Programming Is Diagramming Is Programming

Sabah Al-Fedaghi[1*], Esraa Haidar[2]

[1] Kuwait University, Computer Engineering Department, P.O. Box 5969, Safat 13060 Kuwait.
[2] Nazaha, Kuwait Anti-Corruption Authority, Kuwait.

* Corresponding author. Tel.: +965-99939594; email: sabah.alfedaghi@ku.edu.kw.
Manuscript submitted January 10, 2019; accepted March 8, 2019.
doi: 10.17706/jsw.14.9.410-422

Abstract: It is said that "programming is writing is programming." Both programming and writing involve high-level plans. Programming involves understanding the problem, creating a design, and coding. In this paper, we further explore the nature of programming based on the concept that "programming is diagramming." A diagram can be coded, and both the code and diagram approximate the conceptual (mental) form of the programmer behind both. We adopt a new diagramming technique called a thinging machine (TM) and build a TM diagram of the solution to the involved problem, which is sliced to produce program statements, much as flowcharts are converted to code. The TM introduces a simplified form with its five basic operations, which are repeated throughout the flow of events until reaching the end of the solution description. A case study is given that establishes an account through which a user can apply for a modeled job. The resulting diagram and program point to a viable approach to developing computer programs.

Key words: Abstract machine, conceptual model, computer program, diagramming.

1. Introduction

Programming and writing share the same mechanism: "the translation of a high-level idea into low-level sentences or statements," or, as we say, "programming is writing is programming" [1]. Both involve high-level plans. Programming involves understanding the problem, creating a design, and coding [2]. A programmer thinks of classes or objects to contain some parts of a program that contain methods and fields [1]. Programming is generally divided into three phases: problem solving, implementation, and maintenance [3]. According to Tillman [4], when creating a new program, the programmers are performing ontology. The first question is, "What is there?" The answer is that a system exists made up of objects which requires determining how to "carve reality at the joints" [4].

Alternatively, in this paper, we claim that "programming is diagramming." Drawings can be coded and vice versa, and both approximate the conceptual (mental) form of the programmer behind them. The semantics and syntactic problems in this mapping pose problems, such as equivalence, faithfulness, and communication of meaning, which are not discussed in this paper; rather, we show the foundation of the proposal that the diagram and code are representations of a single abstract machine called a thinging machine (TM).

Thinging can be thought as an answer to Tillman's [4] question, "What is there?" which requires "carving reality at the joints." Strictly, thinging is "defining a boundary around some portion of reality, separating it from everything else, and then labeling that portion of reality with a name" [5]. According to Heidegger **[6]**, a *thing* is self-sustained, self-supporting, or independent—something that stands on its own. TM is an

protracted manifestation of Heidegger's concept of the thing. The condition of being self-supporting transpires by *producing* (creating) the thing. Thinging, from our perspective—which deviates from Heideggerrian thought [6]—is a thing forming itself in the world as a thinging machine (TM). The TM is an abstract machine that creates, processes, and exchanges things. In a TM, programmers are not only doing *ontology* [4] (*Create* in a TM) but also *processing* and *exchanging* things. Accordingly, in this paper, we construct a grand TM of the solution to the involved problem, which is sliced to produce program statements, much as flowcharts are converted to code.

The next section discusses related notions in the field of diagrammatic languages. To achieve a self-contained paper, section 3 reviews the diagrammatic language TM that is adopted in this paper and was used previously in several published papers [7]-[14]. The examples discussed in section 3 are a new contribution. Section 4 presents a case study of building an application system in a real environment using a TM.

2. Related Works

The idea that programming is diagramming is not new. Computer programs go through the flowcharting phase, which describes the program at the abstraction level, before it is coded. A flowchart is a visual language for describing processes [15]. Flowcharts are considered representations of the conceptual structure of complex software systems. Conventional flowchart language tends toward restrictive control structures, which a flowchart cannot describe neatly. Moreover, certain control structures in programming languages cannot be translated into flowcharts and must be built from simpler control structures [16]. According to Ensmenger [17],

In much of the literature on software development, the flowchart serves as the central design document around which systems analysts, computer programmers, and end-users communicate and negotiate. And yet the meaning of any particular flowchart was often highly contested, and the apparent specificity of such design documents rarely reflected reality. In fact, some of the first software "packages" (commercial applications that could be purchased off-the-shelf) were used to reverse-engineer the flowchart specification from already developed computer code. That is to say, the *implementation of many software systems actually preceded its own design*! (Emphasis added)

We claim that a TM can become an effective tool that alleviates some of these flowcharting deficiencies when describing a program at the abstraction level.

Additionally, a TM as a diagrammatic language is related to visual programming, in which most visual languages use flowcharts to some extent, as in the case of flowchart-based visual programming techniques used by multimedia creators [15].

