On Collaboration in Software: Annotation vs. Flow-Based Diagrams

Sabah Al-Fedaghi*
Computer Engineering Department, Kuwait University, P.O. Box 5969 Safat 13060 Kuwait.

** Corresponding author. Tel.: 0096599939594; email: sabah.alfedaghi@ku.edu.kw
Manuscript submitted April 22, 2015; accepted August 27, 2015.
doi: 10.17706/jsw.10.11.1327-1335

Abstract: Programming may involve a fair amount of complexity. Accordingly, development of a description is needed to facilitate understanding and to serve more than documentation and initial planning requirements of a program. This paper focuses on the programming level at the point where stakeholders (e.g., designers, algorithm specialists) have concerns about the implementation method (coding) to be used, a concern that stems from the collective accountability of the software project team. The paper claims that current methodologies of producing high-level description to represent structural units and control streams in software programs fall short of providing a foundation for an easily understandable, high-level overview of what the code actually does. The paper contributes to the solution of such a problem by proposing an alternative approach to diagramming of programs. Without loss of generality, the focus on a very recent tool that generates flowcharts from annotated C++ source code to a set of interconnected high-level UML (Unified Modeling Language) activity diagrams. For comparison purposes, the diagrams produced by the methodologies are put side by side for the same source programs. The proposed diagrammatic representation seems to offer a viable alternative in the examined context to activity diagrams.

Key words: Software complexity, program annotation, C++, diagramming software, activity diagram.

1. Introduction

Programming may involve a fair amount of complexity. This complexity comes, roughly speaking, from the difference between two aspects: the high level (abstract) design on one hand, and the details of the specific code implementation on the other. Moreover, the high level design is not always easily understood from the implementation. The intricate mixture of the two aspects creates an understanding gap between different (groups of) developers with different expertise, hampering their collaboration on a given code. [1], [2].

The word “complex” describes the structural features (e.g., number of components and their connections) and interactions (e.g., calling, sharing) among units of a software system. A key element related to programming is *understanding* of the software system. In this context, understanding refers to *grasping of the mechanism* of the software solution to the problem under consideration. Of course, here, this understanding also refers to various levels of granularity, as in the case of concern in this paper, when the collaborators include programmers and algorithm or science specialists.

Granularity also refers to levels of software development and description. This paper is concerned with the *programming level* at the point where stakeholders (e.g., designers, algorithm specialists) have concerns

about the implementation method (coding), a concern that stems from the collective accountability of the software project team. If their scheme or design is badly implemented, they share responsibility for this failure; accordingly, they must understand the programs even if they are not programmers.

Here, a description is needed to facilitate understanding and to serve more than documentation and initial planning needs of a program. Additionally this description would be useful in:

- Improving communication among technical and non-technical participants.
- Enhancing understanding, maintenance, verification, testing, and parallelism. "Mechanisms for improving program comprehensibility can reduce maintenance cost and maximize return on investments in legacy code by promoting reuse" [3].

The case study examined in this paper suggests using annotation based on activity diagrams for this purpose. We propose an alternative methodology. The basic mechanism in both these methodologies is visualization of software.

Program visualization is a well-known paradigm and has proven to be an effective strategy for supporting programmers in clarifying the meaning of their codes and facilitating program understanding.

For many years, programmers have sought means of getting clearer meaning from their codes and from those of others with the aim of either developing it, maintaining it, or learning from it. Visualization of software or programs is one major way through which this need is met. [4]

Visualization is generally defined as the representation of information or data in diagrammatic form [5]. Visualization "has proven to be an effective strategy for supporting users in coping with complexity in knowledge- and information-rich scenarios" [6]. The "power of a visualization comes from the fact that it is possible to have a far more complex concept structure represented externally in a visual display than can be held in visual and verbal working memories" [7].

