ADVANCED ENGINEERING SYSTEMS AND SOFTWARE DEVELOPMENT SESSION
INVITED PAPER

AN APPROACH FOR SOFTWARE DESIGN AND DEVELOPMENT

Miloš Cvetanović*,
Zaharije Radivojević,
Stefan Tubić

University of Belgrade,
School of Electrical Engineering,
Belgrade, Serbia

Abstract:

One of the important challenges in software design and development is gathering of user requirements and its successful translation into engineering specification of a software product. This paper presents an approach for software design and development that enables gradually gathering of user requirements by using purposefully developed AFD language that enables a top-down functional decomposition. AFD is a text-based language with a simple 14 rules grammar and easy to understand semantics that are developed with computational thinking in mind. The computational thinking methodology is incorporated in multiple levels of decomposition in AFD. The lower levels are predominantly intended for users for expressing the requirements while the upper levels are intended for engineers for deciding upon implementation details. The proposed approach suggests using the first four levels for a software design and using the fifth level for mapping the design to selected software development paradigm. In case of object-oriented development paradigm, AFD provides automatic generation of appropriate UML sequence diagrams.

Keywords:

AFD, Functional Decomposition, UML, Software Design, Software Development.

INTRODUCTION

The first step in developing a software product is the requirements gathering process [1]. The functional requirements are those that relate to intended purpose of the product and its capabilities. The functional requirements serve as a main input for making a design specification, and moreover as a sound point of reference for checks and balances throughout production and quality assurance [2]. The main problem in requirements gathering is absence of a proper way of representing the functional requirements. Currently, the requirements are represented either in textual format that is easy for clients to understand but usually lacks sufficient information for engineers, or in graphical format that is preferred by engineers but usually has complex semantics that clients hardly understand. In both cases, the result is jeopardized expectations between clients and engineers, that leaves plenty of room for errors and ambiguities.

Correspondence:
Miloš Cvetanović

e-mail:
cmilos@etf.bg.ac.rs

DOI: 10.15308/Sinteza-2022-154-162

This paper sheds some light on a solution named Annotated Functional Decomposition (AFD) as an alternative for software design and development. AFD is a text-based language that is easy for clients to use and understand, while at the same time enables engineers to use it for a design and development as it provides extendable semantics with annotations for keeping track of all pieces of required information. Even though it could be used as a stand-alone solution, AFD could also be used to complement existing approaches such as UML (Unified Modelling Language).

The remainder of the paper is organized as follows. The second section presents AFD and explains its underlying methodology. The third section provides an example on how AFD could be used for a design of functionalities of an information system. The fourth section describes an approach for software design and development that connects AFD and UML and repeats some findings regarding the usage of AFD. The fifth section concludes the paper.

2. ANNOTATED FUNCTIONAL DECOMPOSITION

Annotated Functional Decomposition (AFD) is a new language that resembles natural languages (so it is easy to use by clients) and supports some semantics of computer languages (so it is adequate for use by engineers). AFD provides solutions for two main problems that exist during initial steps of software design and development. The first one is that functional requirements usually do not adequately recognize all functionalities needed for fulfilling intended purpose of the software product. The second one is that design specification does not meet all functional requirements. AFD solves those two problems by introducing extendable set of annotations for enabling stepwise refinement of functional requirements and linking to the resources that will implement those requirements.

In the essence, AFD is based on the existing design paradigm named Structured design that performs a top-down functional decomposition [3]. However, in order to provide support for recognizing all needed functionalities during the requirement gathering process AFD builds upon methodological concepts introduced by computational thinking (CT). CT is defined as a mental activity for the formulation of a problem and expressing the solution effectively, in such a way that a machine or a person can perform [4] [5]. To achieve the main goal, CT utilizes four techniques, also known as pillars. All four pillars have great relevance and are independent during the process of formulation of solutions computationally viable. The CT involves identify a complex problem and break it down in sub-problems that are easier to manage (decomposition pillar). Each one of these sub-problems can be analysed individually with greater depth, identifying similar problems which were previously solved (pattern recognition pillar), focusing only on the details that are important, whilst irrelevant information is ignored (abstraction pillar). At last, steps or simple rules can be created to solve each one of the sub-problems found (algorithms pillar). AFD implements all four pillars trough annotations divided into groups, that form different levels of the decomposition.

