Description of the FRS domain

by Stan Jarzabek and Yu Chye Cheong
Department of Computer Science, School of Computing, National University of Singapore
Singapore 117543

1. AN OVERVIEW OF THE FRS DOMAIN

An FRS helps in the reservation of facilities (such as tutorial rooms, hotel rooms, lecture theaters) and specific equipment. Different organizations such as academic institutions, hotels, hospitals and companies, have different physical facilities and arrangements for their reservation. Most often, *users* (individuals or organizations) manage their own reservations with an FRS (e.g., add, delete and modify their own reservations). In some companies, however, users send reservation requests to a middleman who makes reservations for them. An FRS may allow certain users to manage facilities (e.g., add new facilities). In general, users may only perform certain functions according to the permissions assigned to them. An FRS will typically allow users to print out a variety of reports. The calculation of reservation charges, if any, may be performed according to discounts associated with each user and the payment classification of each facility.

Below we list some of the FRS variants:

1. Different institutions have different physical facilities and different rules for making reservations.
2. FRS may need support multiple methods for viewing reservations. In addition to viewing by facility ID (mandatory), FRS may need support viewing reservations by reservation ID and/or by the date.
3. There is a high level of variability in user permissions. In some FRSes, permissions are given across the board to a whole group of users; at other times, there may be a need to individually specify the permissions that each user has.
4. Some FRSes may need manage a database of all the users and/or facilities in a company. One should be able to add new user/facility, enter the proper description and delete an existing user/facility.
5. Some FRSes should also allow the user to search for available facility (e.g., a meeting room) with the necessary piece of equipment.
6. Sometimes a payment must accompany the reservation. The calculation of reservation charges, if any, may be performed according to discounts associated with each user and the payment classification of each facility.

The FRS domain can be considered a sub-domain (i.e., specialization) of the general object allocation domain [28] In case of the FRS, an "object" is a facility, whether it is a room, equipment or even a person. The object allocation domain supports the allocation of an object to another object, which is usually an agent. The allocated objects are typically returned after a period of time for subsequent reallocation. Sub-domains of the FRS domain include offices, academic institutions, hotels, recreational and medical facilities and transportation (e.g., airlines, trains). These sub-domains are not mutually exclusive and the intersection of sub-domains forms a generic core of the FRS domain.

In practice, scoping a product line is done before detailed domain analysis and design of a product line architecture [29] During scoping, we identify areas of concern that are worth the domain engineering effort. These areas of concern may cover different dimensions of a product line such as functional requirements, platform dependencies, runtime concerns (such as component structure, component communication, synchronization and performance), system-wide qualities (such as security, reliability and availability) and many others. Having explicitly engineered a given area of concern within a product line architecture, we expect to efficiently manage variants within that area during system engineering and architecture evolution. Scoping a product line is driven by business factors - we should address a given area of concern only if expected benefits outweigh the domain engineering effort. In this paper, we do not precisely define or further justify the scope of an FRS product line from business perspective.

2. MANDATORY AND VARIANT REQUIREMENTS

A requirement is any characteristic of an application domain or a software system. There are a number of possible types of requirements that we model during domain engineering. A requirement may be a functional or quality requirement (e.g., computation of a salary, employee, system response time, reliability), may refer to design decisions (selection of a sorting algorithm or architectural style) and to the implementation (the choice of a database, programming language or platform). Clearly, in domain engineering, depending on the objectives determined during product line scoping, we may need to deal with any or all of the above variant requirements. Examples in this paper refer to user-level functional requirements, but described techniques apply to other types of requirements, too.

Some of the requirements can be enumerated. We give each such requirement a unique name, e.g., VIEW_RES_BY_FACILITY, for ease of reference. Following [14],[15],[22], we classify requirements as follows:

- *Mandatory Requirements.* Mandatory requirements appear in all the instances of a concept being described. The “concept” may refer to anything that matters in a product line, in particular, a real-world entity from the problem space, specific requirement, a group of systems in a product line, architecture element, design decision, code component, etc. For example, a functional requirement: “ability to add a new user” is mandatory for those FRSes that are expected to maintain a user database, while a requirement: “ability to maintain a record of reservation details” is a mandatory requirement for all FRSes.
- *Variant Requirements*. Variant requirements (or variants for short) appear in some of the instances of a concept being described. Variants are further qualified as optional, alternative and or-requirements. An example of an optional requirement is payment option and other related requirements concerning computation of discounts, cancellation fee, etc. Some of the optional requirements in the FRS domain are listed in Table 1. An alternative describes one-of-many requirements. For example, an FRS may use either command line or graphical user interface. An or-requirement [22] describes any-of-many requirements. For example, an FRS may support any of reservation viewing methods such as by facility or by date. In textual descriptions, we use the naming convention proposed by Tracz [14]: we add a suffix “–OPT” to the requirement’s name for optional requirements, suffix “–ALTn” to indicate alternative requirements and suffix “-ORn” to indicate or-requirements. If the number of alternatives cannot be determined, the suffixes “–ALT” and “-OR” are used. A requirement without a suffix is considered to be mandatory.