Program visualization is usually used to improve understanding of program behavior and in teaching students how to program [8]. Program understanding is one of the most important aspects influencing the maintainability of programs for programmers [9]. "Mechanisms for improving program comprehensibility can reduce maintenance cost and maximize return on investments in legacy code by promoting reuse" [3]. In program visualization, the program is specified in a conventional, textual manner, and graphics are used to illustrate certain aspects of the program or its run-time execution. [10]. Some researchers perform visualization at three different levels: statement level, file level, and system level [11]. Program visualization may focus on visualizing program-execution data, activity in the compiled code, the run-time system, and even the underlying hardware.

This paper claims that *current methodologies of producing high-level description to represent structural units and control streams in software programs fall short of providing a foundation for a human-understandable, high-level overview of the code's actual function*. The paper contributes toward solving such a problem by proposing an alternative approach to diagramming programs that was presented initially in [12]-[15] and has been enriched with advanced features in this paper.

Without loss of generality, here we focus on a very recent tool that generates flowcharts from annotated C++ source code to a set of interconnected high-level UML (Unified Modeling Language) activity diagrams, as given in Kosower *et al*. [1], [2]. This tightening of materials makes it easier to concentrate on a limited domain, i.e., specific C++ examples, thus achieving the capability to explore specific aspects of the proposed method. This approach is also motivated by the difficulty of providing a complete comparison between the general diagrammatic methodologies; accordingly, the strategy in this paper is to substantiate our claims by comparing the diagrams produced by each methodology side by side for the same source programs. At a minimum, the proposed diagrammatic representation appears to offer a viable alternative in certain contexts to activity diagrams. Future work can reveal further capabilities of the new methodology.

2. Focus Case Study: Activity Diagrams for Annotated C++

Kosower *et al*. [1], [2] recently introduced *Flowgen*, a tool that generates flowcharts from annotated C++ source code, a set of interconnected high-level UML activity diagrams [16].

Software documentation for developers does not however offer a human-understandable, high-level overview of what the code actually does. It also fails to keep this overview up to date with the code. Good coding standards and strategies such as code modularity or incremental development surely aid collaborative work, but cannot substitute for a higher-level view. Providing such a view in visual form is the challenge we seek to meet. [1], [2]

Flowgen provides behavioral diagrams that complement structural information. It aims at providing an easy-to-read, shared standard of communication (the activity diagrams) in a project [2]. According to Kosower *et al*. [1], [2], existing tools that generate activity diagrams from C++ source code, e.g., IBM Rational Rhapsody, Crystal FLOW, are closely tied to the code's low-level activity diagrams .

Example (from [1], [2]): As an example, consider a simple C++ source shown in Fig. 1 without annotations or directives. Flowgen reads the C++ program and produces the diagrams shown partially in Fig. 2. This example will be recast using the proposed diagrammatic methodology.

```
int main()
{
   int control flag=0;
   std::cin >> control flag;
   if {(control flag==1){
        VINCIA_ vinciaOBJ = new VINCIA();
        vinciaOBJ->shower();
   }
   return 0;
}

class VINCIA {
public:
     void shower();
};

void VINCIA::shower() {
std::cout << "the parton shower code would go here";
...
return;
};
```

Fig. 1. Sample C++ source program.

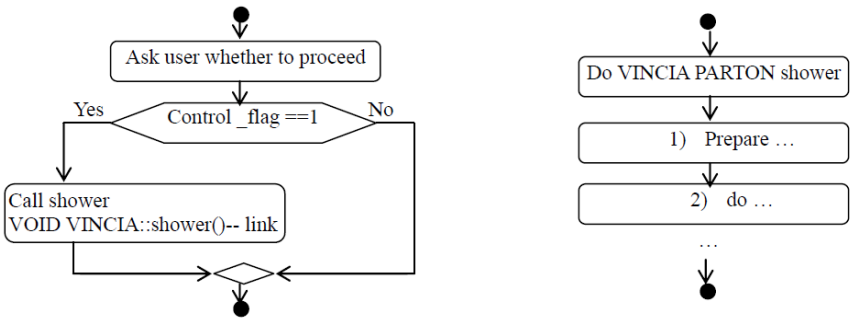

Fig. 2. Partial view of the diagrams produced by Flowgen (from [2]).