Other related works are centered on UML, which provides a standardized and comprehensive language for describing object-oriented systems. A notable aspect of the UML-based approach is the multiplicity of its models, a known problem [18] that contrasts with simply providing a single, integrated diagrammatic model that incorporates function, structure, and behavior [19]. These UML representations are completely heterogeneous in form, with several conceptual bases. These include the fact use cases are narratives, use case diagrams are sketchy, and sequence diagrams reflect the nature of exchange/communication, and so on. Such heterogeneity provides a wide range of options for expression, depending on the situation, but this approach has caused several problems. For example, a problem has arisen in achieving consistency between a UML use case model and its corresponding set of textual descriptions, which are the written explanations of the use case relationships contained in the UML model [20]. Hoffmann [20] notes that "This is a non-trivial problem, because ensuring consistency between a UML model and the textual use case descriptions requires a certain degree of formality in the textual descriptions".

We claim that TM modeling provides some advantages within the context of multiple diagrammatic notions.

3. Thinging Machine

If you are using *Word,* use either the Microsoft Equation Editor or the *MathType* add-on

Thinging involves creating, processing, and exchanging, and exchanging refers to releasing, transferring, and receiving. The TM creates, processes, and exchanges, as described diagrammatically in Fig. 1, which will serve as a foundation for our model.

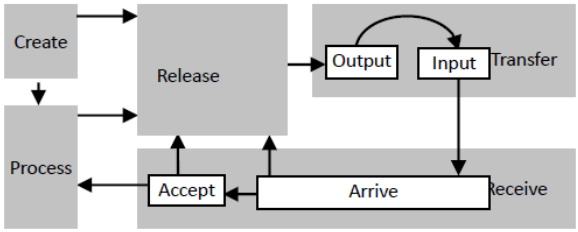


Fig. 1. Thinging machine.

Heidegger [6] introduced the term thinging to refer to the ontological wujud, i.e., the existence, presence, or being of a thing. This is represented in the Create step in Fig. 1 and limited to a certain system. For example, product, factory and customer have wujud in the context of a factory system. A product is a thing that is created, processed, released, transferred, and received from the factory machine to the customer machine (see Fig. 2). Note that the product is also a submachine from another perspective. Raw materials are transferred and received by the product. In object-oriented terminology, this implies that an object can be considered a class from one point of view, and vice versa.

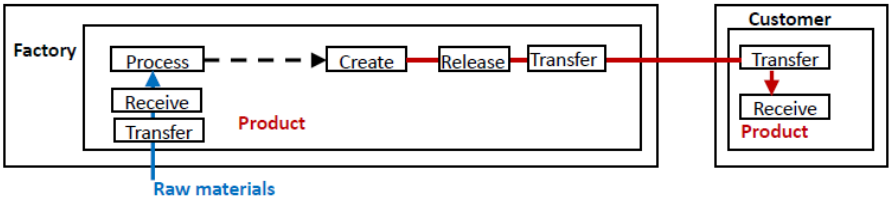


Fig 2. The product is both a thing and a machine.

According to Eckel [21] in his book Thinking with Java, "Think of an object as a fancy variable; it stores data, but you can 'make requests' to that object, asking it to perform operations on itself." However, a TM machine is more complex than a Java class, and a TM thing can have a more complicated structure than an object. Eckel [21] gives an example of an object-oriented representation of a light bulb, as shown in Fig. 3. We assume here a basic knowledge of object-oriented programming. Fig. 4 shows the corresponding TM representation of the bulb. The messages (signals—Circles 1 and 2) flow to the bulb machine to select the state (3) or contrast. Fig. 4 reflects the static description of the bulb machine.

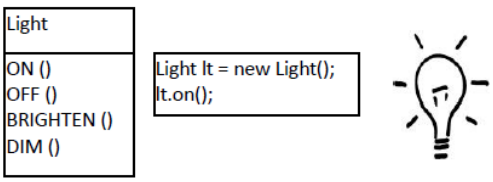


Fig. 3. An object-oriented representation of a light bulb (redrawn from [13]).

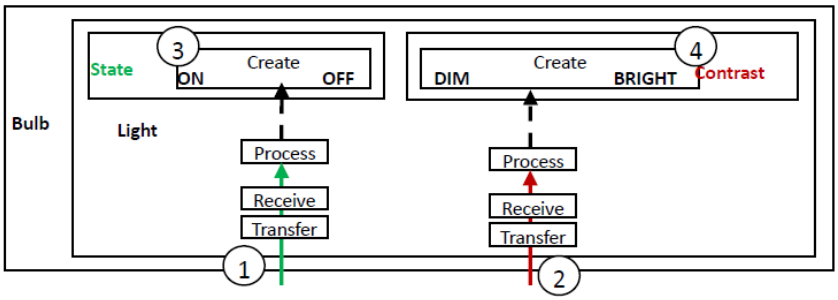


Fig. 4. Bulb machine.

Behavior in a TM is represented by *events*. An event is a thing that can be created, processed, released, transferred, and received. It is also a machine that consists of three submachines: region, time, and the event itself. Consider the event *The light is ON and BRIGHT* (see Fig. 5). It includes the event itself (Circle 1), the region of the things currently being dealt with in the event (2), and the time machine (3). The region is a subgraph of the static representation diagram of Fig. 4. For simplicity's sake, we will represent an event by its region only.