Optional Requirement	Description
BILL_CONSTRUCTION–OPT	The system provides the capability to personalize the bills for a reservation.
CANCELLATION_FEE–OPT	The system shall charge a certain amount whenever a reservation is cancelled within “n” days of the reservation.
CONFLICT_RESOLUTION–OPT	In the event of a conflict[1], the system shall perform some form of conflict resolution.
PAYMENT–OPT	The system provides the capability to handle payment for reservations.
VIEW_RES_BY_DATE–OPT	The system shall produce a list of all the reservations for a specified user.
SUGGEST_ALTERNATIVE_FACILITY–OPT	When a conflict arises (i.e., requested facility has been already reserved for the time period required), then the system shall suggest alternative facilities which are available during the time period required.
RETRIEVE_AVAILABLE_FACILITIES–OPT	Given a particular time period, the system shall produce a list of facilities available during this time period.

Table 1. Examples of optional requirements in FRS domain

We further classify variant requirements as construction-time and runtime variants.

- *Construction-time variant requirements* are addressed during customization of a product line architecture, in the customize-compile-run cycle. Typically, the choice of user interface for an FRS (graphical or command line) or different ways of viewing reservations (by reservation number or by date) would be construction-time variants.
- *Runtime variant requirements* are implemented within a given system and a user can choose a variant of his/her choice during system execution. In FRSes that require a flexible way of assigning reservation permissions to different users and roles, permissions exemplify variants that should be dealt with during runtime.

The decision whether a given variant is to be addressed during construction-time or runtime requires careful analysis, taking both user needs and technical issues (e.g., difficulty of supporting given variants during runtime) into account. Other biding times for variants include program activation time and location.

Mandatory and variant requirements are often depicted in the form of feature diagrams. FODA, Feature-Oriented Domain Analysis [15], was developed at SEI. We use feature diagrams with extensions described in [22]. In FODA, requirements are called features and in this section we use terms *feature* and *requirement* as synonyms.

In a feature diagram, requirements are organized into a tree structure. Different types of variants are depicted using distinct graphical symbols, as shown in Figure 1.

[1] A conflict arises if a new reservation or a modification to an existing one would result in a clash with other reservation(s).

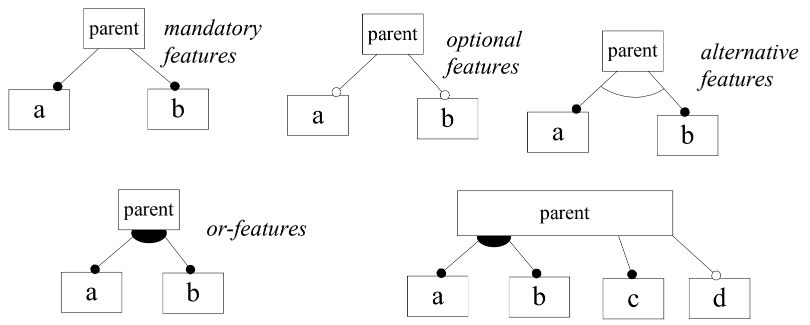
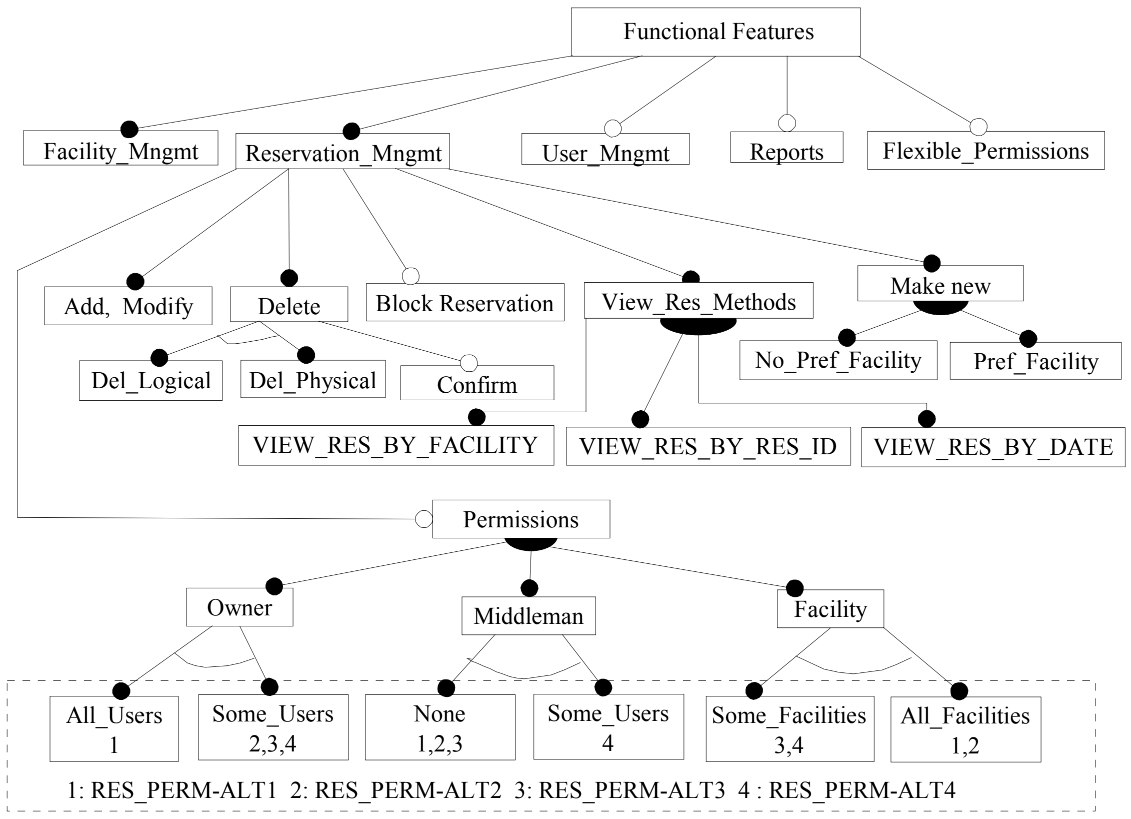