The first level of the decomposition in AFD is mandatory as it describes decomposition of a problem while all other levels are optional as they describe independent and orthogonal aspects of the problem. The second level describes control flow, the third describes data flow, the fourth describes reusage of decomposition parts, and the fifth describes resources regarding implementation. AFD language is formally defined by its context-free grammar that consists of 14 rules, as given in Table 1, which define different levels of decomposition [6]. The flexibility of the AFD is reflected in the fact that the rules are constructed so that the levels of decomposition are made orthogonal. Further extensions of AFD in terms of new levels of decomposition could be easily introduced by adding appropriate rules.

3. EXAMPLE OF AFD USAGE

Usage of AFD language could be demonstrated on an example of designing an information system. Designing process consists of stepwise refinement of functional requirements that starts with abstract description and goes gradually into details that at the end gives sufficient details for development. The steps actually represent different levels of decomposition as defined in AFD, where the first level of decomposition represents the most abstract one easily understanded by client and that is progressively updated by higher decomposition levels leading to the final design specification easily understanded by engineers.

The example resembles a ticket purchasing system. A user can purchase tickets for multiple seats on an event with an optional reservations previously done by the user. For each seat system finds a ticket, and then checks whether the ticket is still available and optionally reserved by the particular user.

No	Rule
1	Function ::= FunctionDef FunctionDecompEntry FunctionList FunctionDecompExit | FunctionDef;
2	FunctionDecompEntry ::= INDENT;
3	FunctionDecompExit ::= DEDENT;
4	FunctionList ::= FunctionList Function | Function;
5	FunctionDef ::= FunctionPrefix FunctionName DataFlows ResourceFlows Condition NEWLINE;
6	FunctionPrefix ::= ID SPACE | FTYPE SPACE | ID FTYPE SPACE | ;
7	FunctionName ::= NAME | NAME HASH | HASH NAME;
8	Condition ::= SPACE CONDITION | ;
9	DataFlows ::= LBRACE DataFlowList RBRACE | ;
10	ResourceFlows ::= LSBRACE ResourceFlowList RSBRACE | ;
11	DataFlowList ::= DataFlowList COMMA DataFlow | DataFlow;
12	DataFlow ::= DIRECTION NAME;
13	ResourceFlowList ::= ResourceFlowList COMMA ResourceFlow | ResourceFlow;
14	ResourceFlow ::= RESOURCETYPE COLON NAME;

Table 1 – AFD grammar

In case of successful checks the found ticket is purchased for the user and reservation, if existed, is removed. Information about all purchased tickets is shown at the end.

The first level of decomposition represents the core for all other decomposition levels. It identifies main function and breaks it down into sub-functions. The process is iteratively done until each function is resolved into sub-functions comprehendible by the user. Each function or sub-function is given in a separate line and is represented by its name that should be meaningful and descriptive in the context of the problem as much as possible. Sub-function is shown indented to the function that it resolves. For the given example, the first level of decomposition is shown in Listing 1.

The second level of decomposition introduces control flow into the result of the first level of decomposition. It identifies the order of the execution, conditional execution, and repetitive execution. The order is represented by numbering each function or sub-function at the particular level of indentation. Conditional execution is represented by a question mark after the number and condition using a forward slash sign and is given after the function name.

Repetitive execution is represented by an asterisk sing after the number and repetition condition using a forward slash sign and is given after the function name. Listing 2 shows both first and second level of decomposition for the given example, while the latter is highlighted in red.

