JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

MODULAR DESIGNING OF SOFTWARE PRODUCT LINE

Dr. Mahua Banerjee
Assistant Professor
Department of Computer Science
Birla Institute of Technology, Mesra, Lalpur Campus, Ranchi, India

Abstract: Model Driven Development (MDD) is a promising area of software development. MDD shifts software development from a code-centric activity to model-centric activity. To accomplish this shift, modeling concepts are required at different level of abstractions which are later transformed to code generic model. This paper has discussed a MDD by developing a Software Product Line (SPL) known as Array Product Line (APL) using flow, state-chart and activity diagrams as a first stage of MDD. The significance of the diagrams is also explained through a case study. Finally, the paper suggested for transiting from traditional Software Engineering to Quantum Software Engineering.

IndexTerms: **Software-product-line, Quantum-Software-Engineering, Model-driven-development, Array-product-line**

1. INTRODUCTION

Refactoring has become an essential tool for handling software evolution. It is known that waterfall model considers the maintenance of software as the last phase of its life cycle and do not take into account the evolution of software. On the other hand, models like Rapid Prototyping, Joint Application Development and Extreme Programming, have better support for software evolution and therefore for refactoring [1]. These methods encourage the use of models at the initial stage of development rather than at code generation languages. The models can be designed in such a way that they may represent different views of the system like the SRS (requirements), different methods for achieving the requirements, interfaces etc. On the other hand, models can be developed for designing the software architecture. This gives a superior view of the different parts of the system. It also gives an enhanced view of how they are related to form the whole system. At a later stage of the Software Development Life Cycle (SDLC), one may need to perform few behaviour-preserving transformations, which accomplish some important design refactoring. These transformations could be performed inside a *behaviour-preservation* mode, where designers could graphically perform design improvements without generating unexpected results on the various codes.

Activity diagrams, flow diagrams and state-chart formalism are popular for the specification, design, and implementation of interactive systems. State-charts generalize finite-state machines by adding hierarchy, concurrency and communication thus enabling concise specifications of the systems. Tools are required to automatically compile state-chart specifications into executable code so that all maintenance can be done on the state-chart rather than on the generated code. Activity diagrams are useful for designing interactive systems which depict interfaces with more clarity. Flow diagrams are necessary to depict the flow of processes in any system.

Refactoring is required not only in the developmental stage but also in the maintenance stage. Since, maintenance phase is the longest phase in SDLC so refactorings are often required in this stage. It is better to prepare the activity diagrams at the designing level, prior to code development so that refactoring required during maintenance can be carried out on activity diagrams rather than on the codes. This approach minimizes the risks involved in direct code level refactoring. In support of this idea, an attempt has been made to produce some flow diagrams and activity diagrams, in a proposed SPL known as Array Product Line (APL). The proposed APL provides all the basic operations of array which is displayed through a GUI environment. This APL allows the user to select few features through a GUI, generates a meta-expression and finally converts the source code on a preferred high-level language. As a designing phase of the present APL, activity diagrams are developed for the user interfaces and a few flow diagrams are designed for the APL. Further, compound refactorings on the proposed APL in modular level are applied.

.

2. OVERVIEW OF SOFTWARE PRODUCT LINES

Features are commonly used to specify non-software products. A very familiar example where features are used for non-software product, is the Dell website. The HTML pages provide simple Declarative Domain Specific Languages for specifying customized configuration of PCs. The customization can be single or dual processor options, monitor options, memory configuration options etc. [2]. Another similar example is BMW's web site that customizes an automobile [2]. All of these examples are designing products by selecting a few features. Since combinations of features are different from each other, different products are generated.

This is the main motivation towards a new approach of software development that is known as Software Product Line (SPL). This is just like a product line which generates different products from different combinations of features.