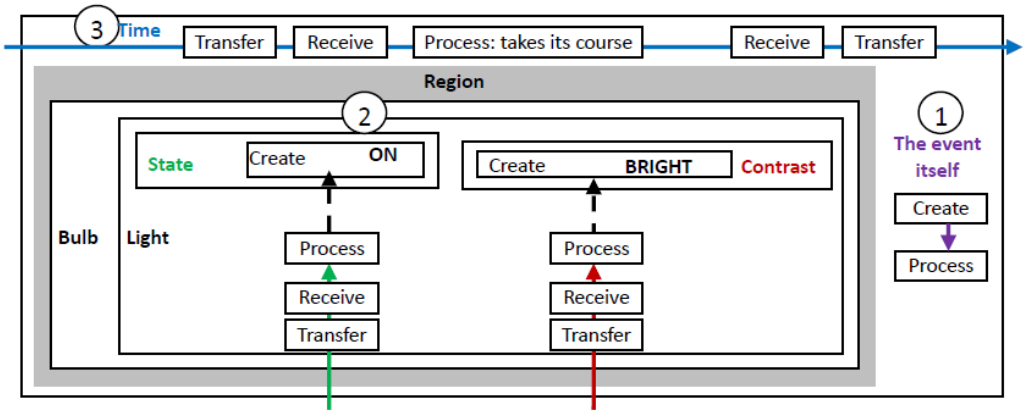


Fig. 5. The event *The bulb is ON and BRIGHT*.

Accordingly, we can identify four basic events in the static description of Fig. 4, assuming completely independent events, as shown in Fig. 6.

- Event 1 (E_1): The light is ON.
- Event 2 (E_2): The light is OFF.
- Event 3 (E_3): The light is DIM.
- Event 4 (E_4): The light is BRIGHT.

Going back to the object-oriented representation of a light bulb in Fig. 3, we can represent the methods of the class light with sub-diagrams. For example, Fig. 7 represents the instruction lt.ON(). The instruction lt.ON() is the event E_1.

Note that control in TM can be defined over the set of events. For example, Fig. 8 shows that if the light does not turn ON, a warning message is created.

4. Case Study: Job Application System

The following case involves a real organization (the second author's workplace, which is in the process of developing its information system, including a system for applying for jobs along with published vacancies). Accordingly, the organization requires a dedicated page in SharePoint to publish vacancies for and receive

CVs from applicants. The application page will have a Web form that includes all basic information of the applicants along with the ability to attach a CV. For our purposes, we will ignore such implementation-based portions of these requirements and focus on the job-application system as a programming project.

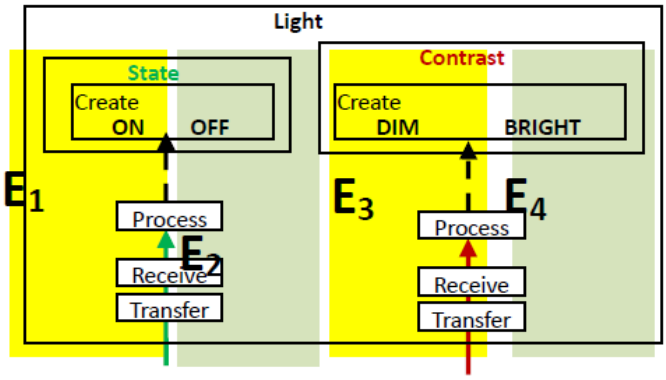


Fig 6. Light events.

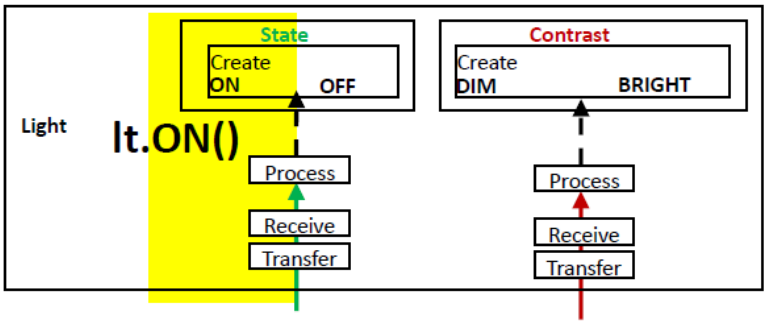


Fig. 7. lt.ON().

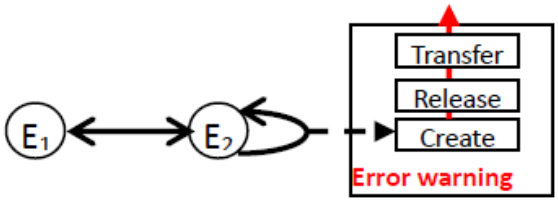


Fig. 8. If the light does not turn on, then a warning message is sent.

The system is described as follows:

(1) A citizen acquires an account.

(2) A login account is given.

(3) The citizen creates a personal file that includes information about him/her.