Flowgen aims at promoting the goals of: Facilitating code development in international collaborations, among developers with different levels of expertise and coding skills, possibly located in distant locations.

This involves speeding up the training of new developers; allowing the participation of "pure specialists" in the subject at hand, scientists with limited programming skills but an understanding of the high-level algorithm(s); simplifying the work of "pure" programmers, allowing them to concentrate on questions of code quality and performance. [2]

However, it seems Flowgen is mainly enhancing annotations in the source program and in the form of an activity diagram.

According to Kosower *et al*. [2], "Comments provide the building blocks for a successful resolution of this challenge [software complexity]." Commenting or, more generally, annotation is a term used in computer programming to refer to documenting code logic. Comments are typically ignored once the code is executed or compiled. Annotation is more general metadata, and it can be added to a field, class, or method, e.g., @Override in Java.

One reason for adding annotation to a programming platform is to assist development and runtime tools by creating a common infrastructure. Annotation gives the ability to provide additional metadata alongside program entities such as classes, fields, and methods. Annotation has other uses, including detection of errors by the compiler and generation of XML files by software tools, and some annotations can be obtained for examination at runtime.

A great deal of effort has been spent on making comments more on-point through developing a minimum set of basic types of annotation and predefined annotation concept.

However, it is difficult to see how heavy annotation would greatly facilitate collaboration among the described participants. The main objective in Flowgen is not mere documentation; rather, "The ultimate aim is to render complex C++ computer codes accessible, and to enhance collaboration between programmers and algorithm or science specialists." Annotations "are a very useful tool for documenting code.... However, annotations add complexity and don't offer enough of a benefit at all to warrant any use" [17]. Of course, this comment refers to the overuse of annotation. There are other difficulties in certain types of annotation, such as often the original resource cannot easily be recovered just by removing the annotation, difficulties arise in placement of whitespaces, etc. [18].

As background for this representation and for the purpose of a self-contained paper, the next section reviews the general features of the model to be utilized as "a ground fabric at the base of the design." The model has been used in many applications [12]-[15]; here it is newly applied to the notion of collaboration in software, and the example in this section is a new contribution.

3. Flowthing Model

To provide an underlying fabric that can serve to unify conceptual descriptions across different collaborations in software, we propose use of the Flowthing Model (FM). We claim that flows and "things that flow" (called flowthings) are fundamental notions in representations of processes. Here, flow refers to the exclusive transformation (i.e., one and only one transformation occurring at a time) of a flowthing passing among six states (also called stages) in a flowsystem: transfer (input/output), process, creation, release, arrival, and acceptance, as shown in Fig. 3. We will use *receive* as a combined stage of *arrive* and *accept* whenever arriving flowthings are always accepted.

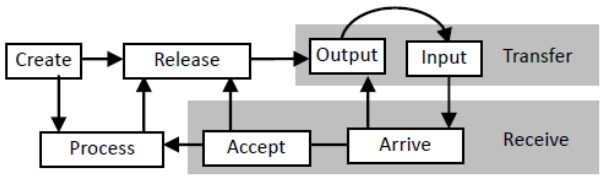

Fig. 3. Flowsystem.

The flowsystem shown in Fig. 3 is a generalization of the input-process-output (IPO) model that has been used in many interdisciplinary applications as well as to convey system fundamentals. A system is typically conceptualized as a set of interrelated components that collect (input), manipulate (process), and disseminate (output) data. The sequence of input-process-output is probably the most used pattern in the field.

