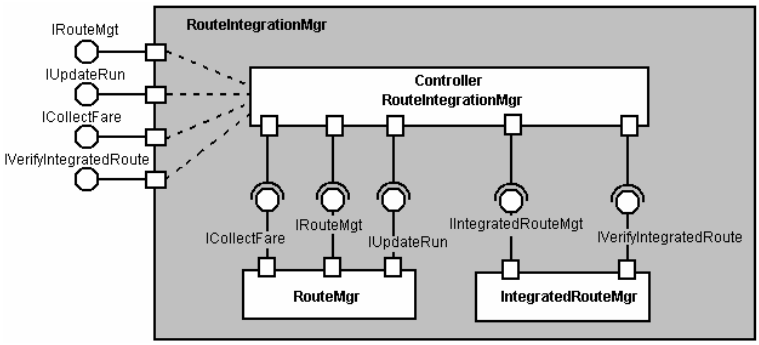


**Fig. 10 – Composed component *RouteIntegrationMgr***

Figure 12 shows the classes *Card*, *PassengerType*, *Trip* and *Payment* that are part of the ETC-SPL and are encapsulated in the *CardMgr* component. The *Card Payment* feature implies variation points in the classes *PassengerType* and *Payment*, altering attributes and operations of these classes, different to the previous example, in which it was necessary to insert a new class into the model. One option is to use parameterized classes, but this option was not adopted to keep interests separated [Gomaa, 2004] so as to maintain the components as black-boxes. We chose then to use classes with variation points and separate the *Card Payment* feature in a new component called *PaymentMgr* with the *IPaymentMgt* as a provided interface. The specification of this interface is shown in Figure 13. Both classes are inserted in one component because they have the same interest (*Card Payment*) and are always used together. If it was important to differentiate them, two interfaces could be provided by the component, separating the methods implemented by different classes.

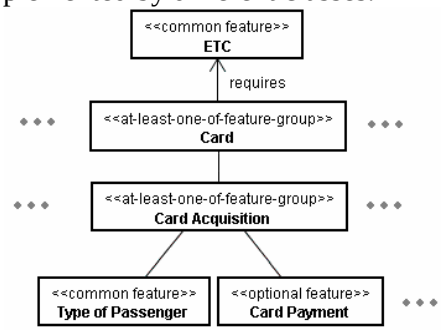


**Fig. 11 – Part of the features diagram related to the *Card Payment* feature**

The components *CardMgr* and *PaymentMgr* are managed by a controller so that the information of the passenger type and the payment remain separated in different components and in different tables in a relational data base and can be treated and joined if needed. Their composed component is called *CardPaymentMgr* and its architecture is shown in Figure 14. The interfaces of the composed component are not changed and therefore it is not necessary to change the components that require the interfaces of this component, for example the *Control Trip* component.