Figure 1. Feature modeling conventions



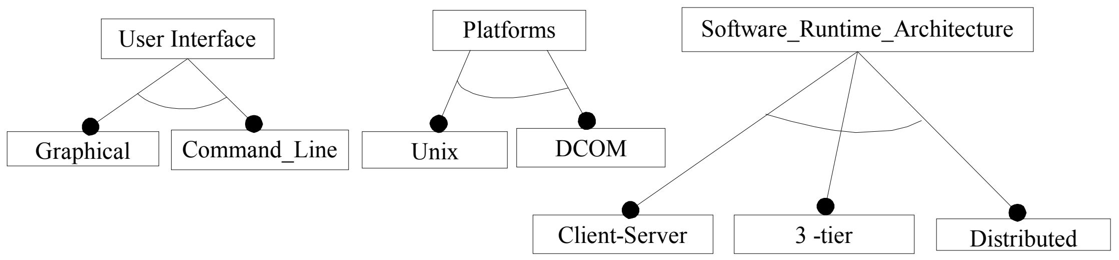


Figure 2. Excerpts from the FRS feature diagram

Figure 2 depicts some of the variants in the FRS domain. Under "Functional Features" we find the "Reservation_Mngmt" group with "Delete" feature defining two modes of deleting a reservation, namely logical deletion and physical deletion. Logical deletion ("Del_Logical-ALT1" alternative) merely marks a reservation as deleted but does not delete the reservation record from the database. Physical deletion ("Del_Physical-ALT2" alternative) deletes the reservation record from the database. Also, under "View_Res_Methods" group, we see a mandatory reservation viewing method, namely VIEW_RES_BY_FACILITY, and two other possible methods, namely VIEW_RES_BY_RES_ID-OR1 and VIEW_RES_BY_DATE-OR2 (we capitalized names of those requirements that we shall refer to later in the paper). Notice that in feature diagrams we can skip suffixes (OPT, ALT and OR) as they are implied by graphical conventions of the feature model. However, for clarity, we append suffixes to variant names in textual descriptions (e.g., VIEW_RES_BY_RES_ID-OR1).

There is a high level of variability in the nature of permissions granted to various users of an FRS. In some cases, permissions are given across the board to a whole group of users; at other times, there may be a need to individually specify the permissions that each user has. This is further complicated by the presence of middlemen in some FRSes. Some FRSes may require a middleman to vet certain user actions, such as adding reservations. In other FRSs, users may be able to make reservations directly without any middlemen. through a specific middleman. In the FRS feature diagram, optional "Flexible_Permission" feature indicates the need for a comprehensive permission control mechanism, in which one can specify exactly what actions, which users can perform. Typical - simpler but less general - default alternatives related to reservation permissions are described below:

RES_PERM–ALT1: Any user can manage reservations for any facility
RES_PERM–ALT2: Specific users can manage reservations for any facility
RES_PERM–ALT3: Specific users can manage reservations for specific facilities
RES_PERM–ALT4: The system shall allow specific users to manage reservations for specific facilities

"Permissions" option under the "Reservation_Mngmnt" feature group (Figure 2) shows how the above defaults.

A feature model covers both a problem space (functional features) and a solution space (program features, design and implementation). The former includes reservation viewing methods and the latter the choice of user interface and runtime architecture. While it is possible to distinguish parts of a model related to different aspects of problem and solution space, we still must highlight the many dependencies among common and variant points in different parts of the model. The concept of a domain in software engineering often embraces both a problem space and software systems that solve a problem, as it appears difficult or even counterproductive to make a clean separation between the two.