2.1 Characteristics of Software Product Line

The traditional software development is facing several challenges in context of maintainability, reusability and flexibility. The post development life cycle needs different techniques and approaches to counter these challenges. This is a problem that not only exists for large-scale and medium-scale software development but it also exists for small-scale software development. Hence, there is a need to analyze the special characteristics of software product line in the following domain.

a) *Reusability and adaptability* - High reusability of product line means that a large part of the SPL can be commonly reused. On the other hand, high adaptability indicates that the effects of feature variations can be suitable for product line assets.
b) *Evolutionary and Incremental development* - Product line assets may need to be evolved by addition of new features or by deletion of existing features. The other way is that they may be developed by incrementally incorporating product line features.
c) *Configurability* - Product line asset components are configured according to the features selected for a particular product. This is done by using Feature-Oriented Programming.

In order to achieve higher reusability and adaptability it is initially required to separate commonalities from variability. This is accomplished by separation of common features from the variable features. Secondly, it is required to separate feature dependencies and feature binding code from components implementing core functionality. The common features and the components implementing the core functionality can be the base modular components and the variable features can be the aspectual components. Therefore, incorporating variable features into product line assets does not have any effect on the existing assets. This also supports evolutionary and incremental development of product lines as one will be able to integrate new features without modification of existing features. Configurability is achieved by Feature Oriented Programming (FOP). In order to get the first two characteristics (reusability and incremental development) Aspect Oriented Programming (AOP) and FOP are required. Hence, AOP and FOP are inherent components for SPL.

2.2 Development of Software Product Line

Studies have shown that considerable improvements in productivity can be obtained by applying the SPL development approach. It has been suggested that product line development can be decomposed into the following three dimensions [3].

a) The first dimension regards the *primary reuse assets of the product line*: architecture, components and systems
b) The second dimension regards the *organizational views of product lines*: business, organization, process and technology
c) The final dimension of product line development regards the *life cycles of the product line assets*: development, deployment and evolution

In context of reuse, the architecture consists of components like features and aspects. These can either be refined or refactored as per the requirement. The second dimension specifies which technology, process etc. are required for this area. Many methodologies have been invented to create product-line architectures [4, 5,6]. Unfortunately, the state-of-the-art is immature. The third-dimension talks about the life-cycle of the software that takes care from development till implementation and evolution.

3. PROPOSED SOFTWARE PRODUCT LINE

In the proposed software product line, Array Product Line (APL) is a typical product line application. A product-line, is a family of related software applications. It consists of a collection of components. The product-line architecture identifies the components which enable the formation of particular software. This is one of the product-line applications. Different product-line applications are represented by different combination of components.

The APL is a miniature product line which provides all the basic operations (like insertion, deletion, searching, multiplication etc.) in a one dimensional and two dimensional array. Similar to a product line, a user of APL specifies the options, displayed in a GUI. The selected option generates a metaexpression and finally the metaexpression gives the required source code. A detail discussion of APL in context of architectural metaprogramming, source code generation etc. can be done as a future work. The present paper designs, the activity diagrams and flow diagrams for the proposed APL. These activity and flow diagrams are required as the first step of meta expression designing.

3.1 Activity Diagrams in Proposed APL

The activity diagrams in the proposed SPL i.e., APL has been presented. The code generation from APL typically defines the preferred language, array type, array dimension and the operations. Initially, activity diagrams are presented for the user interface of the APL and then the flow diagrams are explained to describe the functioning of APL.