The basic IPO conception is captured by "a process P acting on an input I and producing an output O" [19]. It views a system as a black box process with an interface, and the environment denotes everything outside that system. The interface can be invoked either by the system (output) or by the environment (input). The IPO notion of "process" hides structural (i.e., generic) divisions. The flowsystem opens the black box (processes) by decomposing it into several compartments and specifying connections among them.

A flowthing has the capability of being created, released, and transferred, arriving, being accepted, and being processed while flowing within and between units called spheres (subcontexts). A flowsystem comprises six stages and the transformations (edges) between them, as well as flows that can be controlled by the progress (sequence) of the stream of events (creation, release, transfer, transfer within a sphere, release, receipt, …) or by a triggering that initiates a new flow (denoted in FM diagrams by a dashed arrow). Spheres and subspheres are the environments of the flowthing, such as a company with subspheres of a computer and an employee. A sphere can include the sphere of a flowsystem that includes the transfer stage. Triggering is the transformation of one flow to another, e.g., a flow of electricity triggers a flow of air.

Example: Fig. 4 shows a UML activity diagram that illustrates the workflows of components in an automated (or automatic) teller machine (ATM) [20]. Fig. 5 shows the corresponding FM representation. Note that FM presents a "complete" description that maps the flows even outside the information system. It starts at circle 1 when the user enters his/her code that flows to the system (2) where it is processed (validated – 3). If the code is valid, this triggers (4) a request to the user to input the amount to be withdrawn (5). This request flows to the user (6), where it is processed (7) to trigger (8) the input (creation – 9) of the amount. The amount flows to the system (10), where it is processed to trigger:

- Release of the right amount of cash (11), which flows to the user
- Creation of a receipt (12) that flows to the printer (13), then to the user

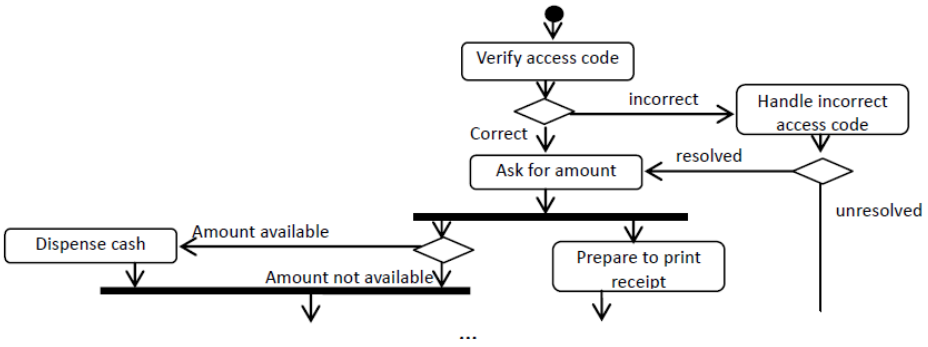

Fig. 4. UML activity diagram for an automated teller machine (partial, redrawn from [20]).

On the other hand, if the processing (3) of the code is negative, then the system creates (14) a request to handle the transaction, which flows (15) to the incorrect code handling system (16). This system, in turn, processes the request (17), and,

- If the code is OK, instructions are sent to the system to continue (19)
- If the code is not OK, then this triggers some type of response (20), e.g., an error message displayed on the screen

The last part about handling incorrect code has been added to the description because it is not clear from

the activity diagram what to do in this case.