3. DEPENDENCIES AMONG VARIANT REQUIREMENTS

Feature diagrams model structural dependencies among variants. Usually, variants display more complex dependencies that cannot be captured in feature models. For example, a certain variant can hold in any given system only if some other requirement also holds. For example, a variant requirement BILL_CONSTRUCTION–OPT can only hold in a system in which reservations are accompanied by payment (PAYMENT–OPT). We say that requirement BILL_CONSTRUCTION–OPT depends on requirement PAYMENT–OPT. We can express legal combination of variants in any given member of a product line as constraints using logic notation, for example:

BILL_CONSTRUCTION–OPT $\Rightarrow$ PAYMENT–OPT

Implication defines the intended meaning of requirement dependency. In the right-hand-side column of Table 2, we listed types of requirement dependencies that we found useful to model. The left-hand-side column of the same figure shows a notation we use to model requirement dependencies. In Figure 3, we give examples of requirement dependencies, referring to optional requirements listed in Table 1.

Notation	What it means
$R_1 \Rightarrow R_2$	R_1 can hold only if R_2 holds
$R_1 \Rightarrow !\, R_2$	R_1 can hold only if R_2 does not hold
$R_1 \Rightarrow\ < R_2 \mid R_3 \mid \ldots \mid R_N >$	R_1 can hold only if any (i.e., at least one) of the requirements $R_2, \ldots, R_N$ holds
$R_1 \Rightarrow <R_2\ \&\ R_3\ \&\ \ldots\ \&\ R_N>$	R_1 can hold only if all of the requirements $R_2, R_3, \ldots, R_N$ hold
[R1, R2, …, R_N]	Either all of the Requirements $R_1, R_2, \ldots, R_N$ hold or none of them holds

Table 2. Types of requirement dependencies

BILL_CONSTRUCTION–OPT $\Rightarrow$ PAYMENT–OPT

CANCELLATION_FEE–OPT $\Rightarrow$ PAYMENT–OPT

SUGGEST_ALTERNATIVE_FACILITY–OPT $\Rightarrow$
CONFLICT_RESOLUTION–OPT & RETRIEVE_AVAILABLE_FACILITIES–OPT

Figure 3. Examples of requirement dependencies

Feature diagrams do not capture the meaning of product line requirements. To convey the domain's semantics, we must use other notations such as UML [16]. However, most of the conventional notations cater for modeling a single system and provide only limited support for modeling variants (such as extension/inclusion points in use cases or class inheritance). Jacobson, Martin and Jonsson [17] formulated the problem of incorporating variants into any modeling notation and analysis component (i.e., any reusable analysis element). Variants are exploited at variation points in analysis components. A variation point identifies one or more locations at which the variation will occur. An analysis component is customized (specialized) by attaching one or several variants to its variation points. Authors describe a number of variability mechanisms for modeling variants and customizing components such as inheritance, macros, templates, parameterization and frames. They discuss different ways of describing variants in UML notations, in particular use cases.

In our projects, we model variants using feature diagrams first and then use UML notations extended with provisions for modeling variants, to enhance the semantics of variants. Modeling variants in extended UML and integrating different model views into a cohesive picture is a difficult task, especially when the number of legal combinations of inter-dependent variants increases. We attempted to tackle this problem with tools that help trace variant dependencies and produce customized views of a domain model for given variant configuration. We refer the reader to other papers for details of our domain modeling method [30] and focus this paper on handling variants in a product line architecture.

4. IMPLEMENTATION ASSUMPTIONS AND A FRS RUNTIME ARCHITECTURE

A product line architecture is developed under design, implementation and runtime architecture constraints. An example of a design constraint is the requirement that the FRS supports a GUI (Graphical User Interface) as opposed to a command-line interface. An example of an implementation constraint is a requirement that the FRS be implemented in a particular programming language or run on a distributed component platform (such as J2EE or .NET). A runtime software architecture is described by a set of components that interact one with another through well defined interfaces (connectors) to deliver the required system behavior [4]. The runtime architecture has impact on a product line architecture so it should be considered first.

Our FRS runtime architecture adopts a three-tiered architectural style depicted in Figure 4 and is implemented in EJB™. In the architecture terminology, a 'tier' is just a large granularity component that may contain many smaller components. Each tier provides services to the tier above it and serves requests from the tier below. The client (user interface tier) provides the means by which the user of the FRS interacts with the system. The business logic tier provides the functionality of the FRS. The database tier is responsible for providing data access and storage services to the business logic tier. Figure 4 depicts a logical view of components (both mandatory and some of the optional) and connectors that make up the generic FRS architecture. The FRS client tier contains components that handle the initialization, display and event-handling for the various Java panels used in reservation, user and facility management. The FRS server tier contains components that provide the actual event-handling code for the various user interface widgets (e.g., buttons). There are also components that set up and shut down connections with the DBMS component. Finally, the DBMS component may contain sub-components such as tables or databases to store data related to users, facilities and reservations. Table 3 describes connector types and Table 4 - connector instances reflected in Figure 4.