(4) He/she can then inspect currently available vacancies.

(5) He/she selects one of these vacancies, which is added to his/her personnel file.

Periodically, these files are collected and sent to a recruitment committee for them to select from the list of applicants.

In this section, we first develop a static description of the job-application system. Then, the system's dynamic behavior is captured by identifying its events. In the next section, these events are sliced separately

and converted to programming language statements.

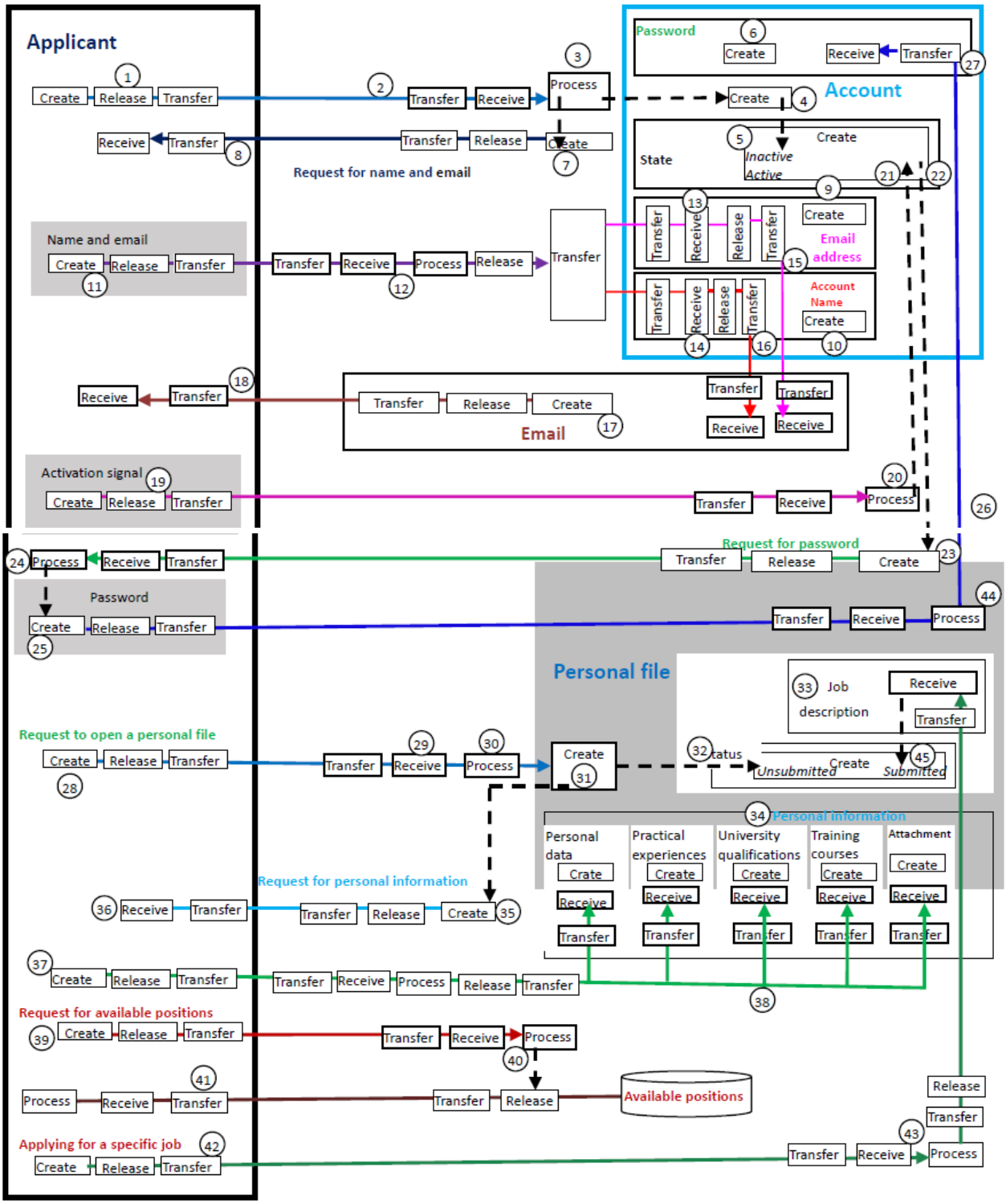


Fig. 9. Job application system.

4.1. TM Static System Description

Fig. 9 shows the TM representation of the system. First, the user creates (Circle 1 in the figure) a request for an account that flows (2) to the system, where it is processed (3). Creating (4) an account involves the following steps:

a) Initializing the state of the account as *inactive* (5).
b) Creating a password with temporary null value (6).
c) Creating the name (7—account) and the email (8), to be filled out later.
d) Requesting a name and email from the user (9 and 10).

When the applicant provides his/her name and email address (11 and 12), the system inserts them into the account record (13 and 14). The name of the account and the email are inserted (15 and 16) into an email (17) that is sent to the applicant (18), informing him/her of the opening of his/her account.

