Experiences in the development and use of Annotated Functional Decomposition

Zaharije Radivojević, Stefan Tubić, Miloš Cvetanović
School of Electrical Engineering, Belgrade University

Workshop
“The impact of pandemic years to informatics education: review and next steps”

Shkoder, Albania
4-8 September 2023

Agenda

- Background,
- Annotated Functional Decomposition,
- Computational Thinking,
- Example,
- Evaluation,
- Conclusion.

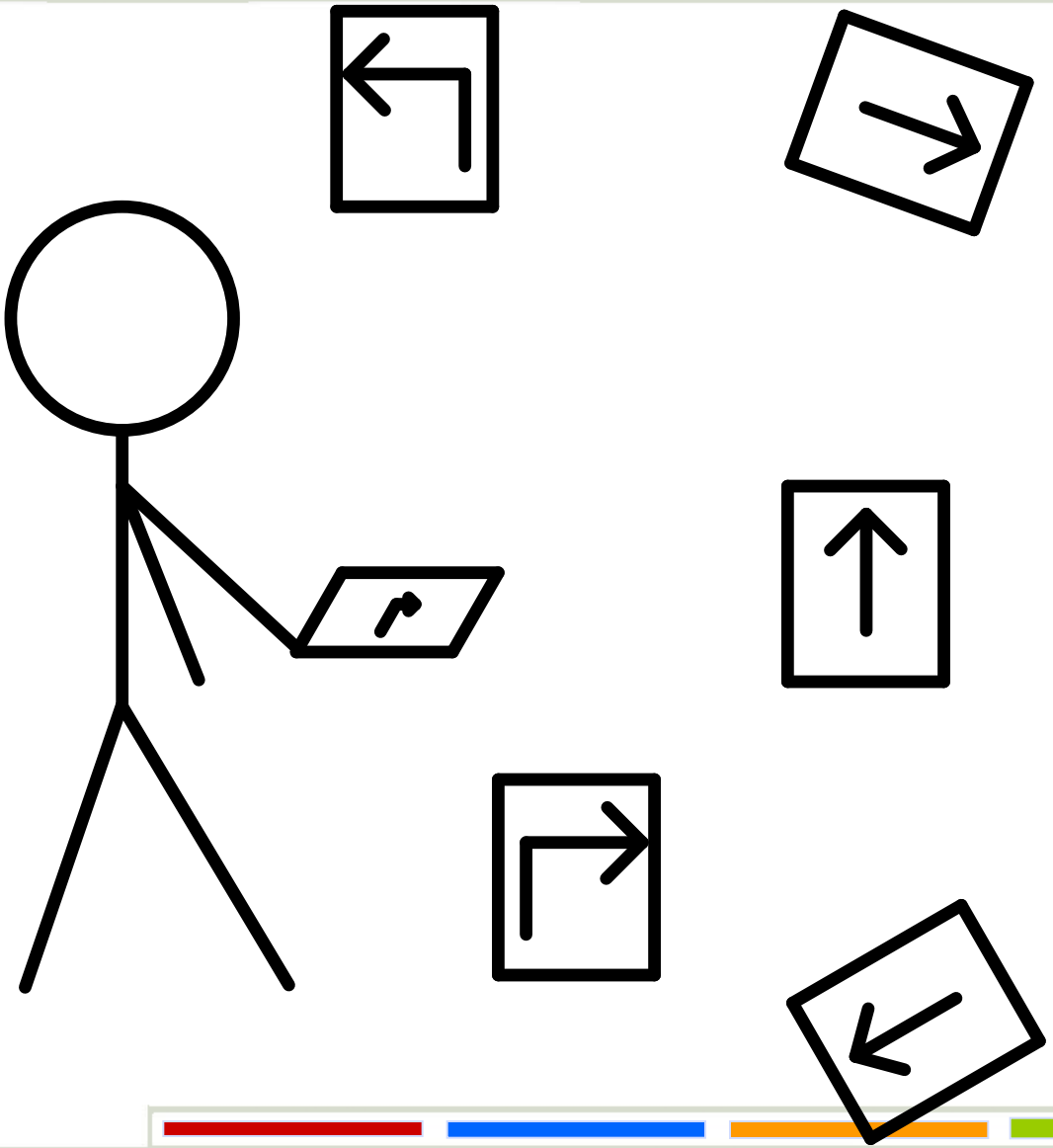
Background

The requirements gathering process

- The first step in a project for developing a software product,
- Takes 35% of the project time?
- Omissions are most costly to recover if noticed deep into the project.