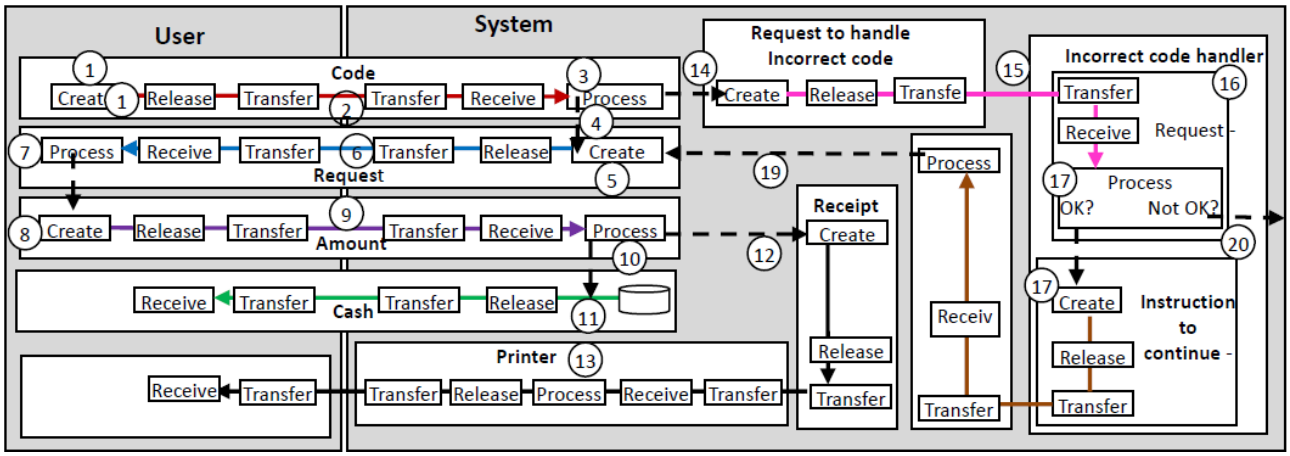

Fig. 5. FM representation of the automated teller machine

4. Applying FM to the Case Study

Fig. 6 shows an illustration of the structure of the FM representation of the program given in Fig. 1. Each statement is a sphere that has one or more flowsystems. Fig. 7 shows the actual FM representation.

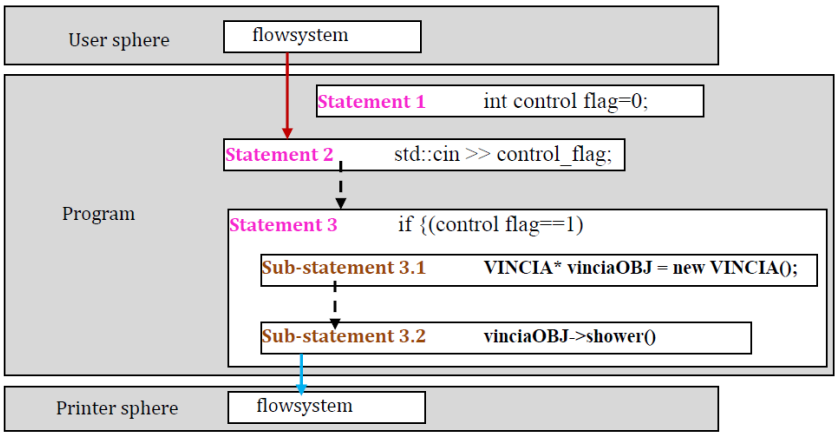

Fig. 6. Illustration of the structure of the FM representation of the case study.

In Fig. 7, the execution starts at *std::cin >> control_flag* (circle 1), when *flag* is initialized to zero. At the second statement (circle 2), user inputs 1 (3), which flows to the *if* statement (4), where it is processed (5) to trigger (6) the statement *VINCIA* vinciaOBJ = new VINCIA()*. This statement starts with the retrieval of a description of class VINCIA (7) to be sent (8) to *new* (9). *New* processes the class description (10), triggering (11) the creation of an object (12). The created object flows (13) to vinciaOBJ (14) in the assignment operator (15). vinciaOBJ is created as a pointer (16) and the incoming object (17).

The statement *vinciaOBJ->shower()* "pulls" in the pointer (18) to be processed (19), to trigger (20) retrieval of the object (21 and 22). The object is processed (23) to trigger (24) the activation (process) of *Shower* (25) that in turn triggers (26) the retrieval of "The parton shower code would go here" (27) and sending it to the screen (28).