The third level of decomposition introduces data flow into the result of the first level of decomposition. It identifies input and output data of the functions and is given in a pair of brackets after the function name. Multiple data are separated by comma signs, while each of them has name and input/output type. Input and output types are represented with greater then and less then signs respectively. On the higher levels of abstraction in a decomposition, a function could have data represented as streams of data that can be resolved into data for sub-functions following that function. In other words, data streams enable data decomposition in the same manner as functional decomposition enables decomposition of functions. Data streams are represented with an equal sign before the input/output sign while a dot sign is used to represent a-part-of relationship between streams and sub-streams. Listing 3 shows the first three levels of decomposition for the given example, while the third level is highlighted in red. All three levels are shown for the purpose of the completeness of the example, while orthogonality between the second and the third level enables their independent visualisation.

```
PurchaseSeats
	Input
	PurchaseSeat
		FindTicket
		CheckTicketAvailability
			HasTheTicketBeenPurchased
			WhetherTheTicketWasReserved
			WhetherTheTicketWasReservedByTheUser
			ReturnAvailability
		PurchaseTicket
			WhetherTheTicketWasReserved
			RemoveReservation
			Purchase
	GetPurchasedSeatsForUser
	Output
```

Listing 1 - The first level of decomposition in AFD for the ticket purchasing system

The fourth level of decomposition represents marking of the same functions. By identifying the same functions their re-usage becomes possible and therefore designing process eventually becomes more efficient and less error prone. The same functions are marked with a hash sign. Putting a hash sing after a function name represents that the function could be re-used, while putting a hash sign before a function name represents that the function is re-usage of some previously defined function.

Listing 4 shows the first four levels of decomposition for the given example, while the fourth level is highlighted in red. Due to orthogonality between levels of decomposition the fourth level could also be independently visualised.

The fifth level of decomposition introduces implementational details that are unessential for the user, however needed for an engineer. In case of object oriented implementation the details are information about classes that implement particular function. A name of a class that implements a function is given in a pair of square brackets that follows the function name.

```
1 PurchaseSeats
	1 Input
	2* PurchaseSeat /seat in seats
		1 FindTicket
		2 CheckTicketAvailability
			1 HasTheTicketBeenPurchased
			2? WhetherTheTicketWasReserved /purchased == false
			3? WhetherTheTicketWasReservedByTheUser/purchased== alse AND reserved == true
			4 ReturnAvailability
		3? PurchaseTicket /available == true
			1 WhetherTheTicketWasReserved
			2? RemoveReservation /reserved == true
			3 Purchase
	3 GetPurchasedSeatsForUser
	4 Output
```

Listing 2 - The second level of decomposition in AFD for the ticket purchasing system

```
1 PurchaseSeats(=>I.PS,<=O.PS)
      1 Input(=>I.PS,<user,<event,<seats)
      2* PurchaseSeat(>user,>event,>seat) /seat in seats
            1 FindTicket(>event,>seat,<ticket)
            2 CheckTicketAvailability(>user,>ticket,<available)
                  1 HasTheTicketBeenPurchased(>ticket,<purchased)
                  2? WhetherTheTicketWasReserved(>ticket,<reserved)/purchased==false
                  3? WhetherTheTicketWasReservedByTheUser(>user,>ticket,<reservedByUser) ⏎
                        /purchased == false AND reserved == true
                  4 ReturnAvailability(>purchased,>reserved,>reservedByUser,<available)
            3? PurchaseTicket(>user,>ticket) /available == true
                  1 WhetherTheTicketWasReserved(>ticket,<reserved)
                  2? RemoveReservation(>ticket) /reserved == true
                  3 Purchase(>user,>ticket)
      3 GetPurchasedSeatsForUser(>user,<purchasedSeats)
      4 Output(>purchasedSeats,<=O.PS)
```

Listing 3 - The third level of decomposition in AFD for the ticket purchasing system

Character C and a colon are used as a prefix for class name in order to denote that an object oriented implementation is used. Listing 5 shows the complete example with all five levels included, while the fifth level is highlighted in red, and just as for all previous levels of decomposition the orthogonality is maintained.

The design of a system given in AFD language could be verified according to the AFD grammar. For the purpose of verification an appropriate AFD Tool is implemented in Java as the plugin for Eclipse IDE, as one of the most widely used integrated development environment. Moreover, besides verification, AFD Tool enables mapping of AFD to UML. Details regarding relationship between AFD and UML are given in the following section.