- The important **problem** in requirements gathering is absence of a proper way of representing the functional requirements.

Requirements gathering



Background

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.

Annotated Functional Decomposition

- Annotated Functional Decomposition (AFD)
  is a new text-based language
- that resembles natural languages
  so, it is easy to use by clients,
- that supports some semantics of computer languages
  so, it is adequate for use by engineers.

- In both cases, the result is jeopardized expectations between clients and engineers,
  that leaves plenty of room for errors and ambiguities.

Annotated Functional Decomposition

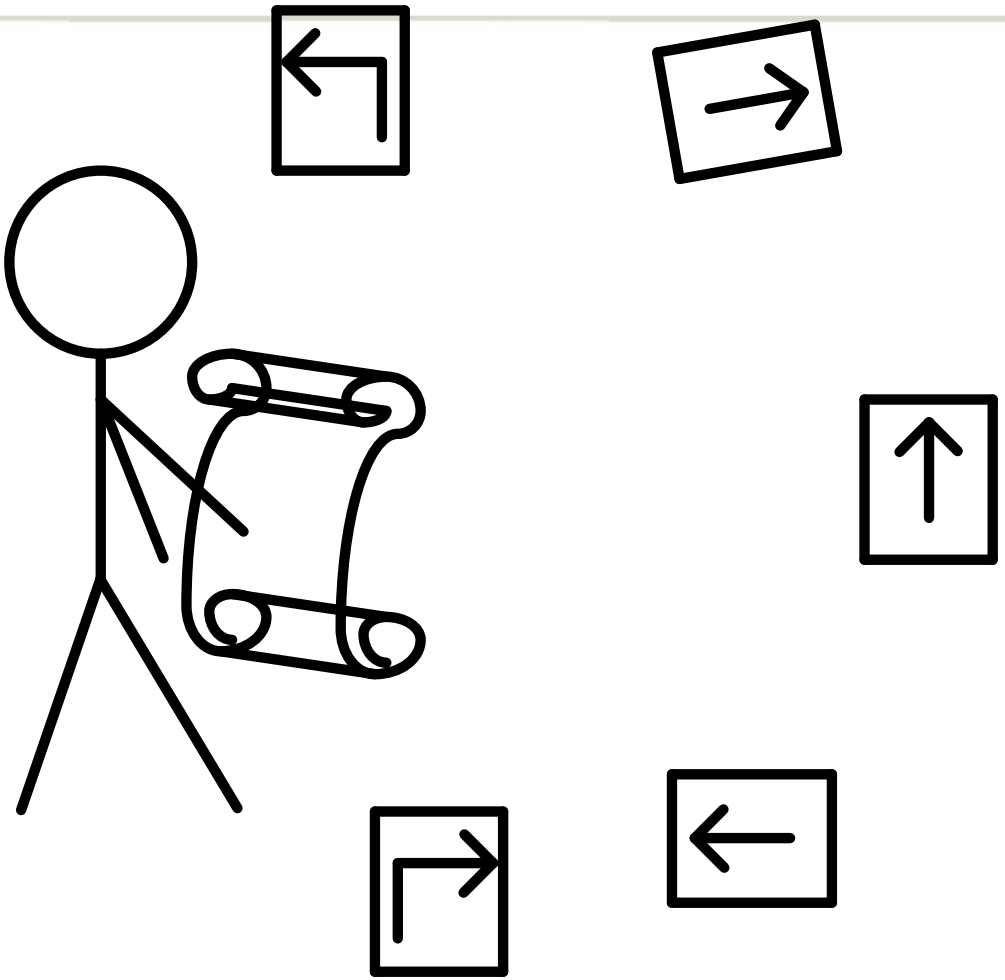
AFD provides solutions for two main problems

1. functional requirements usually do not adequately recognize all functionalities,
2. design specification does not meet all functional requirements.