This conceptual representation models the given program according to a specific technique used by the programmer. The method by which the user triggers the C++ program is obviously a technical decision, because there are many ways this can be accomplished (e.g., pressing any key). Similarly, using a pointer to

point to the object is a specific implementation selected by the programmer. It is probable that the team that includes the programmer developed an initial description of the program as shown in Fig. 8. It roughly includes the basic steps: (A) Receive start signal from the user, (B) Retrieve Class as defined, (C) Create an object, (D) Activate shower, and (E) Print out.

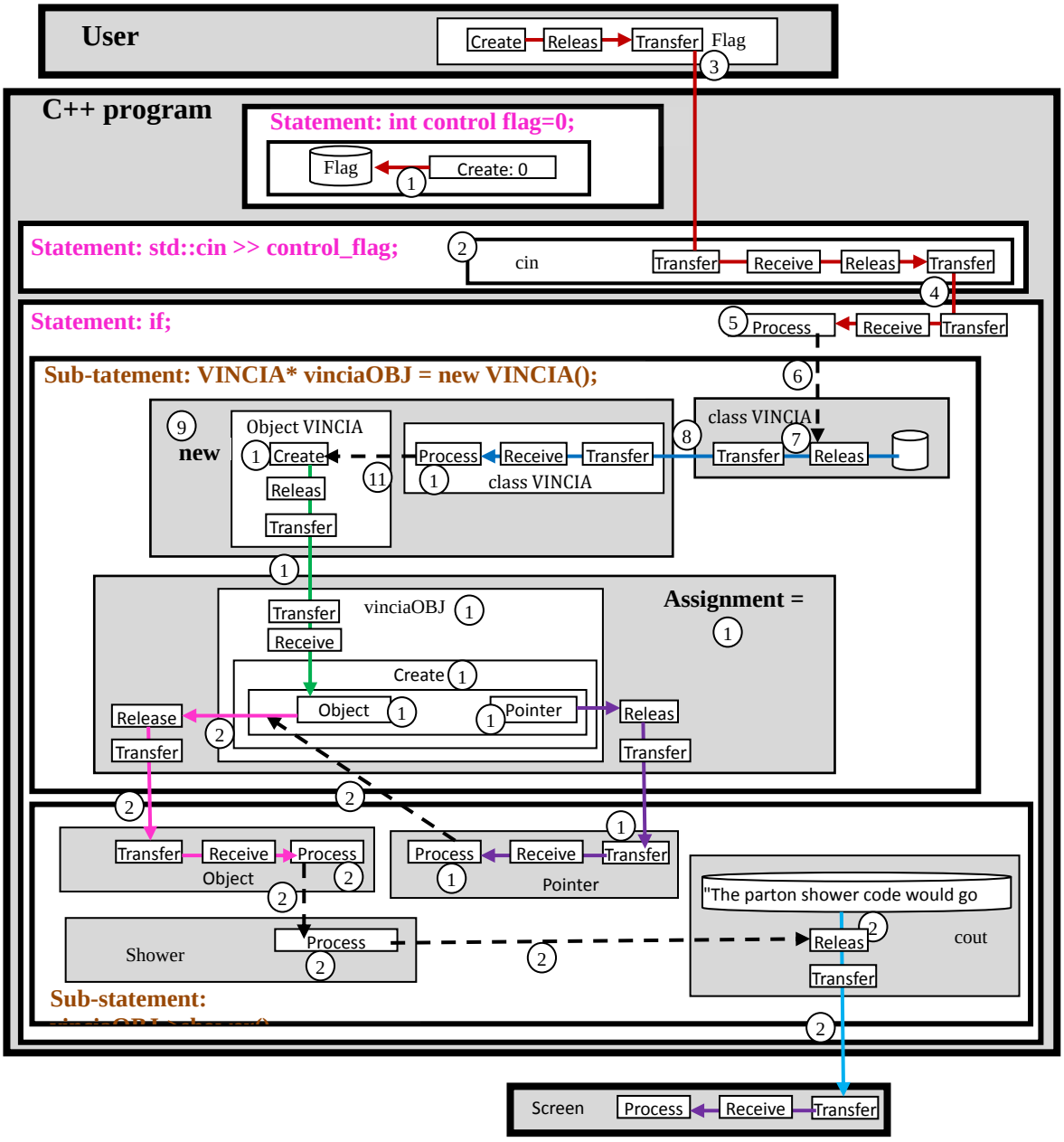

Fig. 7. FM representation of the case study program.