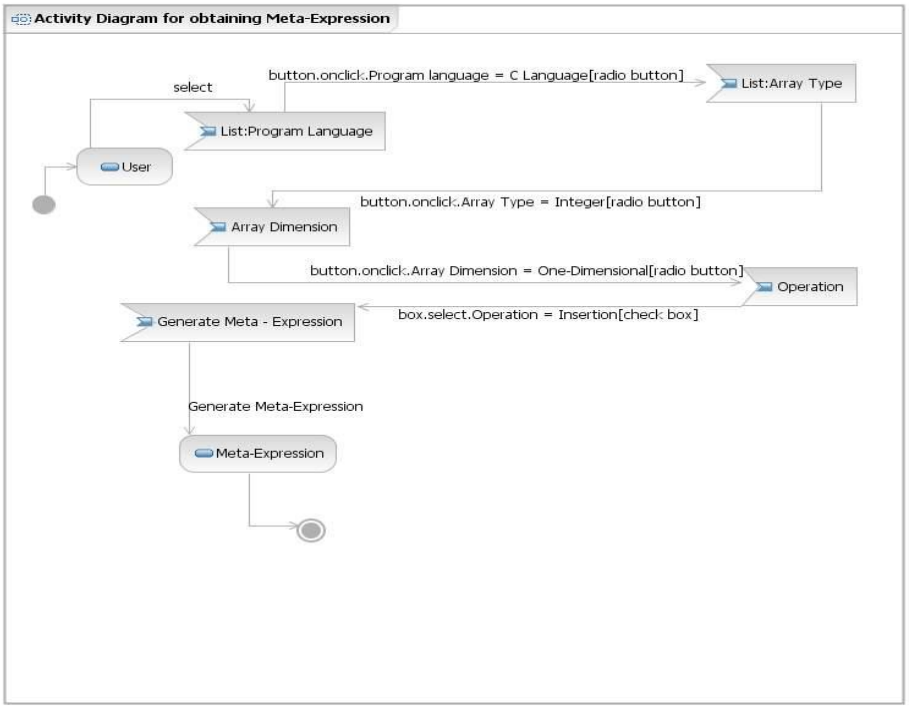

Figure 1: Activity Diagram of GUI environment of APL

The activity diagram in Figure 1 explains how a GUI environment is required to be designed for a specific metaexpression. The user will select the different options as explained below.

a) Programming Language is selected from a list by selecting the required radio button. In the above example, C Language is chosen.
b) Data type is selected as integer.
c) The dimension as one dimension array is chosen.
d) Operation as insertion is chosen through a check box.

The above activity diagram shows the requirement of radio buttons and check boxes in the GUI. Finally, on clicking the Generate Meta-Expression button, the required meta-expression is generated.

The checkbox and Button pressed as user's choice can further be modified by introducing Feature increment. A set of features can be the checkboxes and buttons which can be mapped to the appropriate functions. In a way it is incremental since one selects array type first, followed by the next increment which is dimension and finally one selects the third increment which is operation. All these activities can be chalked down first through activity diagrams and next through feature implementation.

3.2 Flow Diagrams on APL and Security Aspects

Refactoring can also be used to improve the design of activity diagrams. However, as activity diagrams do not simply model the system behaviour but also its operations, hence it is required to present the flow diagrams of APL in different stages. The design is first carried out from user's selection point of view. This is followed by the designing of the complete APL flow diagram. The following two figures (Figure 2 and Figure 3) show the flow diagrams.

(i) Partial i.e., after user's selection onwards, and

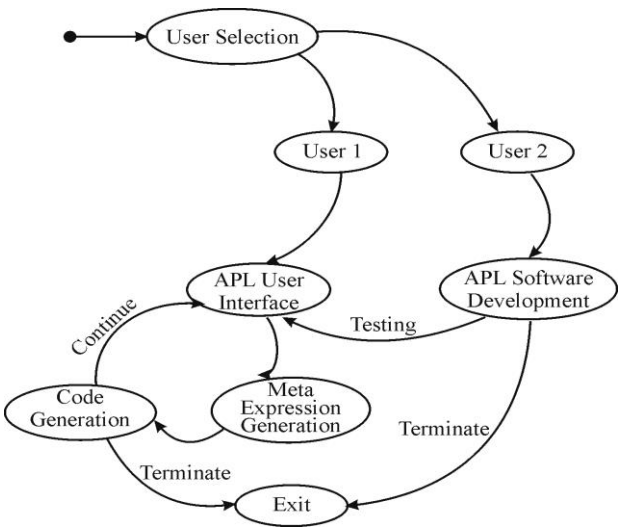

Figure 2: Partial APL flow diagram

Figure 2 shows that two different users have two different accesses. User 1 gets access to APL user interface through which one can develop the meta-expression. This further leads to Code generation which is followed by either continuation of further expression generation through interface or through exit from APL through termination. User 2 belongs to the development team and so is allowed to develop the software. User 2 has an access to the APL user interface for testing purpose. Once one gets into the interface, the process is similar to User 1. User 2 can also exit from APL through termination.