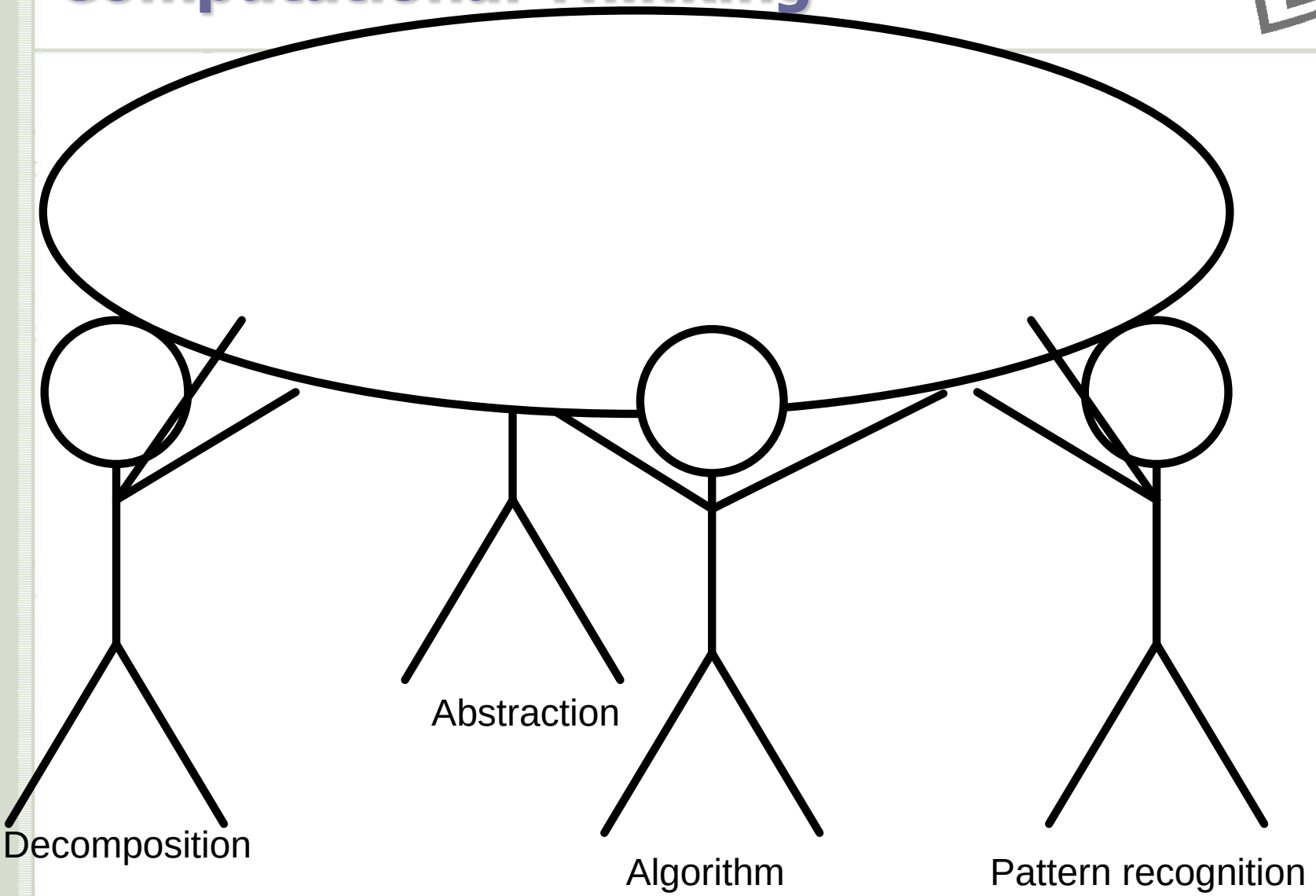
AFD is based on the existing design paradigm named Structured Design

AFD builds upon methodological concepts introduced by Computational Thinking

Structured Design



Computational Thinking



Computational Thinking

four pillars

1. Decomposition – breaking down complex problems into sub-problems
2. Abstraction – focusing only on the details that are important, ignoring the rest
3. Algorithm – creating steps or simple rules for solving each sub-problem
4. Pattern recognition – identifying similar problems which were previously solved

AFD Rules

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;
...	...