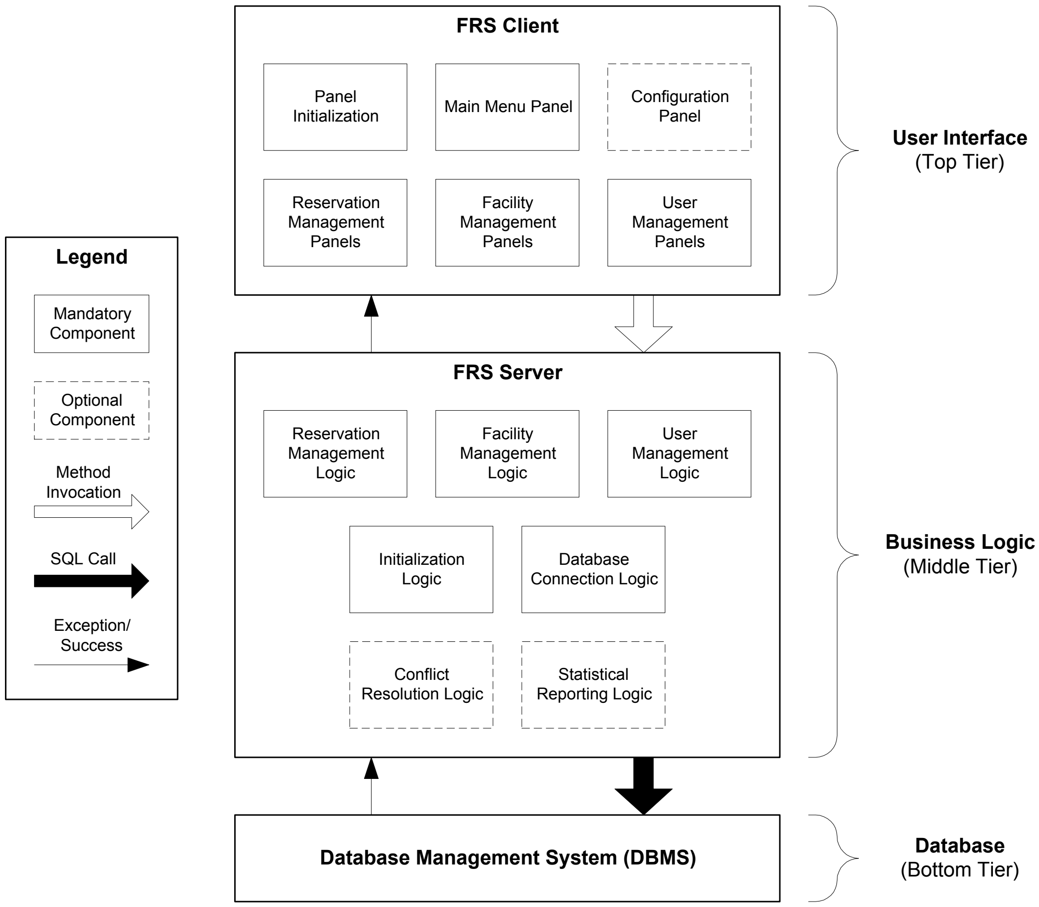

FRS Client
Panel Initialization
Main Menu Panel
Configuration Panel
Reservation Management Panels
Facility Management Panels
User Management Panels
User Interface (Top Tier)
Legend
Mandatory Component
Optional Component
Method Invocation
SQL Call
Exception/ Success
FRS Server
Reservation Management Logic
Facility Management Logic
User Management Logic
Initialization Logic
Database Connection Logic
Conflict Resolution Logic
Statistical Reporting Logic
Business Logic (Middle Tier)
Database Management System (DBMS)
Database (Bottom Tier)

Figure 4. Three-Tiered FRS runtime architecture

Connector Type	Description
⇨	A method invocation over an ORB (Object Request Broker). The ORB is middleware that establishes a client/server relationship between the client that issues the method invocation and the server that implements the method. The ORB intercepts the method invocation and locates the server that implements it.
➡	An SQL call, implemented using a JDBC (Java Database Connection)-to-ODBC (Open Database Connectivity) bridge.
→	Either an exception or success message.

Table 3. Description of connector types

Connector	Description
FRS Client to FRS Server	Method invocations that are triggered as a result of user actions (e.g., pressing a button to reserve a facility) on the client. Examples are adding a reservation and viewing the details of a facility.
FRS Server to FRS Client	May be reservation, user or facility data that is returned as a result of a method invocation. In some cases, error or exception data may be returned.
FRS Server to DBMS	SQL call to retrieve data from one or more databases managed by the DBMS. An example of such calls could be a request to find the largest existing reservation ID.
DBMS to FRS Server	Data from the DBMS' database(s) that is returned as a result of some SQL call.