(ii) Complete flow diagram of APL

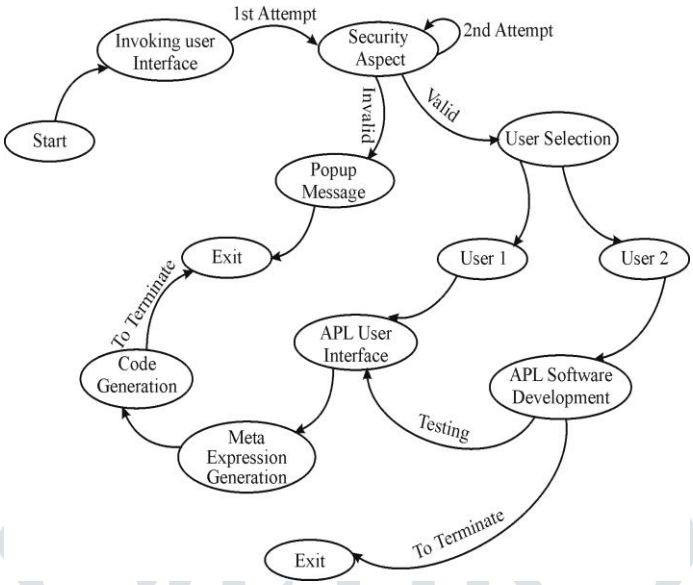

Figure 3: Complete APL flow diagram

Figure 3 shows the complete APL including the security login. It allows two attempts for login which either proceeds for the APL operations or proceeds for invalid entries and makes an exit with suitable messages. The rest is same as that of Figure 2.

The complete APL flow diagram shows that security and APL are two major models. The designing in modular level needs to treat them as two separate entities. Hence, the complexity of the problem increases if the models are designed and implemented together. On the other hand, the designing of the whole system with two major models (along with sub models like GUI designing, message communication, authorization etc.) will make increase in coupling and decrease in cohesion. This is not preferred in software engineering as software engineering experts assure that designs with low coupling and high cohesion lead to products that are more reliable and more maintainable [7, 8]. It is recommended to consider the concept of modularity. It is also recommended to treat individual modules separately. Here, the APL module is the major module as compared to the security. Therefore, concentration has been made on the APL model only for producing a more reliable and maintainable software.

In order to improve the design methodology, the following steps are carried out:

(a) Divide the complete flow diagram into two close independent modules which are the following.
  (i) Security Login Module
  (ii) APL Designing Module

(b) Segregate the security login module from the flow diagram and translate the module to codes through refactoring.

(c) Present the designing models of APL and translate the models to codes through refactoring.

As the steps carried out are same for all modules, option has been made for APL designing both in model level and in design level (which is a major requirement of such kind of SPL) in the present work. In order to achieve the model level designing, the flow diagram of Figure 2 has been refactored. In order to get the refactored flow diagram in Figure 4, the following four refactoring steps are required.

(i) Create a composite super state, named Active, surrounding the whole current diagram.

(ii) Move Idle and initial pseudo state out of Active.

(iii) Merge the “terminate” transitions into a transition leaving the boundary of Active.

(iv) Divide the “entry” transition into a transition from Idle to the boundary of Active and a default pseudo state/transition targeting User Selection.