Example

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

Example

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

Example

```
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)
```

Example

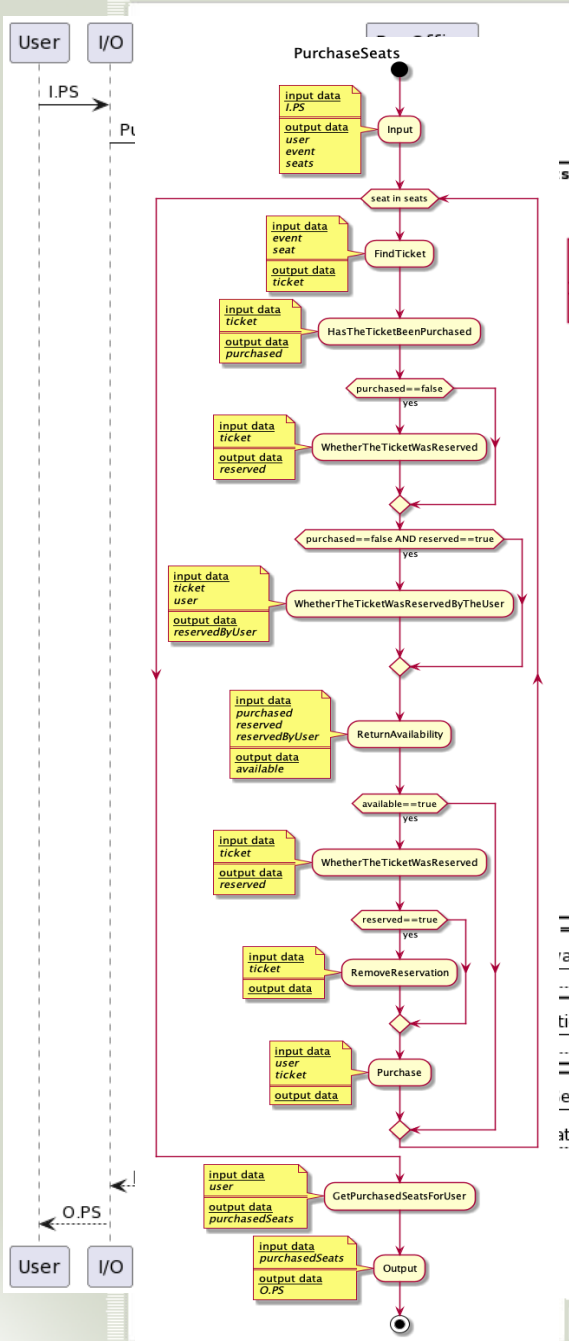
```
1 PurchaseSeats(=>I.PS,<=O.PS)
      1 Input(=>I.PS,<user,<event,<seats)
      2* PurchaseSeat(>user,>event,>seat) /seat in seats
               1 FindTicket(>event,>seat,<ticket)
               2 CheckTicketAvailability(>user,>ticket,<available)
                        1 HasTheTicketBeenPurchased(>ticket,<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)
```