Upon receiving his/her account name, the applicant accesses the account to change its state to active (19, 20, and 21). Accordingly, the system requests the applicant's password (22, 23, and 24). When the system receives the password, it is inserted into the account (25, 26, and 27).

e) Next, the applicant requests the opening of a personal file that includes all information required to apply for a job (28 and 29). The system processes the request (30) to do the following:
f) Create a personal file (31) that includes its status (32—initially *un-submitted*), the job description (33), and personal information fields (34) to be filled later.

Send a message (35) to the applicant requesting personal information (36).

The applicant inputs data into the data fields (37), including his/her personal data, practical experience, university qualifications, and experiences, and also attaches documents (38). Then, the applicant requests current job vacancies (39), which are downloaded for him/her (40 and 41). Accordingly, the applicant selects a specific job (42), which flows to the system (43) to be inserted in his/her personal file (44). This changes the status of the personal file to *submitted* (45).

4.2. Dynamic description

Accordingly, the set of events can be developed as shown in Fig. 10:

- Event 1 (E1): Request to make an account.
- Event 2 (E2): Create an account form for the user.
- Event 3 (E3): Request that the user input an email address and name.
- Event 4 (E4): Inputting an email and name.
- Event 5 (E5): Send an email informing the user and requesting account activation.
- Event 6 (E6): Making the account active.
- Event 7 (E7): Request the password.
- Event 8 (E8): User inputs the password.
- Event 9 (E9): Request user create a personal file.
- Event 10 (E10): User creates a personal file.
- Event 10 (E11): Request the user input personal information.
- Event 11 (E12): User inputs personal information for the application.
- Event 12 (E13): User requests available positions and downloads them.
- Event 13 (E14): User applies for a specific job.
- Event 14 (E15): Insert the job description in the personal file.

5. From Diagramming to Programming

The basic idea of this paper is to convert TM events to programming language statements. We can select any programming language, including Web development languages. We selected C++ because of its popularity. For simplicity's sake, we ignored some features, such as login requests. Accordingly, we list each event and its sub-diagram, and then write a program that includes the corresponding C++ statements.

5.1. Slicing the Grand Machine

First, we slice the static representation into pieces that represent events as follows. For brevity's sake, we introduce here only the first eight sub-diagrams or regions of events. The rest of the events should be processed similarly as follows.