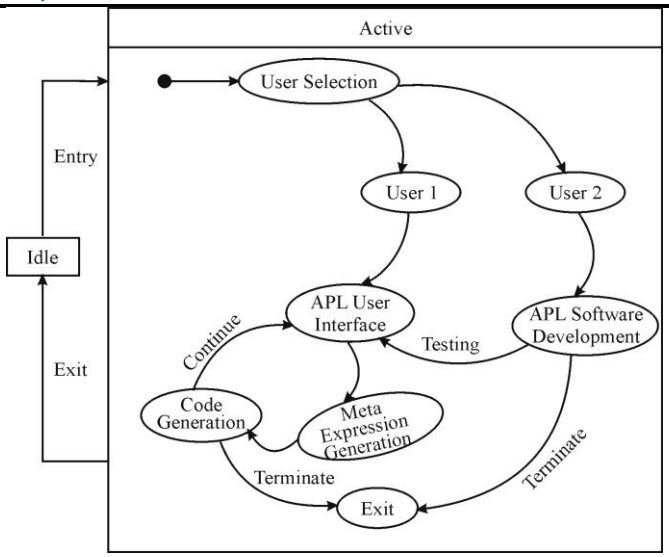

Figure 4: Refactored Flow Diagram

The following are the justifications for the previous transformations which are to be behaviour preserving (Figure 2 to Figure 4):

(i) Segregating the active state and idle state is behaviour preserving.
(ii) Moving the Idle state out is legal in this case as Active has no entry or exit actions, and so the execution order of existing actions is unchanged.
(iii) Transitions exiting Active can be placed to the idle state (a higher level) since there are no activities present during this time.
(iv) The replacement of the start transition is equivalent to entry into the Active state.
(v) The refactoring which have been used are addition, removal, move, generalization and specialization of modelling elements. The last two actions used the generalization relationship to transfer elements up and down a class hierarchy. In the above analysis, simple refactoring has been used. There is a scope of compound refactoring in the present APL which is discussed in the next section.

4. COMPOUND MODULAR REFACTORING

Refactoring is sometimes performed not as an individual operation but as a part of compound modification. For example, some refactoring become effective after another refactoring has already been performed. Furthermore, a prior-refactoring is often effective to correct thereafter the external behaviour of the program. Generally, refactoring and other non-refactoring modifications get entangled.

In the present APL there is an opportunity to perform a compound refactoring. A module of array can first be made which may abstract the member data like the limit of array, the element of array, the array etc. The member functions can be different array operations like insert, delete etc. For the sake of simplicity, a class module instead of the aspect module is considered. The class module is given below.

```
class arrays
{
   int lim, p, i, j;
   public:
        int *insert(int *abc);
       int * delete(int *abc, int  element, int l);
       void traverse(int *abc);   // One and Two dim. Array traversal
       int * search(int *abc, int element);  //  // One and Two dim. Array search
};
```

The class array has been refactored subsequently. This needs two refactoring operations. They are Extract method and Move method. Therefore, it is called compound refactoring. This is demonstrated in Figure 5.