4. RELATIONSHIP BETWEEN AFD AND UML

UML represents de-facto standard in domain of software design and development with more than 25 years of proved usability [7]. UML is a graphical language with extendable semantics that is primarily used for supporting object-oriented design and analysis [8]. In comparison with UML, AFD offers more technology agnostic approach that besides supporting object-oriented programming (due to the fifth level of decomposition) also supports traditional procedural based programming. Moreover, that also means that AFD could be more attractive for emerging technologies and novel programming paradigms (e.g. data-flow, functional programming, reactive programming).

AFD could also be seen as a technology complementing other existing ones. For example, using AFD as a text-based language to expedite creation of some UML diagrams, such as sequential or activity diagrams. Current implementation of the AFD Tool enables automatic generation of UML sequential diagrams according to a design of a system given in AFD language when the design includes all five levels of decomposition. Figure 1 shows a corresponding a UML sequence diagram for the ticket purchasing system given in Listing 5, while Figure 2 shows the sequence fragment that is a result of identified re-usage of a function on the fourth level of the decomposition.

Similarly, AFD could be used for use case scenario definition, or could be integrated with existing requirements management tools in order to provide better traceability of the requirements and their implementation in the final product. In that manner, AFD may complement UML during the requirements gathering process or even be considered as general overview of the specification whose details are elaborated on separate UML diagrams. An approach for using AFD in conjunction with UML for the purpose of software design and development is proposed in Figure 3. The approach suggests using the first four levels of decomposition in AFD for software design and using the fifth level of decomposition in AFD for mapping the design to selected software development paradigm. In case of object-oriented programming paradigm initial versions of some UML diagrams could be automatically generated (currently,

```
1 PurchaseSeats(=>I.PS,<=O.PS)
      1 Input(=>I.PS,<user,<event,<seats)
      2* PurchaseSeat(>user,>event,>seat) /seat in seats
            1 FindTicket(>event,>seat,<ticket)
            2 CheckTicketAvailability(>user,>ticket,<available)
                  1 HasTheTicketBeenPurchased(>ticket,<purchased)
                  2? WhetherTheTicketWasReserved#(>ticket,<reserved) /purchased == false
                  3? WhetherTheTicketWasReservedByTheUser(>user,>ticket,<reservedByUser) ⏎
                        /purchased == false AND reserved == true
                  4 ReturnAvailability(>purchased,>reserved,>reservedByUser,<available)
            3? PurchaseTicket(>user,>ticket) /available == true
                  1 #WhetherTheTicketWasReserved(>ticket,<reserved)
                  2? RemoveReservation(>ticket) /reserved == true
                  3 Purchase(>user,>ticket)
      3 GetPurchasedSeatsForUser(>user,<purchasedSeats)
      4 Output(>purchasedSeats,<=O.PS)
```

Listing 4 - The fourth level of decomposition in AFD for the ticket purchasing system

only sequence diagrams are supported). The rest of the software development activities would depend on the selected software development paradigm. Even though the proposed approach resembles the waterfall development methodology the approach could also support iterative or cyclic development methodologies [9] [10].

In order to support the assumptions regarding the benefits of AFD, a preliminary quantitative evaluation was done. The aim of the evaluation was to assess whether using AFD facilitates focusing on required logical checks and constraints while designing software products in comparison to UML. Groups of students who designed the products using AFD were considered during evaluation as experimental groups, while groups who designed the products using UML as control groups. The results showed that experimental groups achieved higher average score than control groups. On three experiments each one involving more than 100 students, average

```
1 PurchaseSeats(=>I.PS,<=O.PS)[C:BoxOffice]
      1 Input(=>I.PS,<user,<event,<seats)
      2* PurchaseSeat(>user,>event,>seat) /seat in seats
            1 FindTicket(>event,>seat,<ticket)[C:Event]
            2 CheckTicketAvailability(>user,>ticket,<available)
                  1 HasTheTicketBeenPurchased(>ticket,<purchased)[C:Ticket]
                  2? WhetherTheTicketWasReserved#(>ticket,<reserved)[C:Ticket] ⏎
                        /purchased == false
                  3? WhetherTheTicketWasReservedByTheUser(>user,>ticket,<reservedByUser) ⏎
                        [C:Ticket] ⏎ /purchased == false AND reserved == true
                    4 ReturnAvailability(>purchased,>reserved,>reservedByUser,<available)
            3? PurchaseTicket(>user,>ticket) /available == true
                  1 #WhetherTheTicketWasReserved(>ticket,<reserved)[C:Ticket]
                  2? RemoveReservation(>ticket)[C:Ticket] /reserved == true
                  3 Purchase(>user,>ticket)
      3 GetPurchasedSeatsForUser(>user,<purchasedSeats)[C:User]
      4 Output(>purchasedSeats,<=O.PS)
```