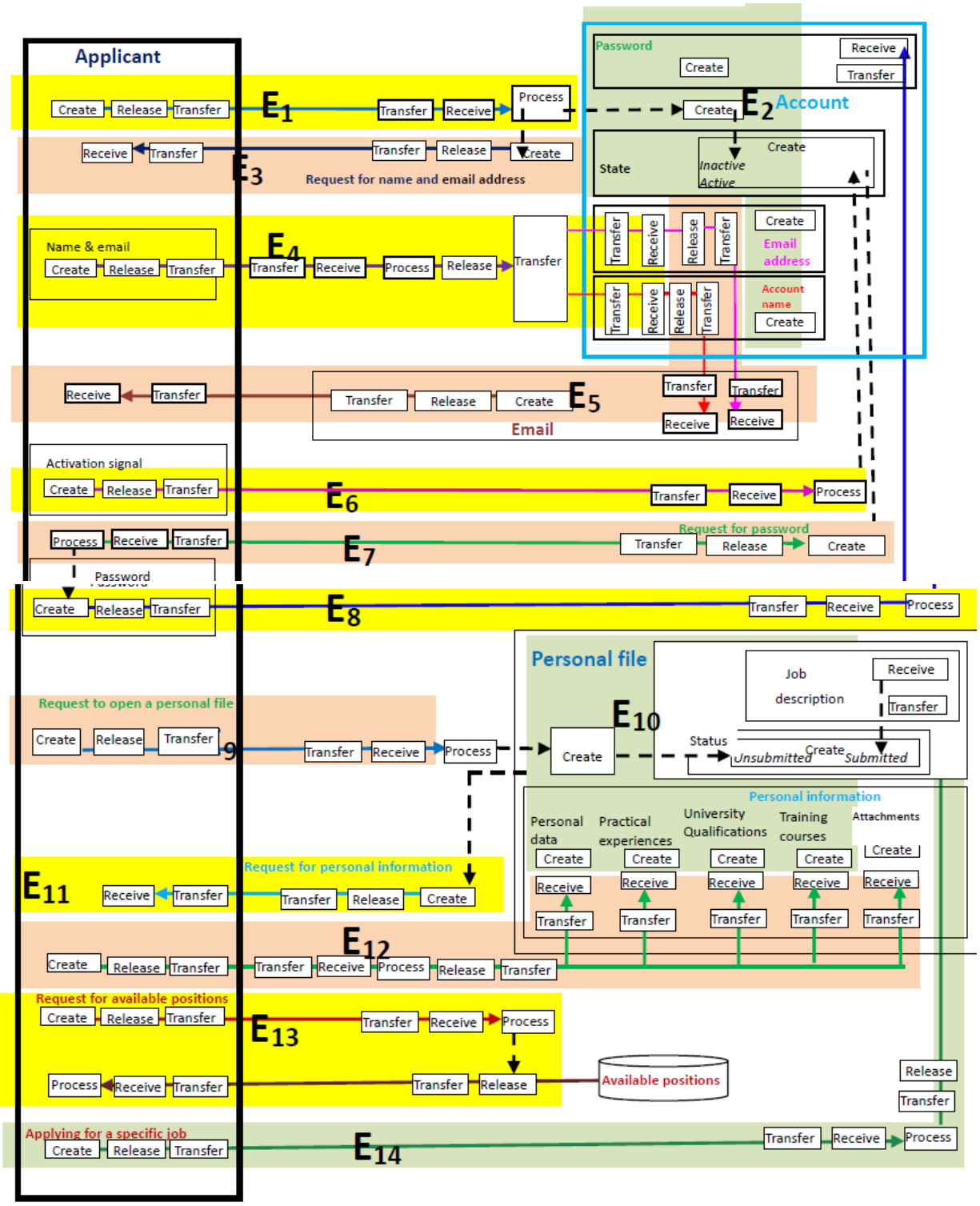


Fig. 10. Events of the job application system.

- E_1: Request to Make an Account (Fig. 11).

Fig. 11. Event 1 (E_1): Requesting to make an account.

- *E*2: Create an Account Form for the User (Fig. 12).

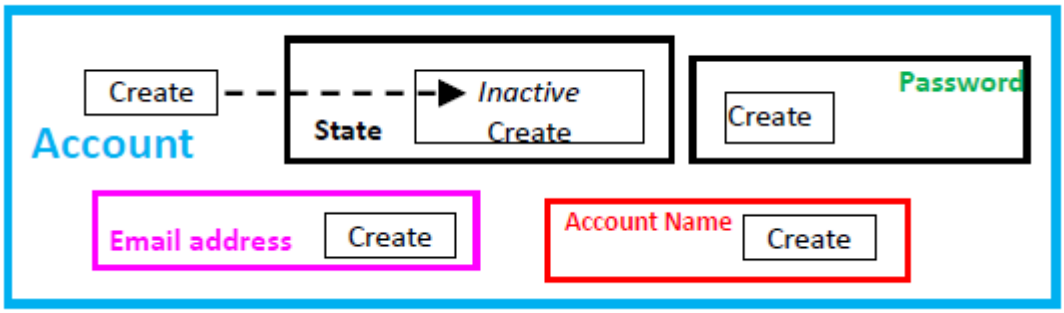


Fig. 12. Event 2 (E2): Create an account form for the user.

It is clear that Account is a class in C++, and it is required to create an object of this class, as follows:

- E3: Request the User Input an Email Address and a Name (Fig. 13).

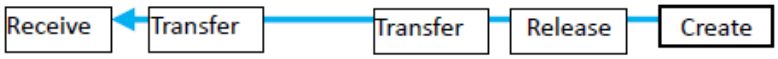


Fig. 13. Event 3 (E3): Request user to input an email address and name.

- E4: *Inputting* an Email Address and Name (Fig. 14).

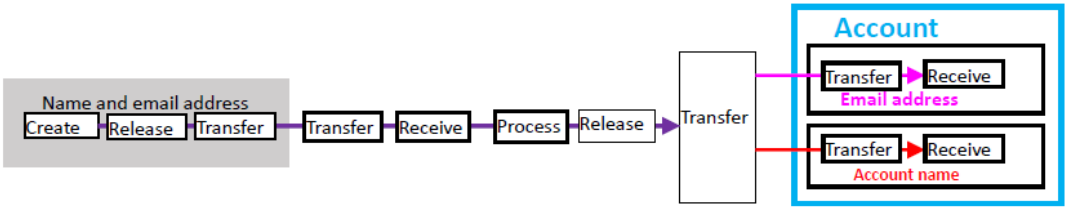


Fig. 14. Event 4 (E4): Inputting an email address and name.

- E5: Send an *Email* Informing the User and Requesting Account Activation (Fig. 15).

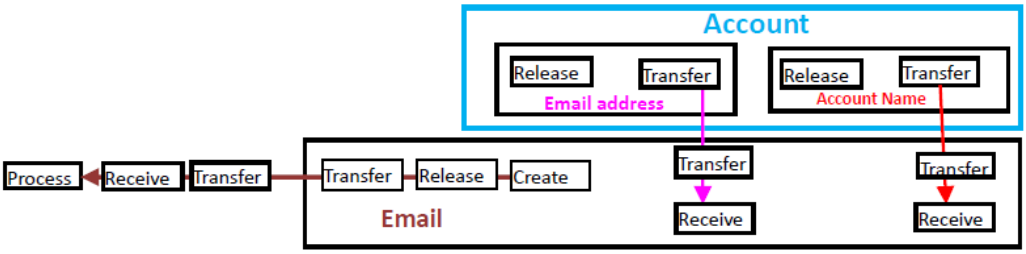


Fig. 15. Event 5 (E5): Send an email informing the user and requesting account activation.

- E6: Activating the Account (Fig. 16).