Table 4. Description of connector instances

REFERENCES

1. Batory, D., Singhai, V., Sirkin, M. and Thomas, J. 'Scalable software libraries', *ACM SIGSOFT'93: Symp. on the Foundations of Software Engineering*, Los Angeles, California, pp. 191-199 (1993)

2. Biggerstaff, T. 'A perspective of generative reuse', *Annals of Software Engineering*, edt. Osman Balci, volume on Systematic Software Reuse, vol. 5, Balzer Science Publishers, pp. 169-226 (1998)
3. Wong, T.W., Jarzabek, S., Soe, M.S., Shen, R. and Zhang, H.Y. 'XML Implementation of Frame Processor', *Symposium on Software Reusability, SSR'01*, Toronto, Canada, May 2001, pp. 164-172 (2001)
4. Shaw, M. and Garlan, D. *Software architecture: perspectives on an emerging discipline*, Prentice-Hall, 1996
5. Bassett, P. *Framing software reuse - lessons from real world*, Yourdon Press, Prentice Hall, 1997
6. Soe, M.S., Zhang, H. and Jarzabek, S. "XVCL: A Tutorial," *Proc. of 14th Int. Conf. on Software Engineering and Knowledge Engineering,SEKE'02,* ACM Press, July 2002, Italy, pp. 341-349
7. Zhang, H., Jarzabek, S.and Myat Swe, S. "XVCL Approach to Separating Concerns in Product Line Assets," *Proc. of 3rd International Conference on Generative and Component-Based Software Engineering,* LNCS, Springer Verlag, September 2001, Erfurt, Germany, pp. 36-47
8. Parnas, D.L. 'On the design of program families', *IEEE Transactions on Software Engineering*, SE-2, pp. 1-9 (1976)
9. Batory, D., Lofaso, B. and Smaragdakis, Y. 'JST: Tools for Implementing Domain-Specific Languages', *Proc. 5th Int. Conf. on Software Reuse*, Victoria, BC, Canada, pp. 143-153 (1998)
10. Batory, D., Coglianese, L., Goodwin, M. and Shafer, S. 'Creating reference architectures: an example from avionics', *Symposium on Software Reusability, SSR'95*, Seattle, USA, May 1995, pp. 27-37 (1995)
11. Bass, L., Clements, P. and Kazman, R. *Software Architecture in Practice*, Addison-Wesley, 1998
12. Neighbours, J. 'The Draco Approach to Constructing Software from Reusable Components', *IEEE Trans. on Software Eng.*, SE-10(5), pp. 564-574 (September 1984)
13. Gomma, H. 'Reusable software requirements and architectures for families of systems', *Journal of Systems Software*, Vol.28, pp. 189-202 (1995)
14. Tracz, W. 'DSSA (Domain-Specific Software Architecture) pedagogical example', *ACM Software Engineering Notes*, pp. 49-62 (1995)
15. Kang, K., Cohen, S.G., Hess, J.A., Novak, W.E. and Peterson, A.S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990
16. Rumbaugh, J., Jacobson, I. and Booch, G. *The Unified Modeling Language, Reference Manual*, Addison-Wesley, 1999
17. Jacobson, I., Griss, M. and Jonsson, P. 1997. Software Reuse. Architecture, Process and Organization for Business Success. ACM Press
18. Johnson, R. 'Frameworks = (components + patterns)', *Communications of the ACM*, 40, 10, pp. 39-42 (1997)
19. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
20. Schmid, H. 'Systematic framework design by generalization', Communications of the ACM, 40, 10, pp. 48-59 (1997)
21. Digre, T. 'Business Component Architecture', *IEEE Software*, pp. 60-69 (September/October 1998)
22. Czarnecki, K. and Eisenecker, U., *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000
23. Kiczales, G, Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J. 'Aspect-Oriented Programming', *Proc. European Conference on Object-Oriented Programming (ECOOP'97*), Finland, Springer-Verlag LNCS 1241, pp. 220-242 (1997)
24. Tarr, P., Ossher, H., Harrison, W. and Sutton, S. 'N Degrees of Separation: Multi-Dimensional Separation of Concerns', *Int. Conference on Software Engineering, ICSE'99*, Los Angeles, pp. 107-119 (1999)
25. Goguen, J. 'Parameterized programming', *IEEE Transactions on Software Engineering*, SE-10, No. 5, pp. 528-543 (1994)
26. Sommerville, I. and Dean, G. 'PCL: A language for modeling evolving system architectures', *IEE Software Engineering Journal*, pp. 111-121 (1996)
27. Karhinen, A., Ran, A. and Tallgren, T. 'Configuring designs for reuse', *Proc. of the 1997 International Conference on Software Engineering*, *ICSE'97,* Boston, MA., pp. 701-710 (1997)
28. Lung, C. and Urban, J. 'An approach to the classification of domain models in support of analogical reuse', *Proc. of ACM Symposium on Software Reusability, SSDR'95*, Seattle, Washington., pp. 169-178 (1995)
29. DeBaud, J.M. and Schmid, K. 'A Systematic Approach to Derive the Scope of Software product lines', *Int. Conf. on Software Engineering, ICSE'99*, Los Angeles, pp. 34-43 (May 1999)
30. Jarzabek, S. and Zhang, H. "XML-based Method and Tool for Handling Variant Requirements in Domain Models", *Proc. of 5th IEEE International Symposium on Requirements Engineering, RE'01,* Toronto, Canada, pp. 166-173 (August 2001)
31. Cheong, Y.C. and Jarzabek, S. 'Frame-based Method for Customizing Product line architectures' *Proc. Symposium on Software Reusability, SSR'99*, Los Angeles, pp. 103-112 (1999)