The programmer may have suggested using the flag method to activate the process, and using a pointer for the object for, say, efficiency consideration. Upon this agreement with the group the complete FM representation would be developed and discussed by the group.

It is not difficult to imagine the software team coming together around such an FM representation at different levels of description to agree on the next details and on motives for adopting a certain technique, analogous to an architectural team poring over drawings, hammering out design details. The whole FM depiction is conceptual, in the sense that it is not based on any particular software or hardware.

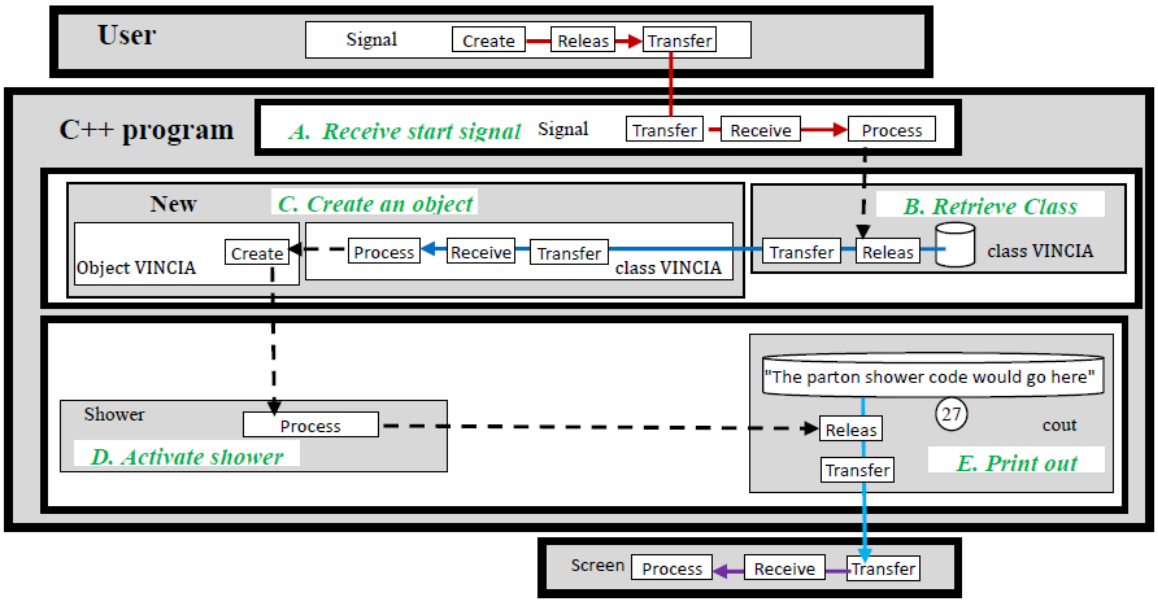

Fig. 8. An initial design of the program.

5. Conclusion

This paper proposes to use a flow-based representation as a base to facilitate understanding needed at the programming level where stakeholders have concerns about the implementation method. The paper demonstrates the viability of the proposed methodology by contrasting it with annotated activity diagrams. Future research can further develop the approach of FM representation programs in different languages.

References

[1] Kosower, D. A., Lopez-Villarejo, J. J., & Roubtsov, S. A. (2014). Flowgen: Flowchart-based documentation framework for C++. *Proceedings of the 2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation* (pp. 59-64).