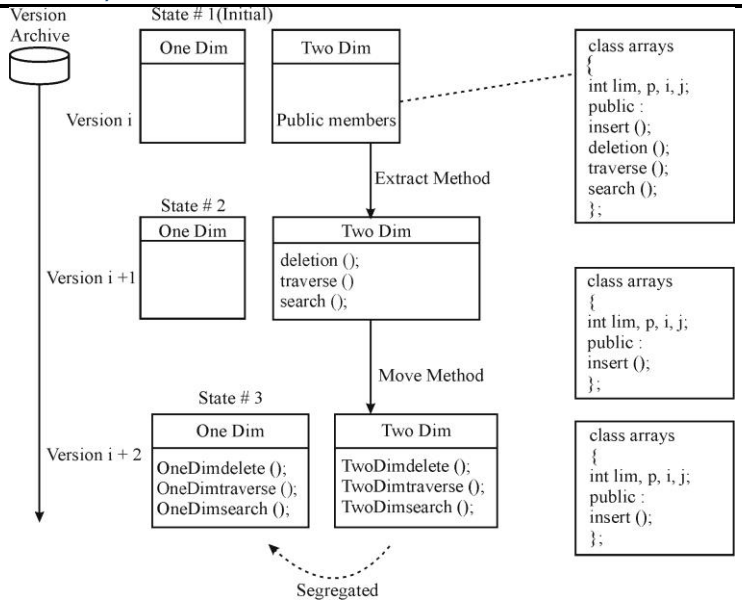

Figure 5: A compound refactoring of class members of APL

The compound refactoring consists of three stages. Stage #1 consists of the class arrays with its members and has two separate entities. They are One Dim and Two Dim. First Extract Method works for segregating the methods from the original code into the model Two Dim. This generates Stage #2. Next, the appropriate methods are moved to One Dim, which generates Stage #3. If the moving operations are performed directly from Stage #1 to Stage #3, then it will be an impure refactoring. This is because in Refactoring Flow Diagram (RFD) the two graphs of current state and target state (Stage #1 and Stage #3) are not isomorphic. Such refactoring is impure refactoring. Hence, Stage #2 is required to avoid impure refactoring.

One can have graphs for Version i to i+1 and Version i+1 to i+2 in order to illustrate that it can be incorporated in any RFD system. Version i+2 gives the methods name in two models One Dim and Two Dim. This represents the exact function to be carried out.

5. CASE STUDY: SMART TELEPHONE NETWORK PRODUCT LINE

The entire functioning of Smart Telephone Network Product Line (STNPL) can be designed in flow diagrams. This product line is different than APL. Hence, the processes are presented in flow diagram as the user interface is not graphical, rather than voice communication. This case needs different interfaces like the user to phone, the user to aspect, the aspect to PSTN and the PSTN to server. The interaction between the user and other objects must be designed carefully to depict the entire scenario. There is a need to identify the processes, interfaces and functions required in the flow diagram. The main objective is to show that depiction of the processes and their flow are same in a product line and software product line. The next section describes the flow diagram of STNPL.

5.1 FLOW DIAGRAM OF STNPL

The flow diagram given in Figure 6 explains the entire phone call process which includes the security check system. The interfaces, functions and the processes are placed in oval shape whereas the arrows show the flow of the processes. Certain situations are placed as text along the arrows.