Listing 5 - The fifth level of decomposition in AFD for the ticket purchasing system

grades for experimental vs. control groups, on a scale 0-10, were as follows: 8.02 vs. 7.60, 7.95 vs. 7.75, 8.49 vs. 7.12. The results of the evaluation suggest that it could be expected that AFD could help communication between users and engineers and to smooth transition from requirements to specification. However, further improvements of AFD would be required in order to integrate it with the existing tools and paradigms.

5. CONCLUSION

Performance of a design and development methodology depends on its ability to provide easy understanding for users and sufficient information for engineers. This paper presents an approach that suggests using AFD language for software design and its automatic mapping to UML for the purpose of software development. AFD is a text-based language based on computational thinking that enables stepwise refinement of a software product design in order to make it more

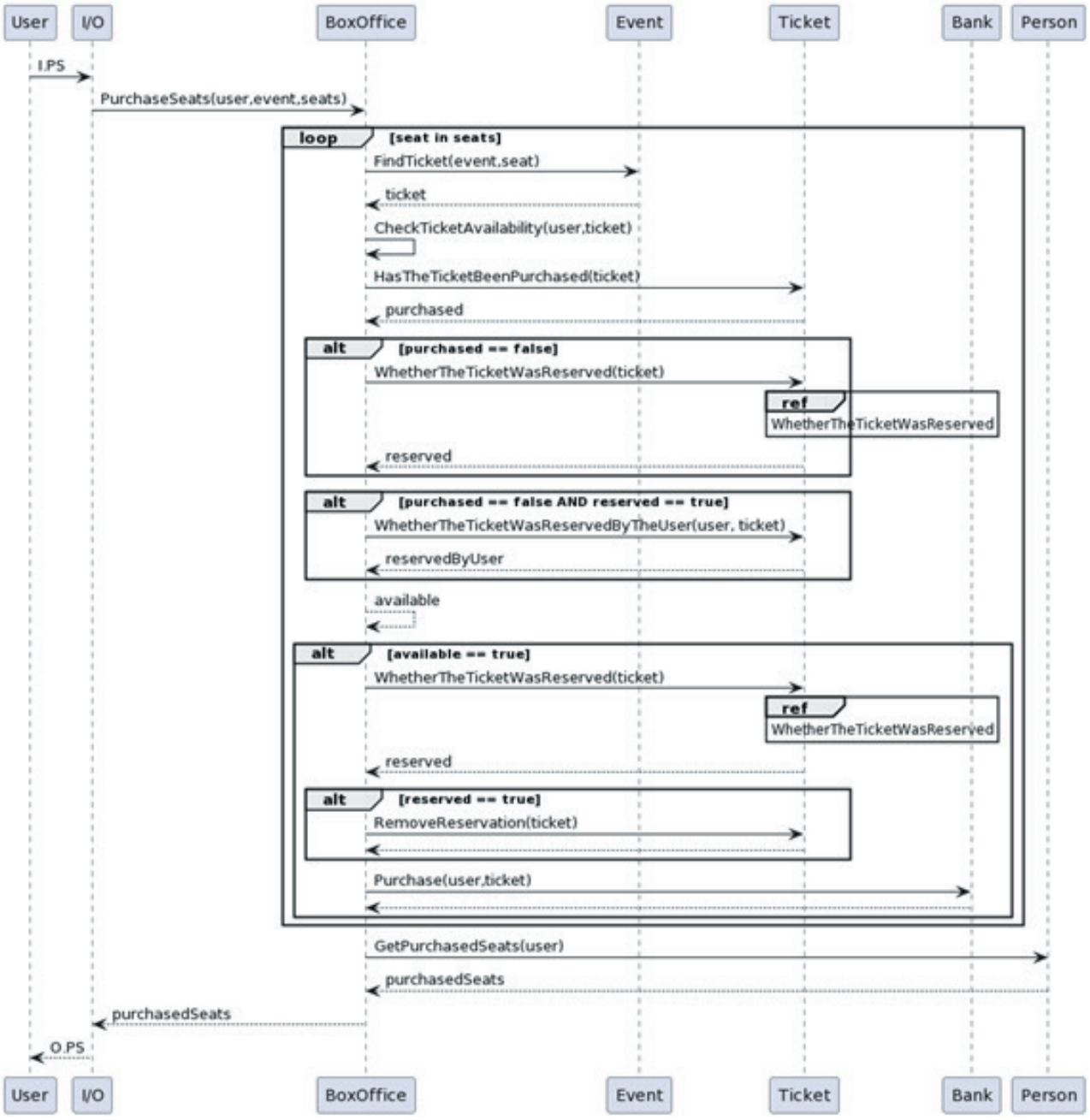

Figure 1 - Example of a corresponding UML for the ticket purchasing system designed in AFD (sequence diagram)

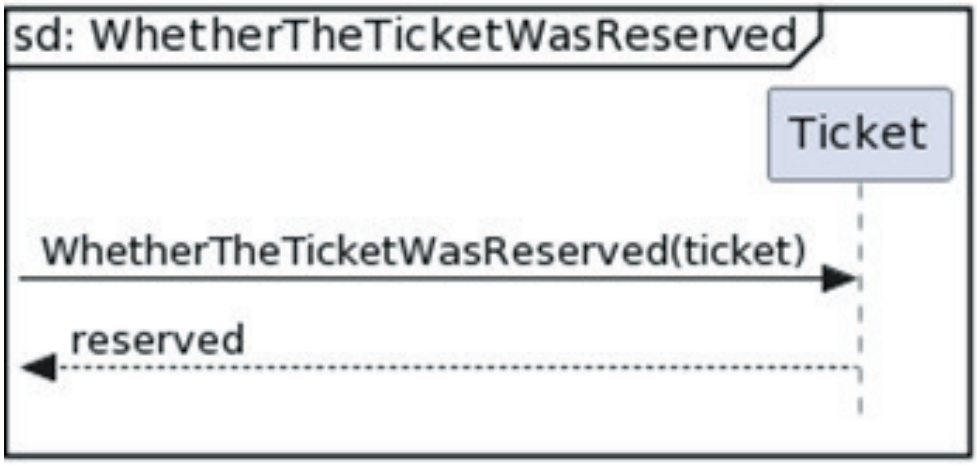

Figure 2 - Example of a corresponding UML for the ticket purchasing system designed in AFD (sequence fragment)