[2] Kosower, D. A., & Lopez-Villarejo, J. J. (2014). Flowgen: Flowchart-based documentation for C++ codes.

[3] Kiper, J., Ames, C., & Howard, L. (1995). Using program visualization to enhance maintainability and promote reuse. *Proceedings of Computing in Aerospace, American Institute of Aeronautics and Astronautics*.

[4] Oluwatosin, A. H. (2007) Program visualization specification with UML (thesis). College of Natural Sciences, University of Agriculture, Abeokuta Ogun State, Nigeria.

[5] Gardenfors, P. (2000). *Conceptual Spaces: The Geometry of Thought*. Cambridge, MA: MIT Press.

[6] Keller, T., & Tergan, S. (2005). Visualizing knowledge and information: An introduction. In S. Tergan & T. Keller (Eds.), *Knowledge and Information Visualization*. Lecture Notes in Computer Science.

[7] Ware, C. (2004). *Information Visualization: Perception for Design* (2nd ed.). San Francisco: Morgan Kaufman.

[8] Myers, B. A. (1988). *The State of the Art in Visual Programming and Program Visualization*. Carnegie Mellon University, Computer Science Department.

[9] Basili, V. R. (1990). Viewing maintenance as reuse-oriented software development. *IEEE Software, 7(1)*.

[10] Bentrad, S., & Meslati, D. (2011). Visual programming and program visualization: Towards an ideal visual software engineering system. *ACEEE International Journal on Information Technology, 1(3)*.

[11] Orso, A., Jones, J., & Harrold, M. J. (2003). Visualization of program-execution data for deployed software. *Proceedings of the ACM Symposium on Software Visualization*, San Diego, California, USA.
[12] Al-Fedaghi, S. (2008). *Conceptualizing effects, and uses of information*. presented at Information Seeking in Context Conference (ISIC 2008), Vilnius. Lithuania.
[13] Al-Fedaghi, S. (2008). Scrutinizing the rule: Privacy realization in HIPAA. *International Journal of Healthcare Information Systems and Informatics (IJHISI) SCOPUS, 3(2)*, 32–47.
[14] Al-Fedaghi, S. (2013). Schematizing proofs based on flow of truth values in logic. presented at IEEE International Conference on Systems, Man, and Cybernetics , Manchester, UK.
[15] Al-Fedaghi, S. (2013). Flow-based enterprise process modeling. *International Journal of Database Theory and Application Compendex, 6(3)*, 59–70.
[16] Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The Unified Modeling Language User Guide* (2nd ed.). Addison-Wesley Professional.
[17] Butler, T. PHP: Annotations are an abomination [blog posting]. Retrieved January 23, 2013, from https://r.je/php-annotations-are-an-abomination.html
[18] Eckart, K. Aspects of annotations. Universität. Retrieved 2012, from Stuttgart.http://media.dwds.de/clarin/userguide/text/ann*otation_aspects.xht*ml
[19] Gile, D. (1994). Opening up in interpretation studies. In M. Snell-Hornby, F. Pöchhacker, & K. Kaindl (Eds.), *Translation studies: An interdiscipline* (pp. 149–158). Amsterdam: John Benjamins.
[20] CS Odessa Corp. *ConceptDraw: UML Activity Diagram - Cash withdrawal from ATM*. Retrieved 2015, from http://www.conceptdraw.com/examples/uml-activity-diagram

Sabah Al-Fedaghi holds an MS and a PhD in computer science from the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois, and a BS in engineering science from Arizona State University, Tempe. He has published two books and more than 100 articles in different peer-reviewed journals. He has, also, published more than 120 articles in conferences on software engineering, database systems, and artificial agents. He is an associate professor in the Computer Engineering Department, Kuwait University. He previously worked as a programmer at the Kuwait Oil Company and headed the Electrical and Computer Engineering Department during 1991–1994 and the Computer Engineering Department during 2000–2007.