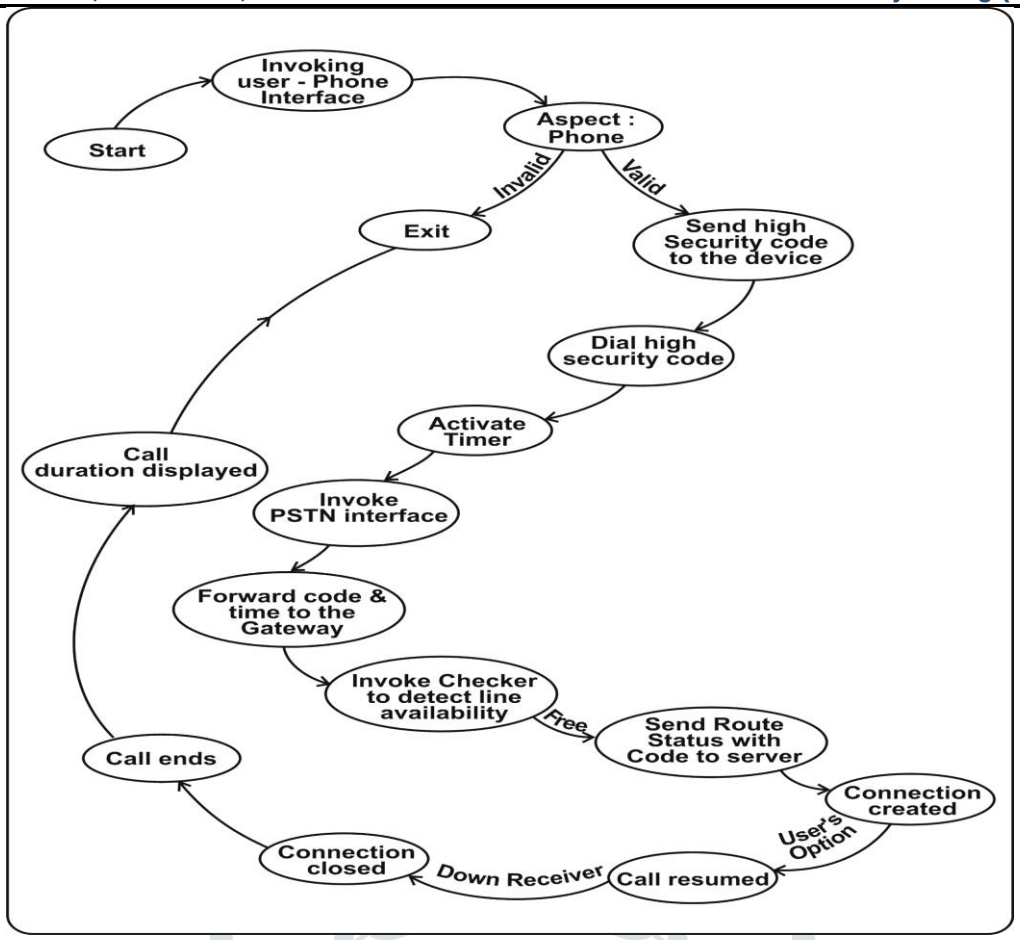

Figure: 6 Flow Diagram of STNPL

The user phone interface is first activated and then followed by the security process activation which is known as phone aspect. For an invalid entry, the system terminates. A valid entry needs a second round of validation by providing the security code that is obtained in the registered device. On dialling the security code, timer starts time recording and proceeds for the next interface. The PSTN interface gets the code and time for further processing. This is forwarded to the Gateway, which in turn invokes the checker for line availability. A busy line continuously invokes the checker unless it detects that the line is free. The next process is to establish a link with the server providing the code. The connection is established, and server to user communication begins. As per the user's option, the call resumes till the time the user hangs down the phone. The connection is removed and the call process ends with a message of call duration.

The advantages of this flow diagram are the following.

a) Identifying the processes
b) Identifying the interfaces
c) Sequencing the processes
d) Finding the relationships between interfaces and processes

It is recommended to use flow diagrams at the design stage to get a better view of the entire system. Activity diagrams do not give the exact processes, interfaces etc. except for the activities and their sequencing. Hence, it is better to have activity diagrams first and then the flow diagrams for detailing the activities. The STNPL case also has been demonstrated in the same way, giving the activity diagrams in the previous chapters and finally the flow diagram. The modular designing of STNPL gives an idea on the use of different types of diagrams in different situations and stages. It is felt that this modular designing of STNPL can help in implementing it to any organization/institution.

6. MODULAR DESIGNING USING QUANTUM SOFTWARE ENGINEERING

The Model Driven Development needs to incorporate the latest concepts and techniques. An emerging trend is Quantum Software Engineering. Designing the new techniques needed in the field of Quantum Software Engineering (QSE) requires taking into account all the new concepts along with all the lessons learned in classical software engineering development. During the last sixty years the classical software engineering generated advances, even overcoming several crises, that made possible the construction of complex software systems that deeply penetrated all aspects of today's society. The following section gives an overview of Quantum Software Engineering.

6.1 Model Driven Development in Quantum Software Engineering

The term "Quantum Software Engineering" was initially used by Stepney et al. in 2004 [9], but it has recently acquired popularity due to recent developments in quantum computing. Jianjun Zhao provided the following description of quantum software engineering, which is influenced by the traditional definition of software engineering [10]:

"Quantum Software Engineering (QSE) is the use of sound engineering principles for the development, operation, and maintenance of quantum software and the associated document to obtain economically reliable quantum software that works efficiently on quantum computers," according to the definition given by the IEEE. The authors of this definition noted three crucial problems in quantum software engineering: Initially, it's crucial to incorporate "sound engineering concepts" into the creation of quantum software. Second, "economical" construction of the quantum software is necessary. Finally, quantum software must operate "efficiently" on quantum computers and be "reliable" [11].