32. Karhinen, A. and Kuusela, J. 'Structuring design decisions for evolution', *Proc. 2nd Int. Workshop on Development and Evolution of Software Architectures for Product Families*. Lecture Notes in Computer Science, Springer-Verlag (1995)

Appendix

Two feature diagrams in this appendix may give you hints about some other variants in the FRS domain. However, you must read those feature diagrams with care as they may not be consistent with the feature diagram of Figure 2.

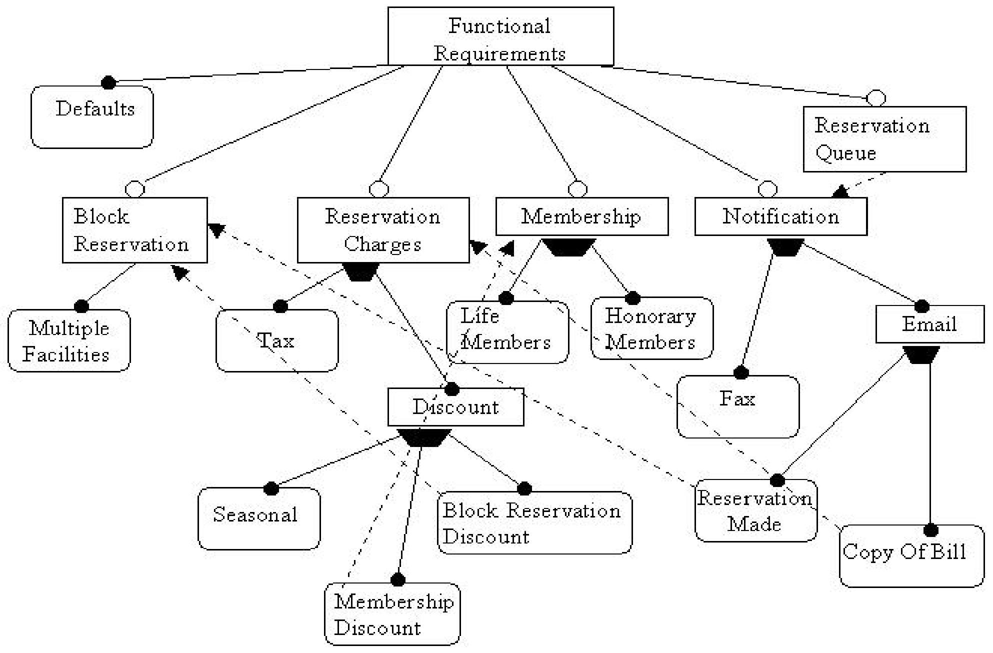


Figure 5. Extra features (1)

Explanation:

Reservation Charges:

Universities do not charge for reserving facilities such as tutorial rooms. However Hotels, Business centers etc. usually levy a charge on the rooms/function rooms they rent out. Variant *Reservation Charges* addresses the charging mechanism for renting of facilities. This includes methods to set the facility charges and to calculate the rent due for the facility. *Tax* variant and *Discount* variant groups depend on *Reservation charges*. *Discount* can be *Sesonal discount, Membership discount* and *Block reservation discount*. Dependencies are shown by dotted lines in Figure 5 (for example, *Membership discount* is only included if both discount and membership variants are included in the customized system). Also *Block reservation discount* should be included only when *Block Reservation* Variant and *Discount* variant are both included.

Membership:

Members are granted special privileges such as membership discount, Priority in reservations. Membership can be of two types *life members* or *honorary members* or both.

Notification:

Some FRSes notify its users about the details/status of their reservations. The notification medium may also vary, for instance, we could use fax, letter or email. A user may be notified using just one medium or multiple media.

Block Reservation

Block reservation allows a user to reserve multiple facilities at the same time. Users making *Block reservations* may be eligible for special discounts.

Reservation Queue

Reservation queue is a variant, which allows the system to create a reservation queue if the facility is not available. If a reservation is cancelled, the system will check the queue for queued reservation and allocate the facility to the reservation. Further enhancements can be email notification when the facility is available.