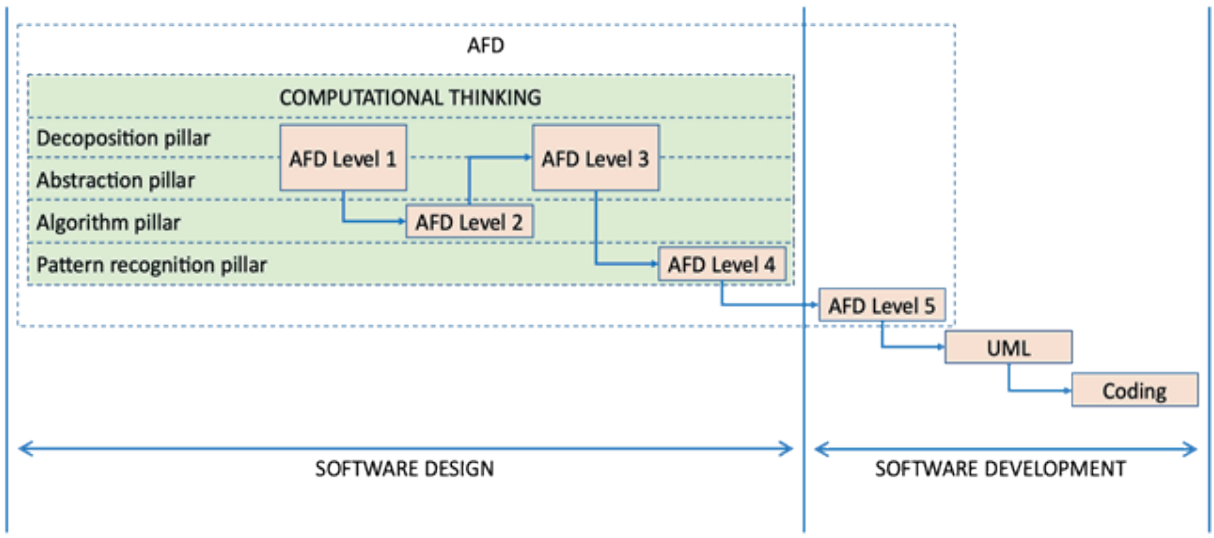

Figure 3 - Approach for software design and development

comprehendible for users and linking it to implementation details required by engineers in order to make software development more consistent with the design. The stepwise refinement is supported with multiple levels of decomposition in AFD. The approach suggests using the first four levels for a software design and using the fifth level for mapping the design to selected software development paradigm. The lower levels are predominantly intended for users for expressing the requirements while the upper levels are intended for engineers for deciding upon implementation details. AFD as a technology that, used alone or in conjunction with other available technologies, tends to help in production of more reliable and robust software products that fulfil end-users expectations. Having in mind influence of software industry on the global economy then each even minor step of improvement may have great value and importance.

6. ACKNOWLEDGEMENTS

Science Fund of the Republic of Serbia, Grant/Award Number: AVANTES; Ministry of Education, Science, and Technological Development of the Republic of Serbia, Grant/Award Numbers: III44009, TR32047. The authors are grateful for the provided financial support.

7. REFERENCES

[1] K. Curcio, T. Navarro, A. Malucelli and S. Reinehr, "Requirements engineering: A systematic mapping study in agile software development," *Journal of Systems and Software*, vol. 139, pp. 32-50, 2018.

[2] S. Goericke, The future of software quality assurance, Springer Nature, 2020.

[3] E. Brimhall, R. Wise, R. Simko, J. Huggins and W. Matteson, "A systematic process for functional decomposition in the absence of formal requirements," *INCOSE International Symposium*, vol. 26, no. 1, pp. 1204-1218, 2016.

[4] A. Labusch, B. Eickelmann and M. Vennemann, "Computational Thinking Processes and Their Congruence with Problem-Solving and Information Processing," in *Computational Thinking Education*, Springer Singapore, 2019, pp. 65-78.

[5] J. M. Wing, "Computational thinking's influence on research and education for all," *Italian Journal of Educational Technology*, vol. 25, no. 2, pp. 7-14.

[6] S. Tubić, M. Cvetanović, Z. Radivojević and S. Stojanović, "Annotated functional decomposition," *Computer Applications in Engineering Education*, vol. 29, no. 5, pp. 1390-1402, 2021.

[7] M. Ozkaya, "Are the UML modelling tools powerful enough for practitioners? A literature review," *IET Software*, vol. 13, no. 5, p. 338 – 354, 2019.

[8] M. Gogolla, F. Büttner and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Science of Computer Programming*, vol. 69, no. 1, pp. 27-34, 2007.

[9] M. Nawaz, T. Nazir, S. Islam, M. Masood, A. Mehmood and S. Kanwal, "Agile Software Development Techniques: A Survey," *Proceedings of the Pakistan Academy of Sciences: A. Physical and Computational Sciences*, vol. 58, no. 1, pp. 17-33, 2021.

[10] A. Jarzębowicz and P. Weichbroth, "A Qualitative Study on Non-Functional Requirements in Agile Software Development," *IEEE Access*, vol. 9, pp. 40458-40475, 2021.