The traditional Software Product Line also needs to be aligned in this QSE principles. The changes need to be focused are incorporating the engineering concepts, adaptable in quantum computers and economically viable.

There are many expectations placed on Quantum Computing to contribute to almost any area of science. For example, exact calculations of molecular properties are currently intractable because their computational cost grows exponentially with the number of atoms. However, the power of quantum computer can make it a tractable problem [11]. Similarly, the SPL needs to be aligned in this new emerging field, though there are several challenges.

7. CONCLUSION

The designed activity and flow diagrams for APL justify that interface, processes, functions etc. can be visualized in a better way. As a modular designing of SPL or any product line, activity diagrams and flow diagrams play an important role. On the other hand, restructuring of the product lines can also become easy by applying refactoring in model level. It has been observed that activity and flow diagrams are very useful for software designing as they not only help in code generation but also help in software maintenance life cycle, where refactoring is required for up gradation. Refactoring the activity and flow diagrams does not affect the code. This reduces the risk involved in code level refactoring. This has been done by refactoring flow diagrams and also by performing compound refactorings which is required in modular level. Hence, flow diagrams and activity diagrams play an important role in modular designing. This has been revealed in the case study also. The present flow diagrams and the activity diagrams can be used in order to transit from the traditional Model Driven Development techniques towards the Quantum Software engineering techniques.

REFERENCES

[1] K. Beck, “Extreme Programming Explained: Embracing Change”, Addison-Wesley, 1999.
[2] D. Batory, “Program Refactoring, Program Synthesis, and Model-Driven Development”, In ETPAS Compiler Construction Conference, vol. 4420 of LNCS, pp. 156-171, Springer, 2007.
[3] J. Bosch, “Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study”, In Proceedings on the First Working IFIP Conference on Software Architecture, vol. 12, pp. 321-339, 1999.
[4] K. Czarnecki and U. Eisenecker, “Generative Programming: Methods, Tools, and Applications”, Addison-Wesley, 2000.
[5] P. America, H. Obbink, R.V Ommering, and F.V.D Linden, “CoPAM: A Component-Oriented Platform Architecting Method Family for Product Family Engineering”, vol. 576, pp.167-180, Springer, 2000.
[6] J. M. DeBaud and K. Schmid, “A Systematic Approach to Derive the Scope of Software Product Lines”, In Proceedings of the 21st International Conference on Software Engineering, pp. 34-43, 1999.
[7] N. E. Fenton, “Software Metrics - A Rigorous Approach”, Chapman & Hall, 1992.
[8] A. Macro, J.Buxton, “The Craft of Software Engineering”, Addison-Wesley, 1987.
[9] Stepney, S., S.L. Braunstein, J.A. Clark, A. Tyrrell, A. Adamatzky, R.E. Smith, T. Addis, C. Johnson, J. Timmis, P. Welch, R. Milner, and D. Partridge, Journeys in Non-Classical Computation. A Grand Challenge for Computing Research. 2004, University of York. p. 30.
[10] Weder, B., Barzen, J., Leymann, F., Salm, M., Vietz, D.: The Quantum Software Lifecycle. In: Proceedings of the 1st ACM SIGSOFT International Workshop on Architectures and Paradigms for Engineering Quantum Software (APEQS), pp. 2–9. ACM (2020).
[11] B. P. Lanyon, J. D. Whitfield, G. G. Gillett, M. E. Goggin, M. P. Almeida, I. Kassal, J. D. Biamonte, M. Mohseni, B. J. Powell, M. Barbieri, et al., Towards quantum chemistry on a quantum computer, Nature chemistry 2 (2010) 106–111.