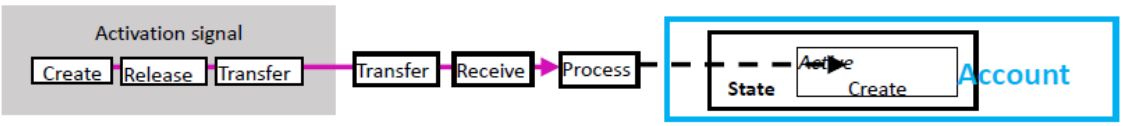


Fig. 16. Event 6 (E6): Activating the account.

- E7: Request Password (Fig. 17).

Fig. 17. Event 7 (E7): Request password.

- E8: *Inputting* the Password (Fig. 18).
-

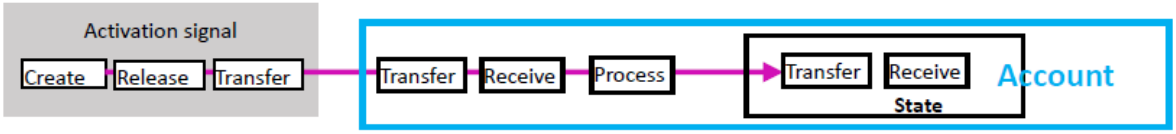


Fig. 18. Event 8 (E8): Inputting the password.

```
1         Class Recruitment      //Event1 ******
2         {
3          public
4          void Account()      //Event2 begin ******
5        void setemail(user[]); // set email
6        void account_name (user[]);        // set account_name
7        void setstate (user[]);         // set state
8        void setpassword (user[]);         // set password
9      void JobDescription (user[]);
10     void PersonalInformation (user[]);
9      private:
10       user email;
11       user account_name;
12       Bool state = False;
13       user password;  //Event2 ends ******
14        Bool status = False;
15        Void main()
16        {
17        char Name;           //Event3 begin *******
18        char EmailName;     //Event3 Ends *******
19        Cout << "Please Enter Email Address and Name
20        If (EmailName == new EmailName)  //Event4 begins********
21        Cout << "Email Name Already Exists"
22        Cout << "Enter Email Again"
23        cin>> EmailName;
24        else
25        char EmailName == new EmailName
26        char Name = new Name   //Event5 **********
27        //***************Event4 Ends ***************
28        set state = True      // Event6 *****
29
30        Cout << "Please Enter Password"   //Event7 ********
31
32        cin>> password;   //Event8 *********
.         …
n         }
```

Fig. 19. C++ program portion for the eight events.

5.2. Mapping Slices to C++

Accordingly, each sub-diagram is expressed in C++. Fig. 19 shows the resultant program portions for the eight events.

As mentioned in the bulb example in Section 2, control can be superimposed onto the events of the TM system. Suppose that in the job application case study, we want to impose the following constraints, motivated by the accumulation of inactive accounts:

If the new account is not initiated within a specific period, the account will be cancelled.

Fig. 20 shows the declaration of this rule over the chronology of events E_1 though E_6 because E_1 is the event in which the account is created. In the Fig. 20, when the account is created, the time and date are registered and processed, and the grace period (e.g., one month) is added. If E_6 occurs, this grace period will be cancelled; otherwise, the account will be deleted.

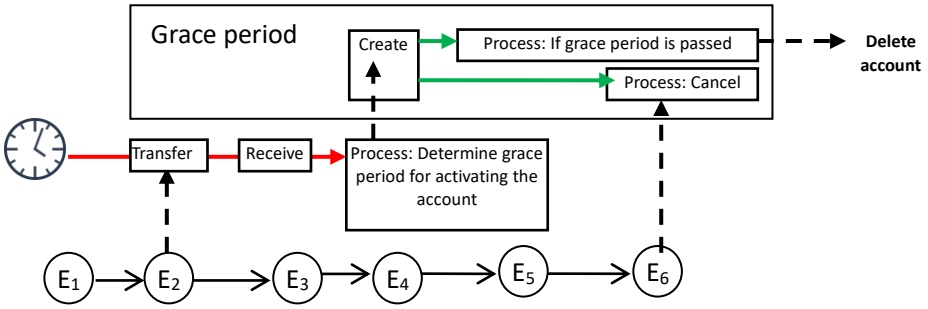


Fig. 20. Shows the declaration of this rule over the chronology of events E_1 though E_6 because E_1 is the event in which the account is created.

6. Conclusion

This paper adopts the TM model to describe a visual representation of the sequence of steps and decisions needed to perform a process at the abstraction level and before the program is coded, in a way similar to flowcharting. The TM introduces a simplified form using five basic operations that are repeated throughout the flow of events until reaching the end of the solution description. A case study is given that involves establishing an account through which a user can apply for a job, modeled as follows:

- The user requests an account
- The system initiates an account
- The system requests the user input the required data in the account
- …

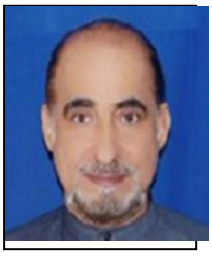
These steps are represented diagrammatically in terms of the five operations: *Create* (e.g., an account), *Release* and *Transfer* (e.g., a request), *Process* (e.g., the state of the account), and *Receive* (e.g., data). Accordingly, "chunks" of these elementary operations (e.g., the user requests an account from the system) are identified as events in time. These events are mapped to program (in this paper, C++) statements to produce the code. This reveals a logical piecemeal approach to translating the diagrammatic description into a programming language form.