Dependencies among variants:

The simplified description of the facility reservation domain hints numerous variant requirements and dependencies among requirements. These dependencies are also called crosscutting among variants. Some examples of cross cutting are:

Membership Discount: Membership discount variant can be included only if both discount variant and charge variants are included in the system.

Copy of Bill: Copy of bill is only generated when both Notification variant and charges variant are included in the system.

Block reservation Discount: Block reservation discount is only give in a system when both charge variant and block reservation variants are included in the system.

These are just few examples of cross cutting variants. Many more can be identified in a commercial product line system.

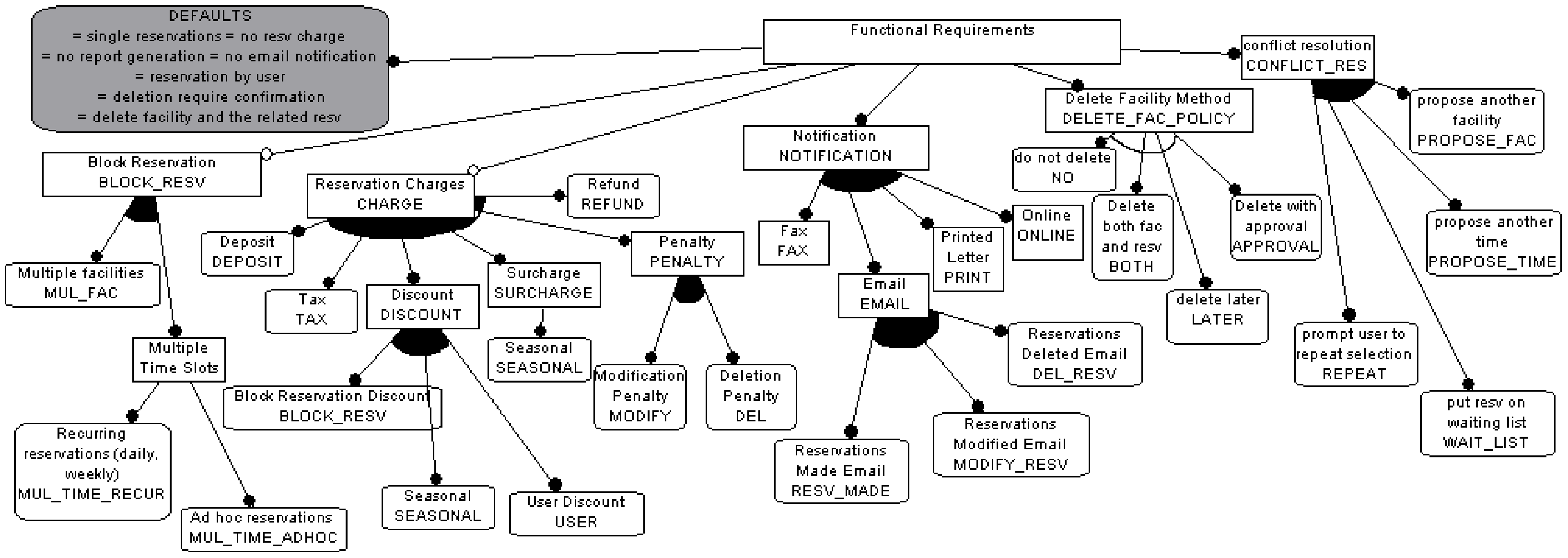

DEFAULTS
= single reservations = no resv charge
= no report generation = no email notification
= reservation by user
= deletion require confirmation
= delete facility and the related resv
Functional Requirements
conflict resolution CONFLICT_RES
Block Reservation BLOCK_RESV
Reservation Charges CHARGE
Refund REFUND
Notification NOTIFICATION
Delete Facility Method DELETE_FAC_POLICY
propose another facility PROPOSE_FAC
do not delete NO
Delete both fac and resv BOTH
Delete with approval APPROVAL
propose another time PROPOSE_TIME
Deposit DEPOSIT
Penalty PENALTY
Fax FAX
Printed Letter PRINT
Online ONLINE
Multiple facilities MUL_FAC
Tax TAX
Discount DISCOUNT
Surcharge SURCHARGE
Email EMAIL
delete later LATER
Multiple Time Slots
Seasonal SEASONAL
Reservations Deleted Email DEL_RESV
prompt user to repeat selection REPEAT
Block Reservation Discount BLOCK_RESV
Modification Penalty MODIFY
Deletion Penalty DEL
Recurring reservations (daily, weekly) MUL_TIME_RECUR
Reservations Modified Email MODIFY_RESV
put resv on waiting list WAIT_LIST
Reservations Made Email RESV_MADE
Ad hoc reservations MUL_TIME_ADHOC
Seasonal SEASONAL
User Discount USER

Figure 6. Extra features (2)