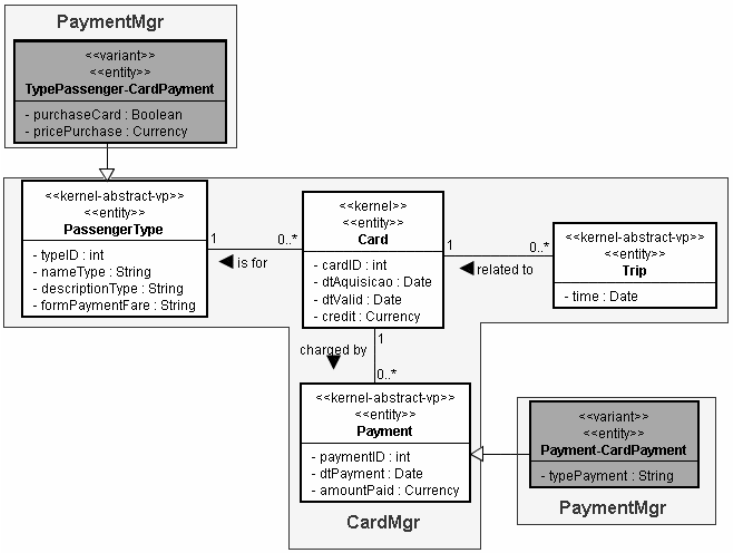


**Fig. 12 – Part of the classes' model related to the "*Card Payment*" feature**

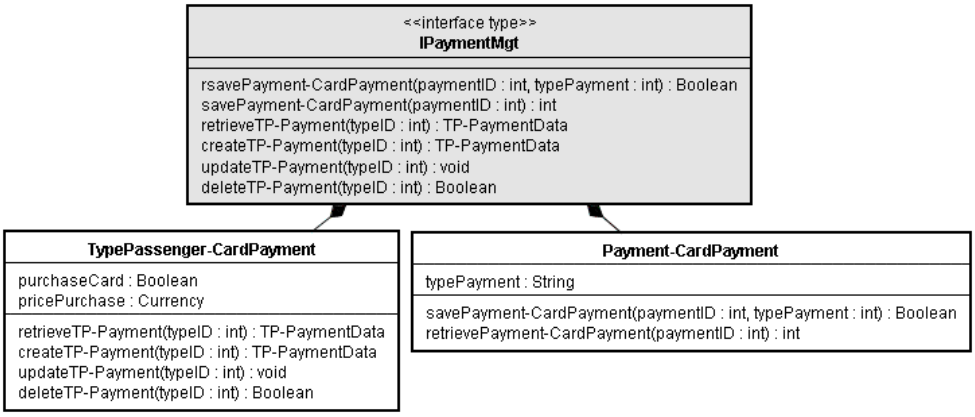


**Fig. 13 – Specification of the *IPaymentMgt* interface and related classes**

Independently of this composed component, the feature *Card Payment* needs a modification in the *Card Acquisition* component so that it records the reception of a payment in the financial system by requiring the *IFinancialSystemMgr* interface from the *Financial System* component. This way the *Acquisition Card* component is substituted by a composed component that is specific for the *Card Payment* feature.

This example also shows how a white-box solution would be easily created by a generator by maintaining the component's name and its interfaces and, when needed, changing its composition by adding subclasses according to options chosen on the generator tool. The solution presented requires more communication between classes that implement the component's interfaces and can be more inefficient, but even so we chose this solution so that black-box components could be used. The controllers correspond to generated glue code to connect the lower level assets.

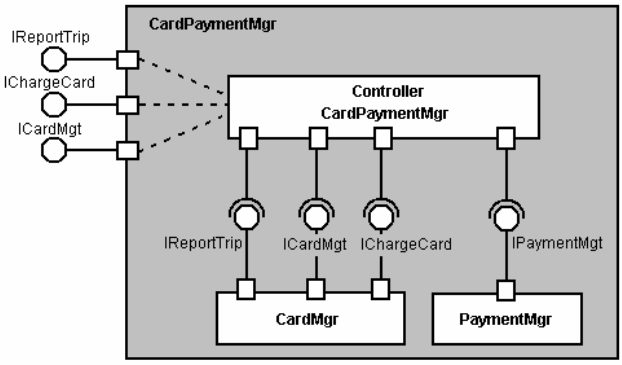


**Fig. 14 – Composed component *CardPaymentMgr***

## 6. Using a Code Generator

An automated process is planned to be implemented in iteration 5 to generate applications for this SPL. It is relatively easy to see that the list of features shown in Section 4 is an initial sketch of the Application's Modeling Language (AML) (Weiss and Lai, 1999) for the ETC domain and based on it an automated application generator can be created. We intend to use the configurable code generator Captor developed by our research group [Schimabukuro et al, 2006].

For the solution presented here, based on black-box components, the generator code will act like a "configurator", starting from the kernel architecture, replacing and including the necessary black-box components from the library of core assets and generating glue code for each pair of components being composed. Note that if the automation had been done before iteration 5, each new horizontal version designed and implemented would need considerable rework in the generator.

Another solution that could be used for this case, considering white-box components, would be to make the generator perform the changes inside each component thereby generating additional classes and modifying other elements inside the components according to the need of each application. The generator in this case would be much more complex and act as a composer, according to the definition of Weiss and Lai (1999). Thus, less core assets would be needed. Both solutions are acceptable, however, but depend on the previous decision of using black-box or white-box components. A combination of both is also possible.