The results point to a viable approach to develop a visual language for describing processes. Further research will explore the applicability of TMs in other types of programming problems (e.g., mathematical) and to certain applications (e.g., education).

References

[1] Hermans, F., & Aldewereld, M. (2017). Programming is writing is programming. *The First International*

Conference on the Art, Science and Engineering of Programming. Brussels, Belgium.
[2] Weinberg, G. (1985). *The Psychology of Computer Programming*. John Wiley and Sons, New York.
[3] Dale, N., & Weems, C. (1992). Introduction to pascal and structured design: Turbo version. 3rd edn. D.C. Heath and Company, Lexington.
[4] Tillman, M. (2008). Computer programming is practical philosophy. Retrieved from: http://micahtillman.com/ computer-programming-practical-philosophy/
[5] Carreira, J. (2018). Philosophy is not a luxury. Retrieved from: https://philosophyisnotaluxury.com/2011/03/02/ to-thing-a-new-verb/
[6] Heidegger, M. (1975). The thing. In: Hofstadter, A. (trans.) Poetry, language, thought. Harper & Row, New York.
[7] Al-Fedaghi, S. (2018). Thinging for software engineers. *International Journal of Computer Science and Information Security*, *16(7)*.
[8] Al-Fedaghi, S. (2018). Thinging vs objectifying in software engineering. *International Journal of Computer Science and Information Security*, *16(10)*.
[9] Al-Fedaghi, S., & Aljenfawi, H. (2018). A small company as a thinging machine. *Proceedings of the 10th Int. Conf. on Info. Mgmt. and Eng. (ICIME)*, University of Salford, Manchester, England, September 22–24.
[10] Al-Fedaghi, S., & Al-Huwais, N. (2018). Enterprise asset management as a flow machine. *International Journal of Modeling and Optimization*, *8(5)*, 290–300.
[11] Al-Fedaghi, S., & Alnasser, H. (2018). Network architecture as a thinging machine. *Proceedings of the Symposium on Mobile Computing, Wireless Networks, & Security (CSCI-ISMC)*. Las Vegas, NV, USA.
[12] Al-Fedaghi, S., & Alsharah, M. (2018). Security processes as machines: A case study. *Proceedings of the 8th Int. Conf. on Innov. Comp. Technol. (INTECH)*. London, England.
[13] Al-Fedaghi, S., & AlQallaf, A. (2018). Modeling and control of engineering plant processes. *International Journal of Applied Systemic Studies*, *8(3)*, 255–277.
[14] Al-Fedaghi, S., & Atiyah, Y. (2018). Tracking systems as thinging machine: A case study of a service company. *International Journal of Advanced Computer Science and Applications*, *9(10)*, 110–119.
[15] Revell, M. (2018). What Is Visual Programming? OutSystems Blog. https://www.outsystems.com/blog/what-is-visual-programming.html
[16] Nassi, I., & Shneiderman, B. (1973). Flowchart techniques for structured programming.
[17] Ensmenger, N. (20/12/2008 - Access) Thought Experiments in the History of Artificial Intelligence. Retrieved from: https:// www.ischool.utexas.edu/~nathan/categories/research/
[18] Dori, D. (2002). Why significant UML change is unlikely. *Communications of the ACM*, *45(11)*, 82–85.
[19] Dori, D. (2001) Object-process methodology applied to modeling credit card transactions. *Journal of Database Management*, *12(1)*, 4–14.
[20] Hoffmann, V., Lichter, H., Nyßen, A., & Walter, A. (2009). Towards the integration of UML- and textual use case modeling. *J. Object Tech*, 8(3).
[21] Eckel, B. (2006). *Thinking with Java*. 4th edn., Prentice Hall.



Sabah Al-Fedaghi is an associate professor in the Department of Computer Engineering at Kuwait University. He holds an MS and a PhD from the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois, and a BS in engineering sciences (computer) from Arizona State University. He has published more than 310 journal articles and papers in conferences on Software Engineering, Database Systems, information Ethics, Privacy and Security. He previously worked as a programmer

at the Kuwait Oil Company and headed the Electrical and Computer Engineering Department (1991–1994) and the Computer Engineering Department (2000–2007).

Esraa Haidar is a computer engineer graduated from Kuwait University, Collage of Engineering and Petroleum, Computer Engineering Department with bachelor degree (2004 - 2009). And currently, a master student in computer engineering department in Kuwait University, Collage of Engineering and Petroleum started on 2018. Previously working as programmer in ministry of public works (MPW) and now working as a computer engineer, system analyst in Kuwait Anti-Corruption Authority (NAZAHA).