The choice of automating the composition process influences the design as well as the moment of introducing the automation in the SPL. If automation is used to generate the products from the first version, this decision influences the design of the new versions of the SPL. Also, for each new horizontal iteration, considerable rework in the application generator would be needed.

## 7. Final Considerations

In the current state of development of the ETC-SPL the design of the kernel and of version 2 (Fortaleza) have already been made. Some other features have also been designed vertically with the intention of investigating different solutions to those shown

in this paper. The implementation of the kernel is ongoing, aiming to create the SPL's assets.

A lesson learned so far from the development of the ETC-SPL is that having decided to evolve the line in horizontal iterations, it is very important that some time be taken to analyse how feature groups will evolve in the following iterations before committing to a design that cannot be easily modified or reused in the next versions. The example discussed in this paper for the route integration feature has shown the trade-offs between horizontal and vertical development. It also showed that the decision of using black box or white box components is crucial. In general, it is better to design a black box component and compositions of them, as it is always possible later, if an off-the-shelf component is not found, that implements the required interface, to design one that performs the required function.

## References

Atkinson, C; Bayer, J.; Muthig, D. (2000) Component-Based Product Line Development: The KobrA Approach. *1st Software Product Line Conference*, 19p.

Atkinson, C; Muthig, D. (2002) Enhancing Component Reusability through Product Line Technology. *Proceedings of the 7th International Conference in Software Reuse (ICSR02)*, Springer Press, p. 93-108.

Bachmann, F.; Goedicke, M.; Leite, J.; Nord, R.; Pohl, K.; Ramesh, B.; Vilbig, A. (2004) A Meta-model for Representing Variability in Product Family Development. *Proceedings of the 5th International Workshop on Software Product-Family Engineering*, Springer, p. 66-80.

Becker, M. (2003) Towards a General Model of Variability in Product Families. *Proceedings of the 1st Workshop on Software Variability Management*, Groningen, 9p.

Bosch, J. (2000) Design et Use of Software Architectures: adopting and evolving a product-line approach, Addison-Wesley, 354p.

Bosch, J.; Florijn, G.; Greefhorst, D.; Kuusela, J.; Obbink, H.; Pohl, K. (2001) Variability Issues in Software Product Lines. *Proceedings of the 4th International Workshop on Product Family Engineering*, Springer, p. 11-19.

Cheesman, J.; Daniels, J. (2001) *UML Components: a simple process for specifying component-based software*, Addison-Wesley, 176p.

Clements, P.; Northrop, L. (2002) *Software Product Lines: Practices and Patterns*, Addison-Wesley, 563p.

Gomaa, H. (2004) *Designing Software Product Lines with UML*. Addison-Wesley, 701p.

Heo, S-H.; Choi, E. M. (2006) Representation of Variability in Software Product Line Using Aspect-Oriented Programming. *Proceedings of the 4th International Conference on Software Engineering Research Management and Applications (SERA)*, 8p.

Junior, E. A. O.; Gimenes, I. M. S.; Huzita, E. H. M.; Maldonado, J. C. (2005) A Variability Management Process for Software Product Lines. *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, Cascon, p. 225-241.

Kiczales, G. (1996) Aspect-Oriented Programming. *ACM Computing Surveys (CSUR)*, v.28, n. 4es, p. 220-242.

Roberts, D; Johnson, R. (1998) Evolving Frameworks: a pattern language for developing object-oriented frameworks. In: Martin, R.C.; Riehle, D.; Buschman, F. *Pattern Languages of Program Design 3*, Addison-Wesley, p. 471-486.

Schimabukuro, E. K. J.; Masiero, P. C.; Braga, R. T. V. (2006) Captor: A Configurable Application Generator (in Portuguese). *XIII Tools Session of the Brazilian Symposium of Software Engineering*, 6p.

Sugumaran,V.; Park, S.; Kang, K.C. (2006) Software Product Line Engineering. *Communications of the ACM*, Vol 49, No. 12, p. 29-32.

Suvée, D.; Fraine, B. D.; Vanderperren, W. (2006) A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development. In: *Component Based Software Engineering*, p. 114-122.

Weiss, D. M.; Lai, C. R. R. (1999) *Software product-line engineering: a family-based software development process*. Addison-Wesley, 426p.