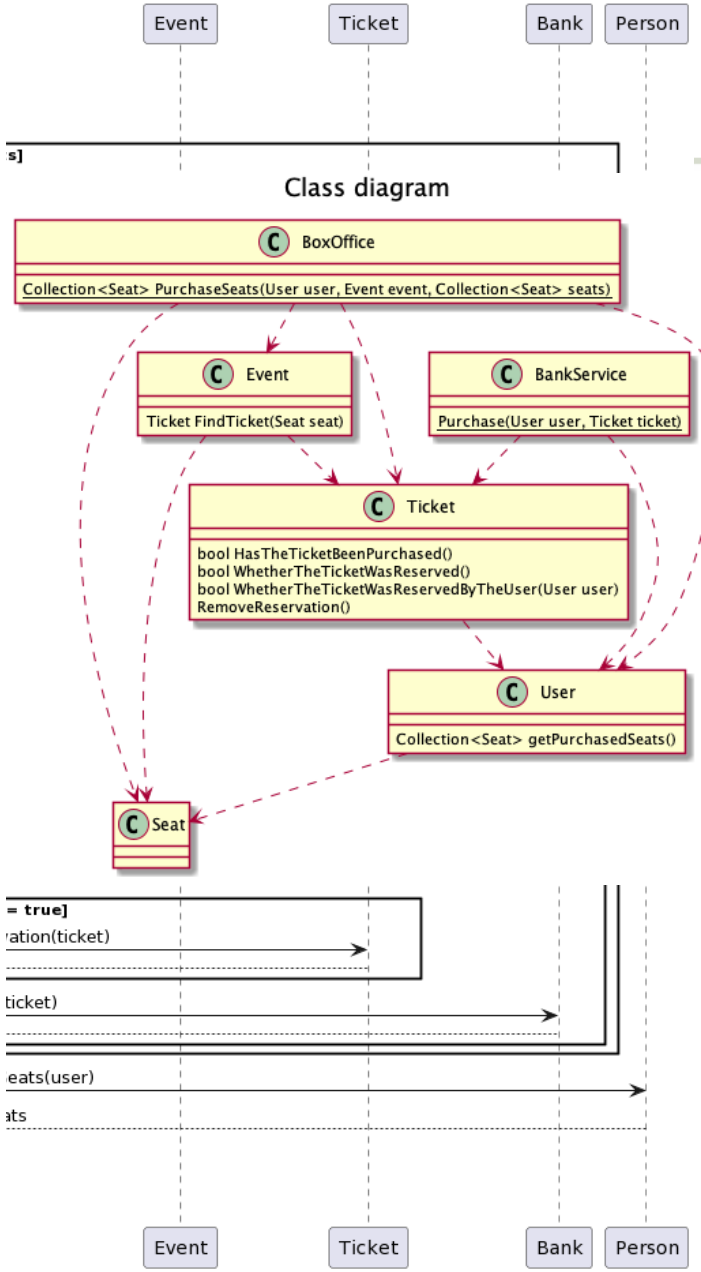
Example

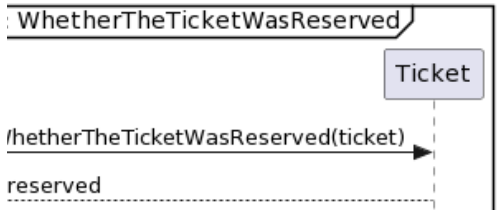
```
1 PurchaseSeats(=>I.PS,<=O.PS)[C:BoxOffice]
       1 Input(=>I.PS,<user,<event,<seats)
       2* PurchaseSeat(>user,>event,>seat) /seat in seats
                1 FindTicket(>event,>seat,<ticket)[C:Event]
                2 CheckTicketAvailability(>user,>ticket,<available)
                        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)
```

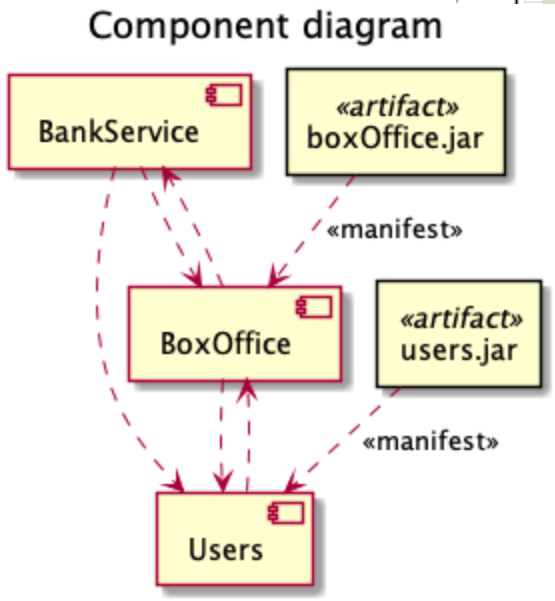
Example

```
1 PurchaseSeats(=>I.PS,<=O.PS)[C:BoxOffice(boxOffice.jar:BoxOffice)]^User@BoxOfficeServer:www.boxoffice.com
        1 Input(=>I.PS,<user:User(users.jar:Users),<event:Event(:BoxOffice),<seats:{}Seat)
        2* PurchaseSeat(>user,>event,>seat) /seat in seats
                1 FindTicket(>event,>seat:Seat(:BoxOffice),<ticket:Ticket(:BoxOffice))[O:event]
                2 CheckTicketAvailability(>user,>ticket,<available)
                        1 HasTheTicketBeenPurchased(>ticket,<purchased:bool)[O:ticket]
                        2? WhetherTheTicketWasReserved#(>ticket,<reserved:bool)[O:ticket] /purchased==false
                        3? WhetherTheTicketWasReservedByTheUser(>ticket,>user,<reservedByUser:bool)[O:ticket]
                                                /purchased==false AND reserved==true
                        4 ReturnAvailability(>purchased,>reserved,>reservedByUser,<available:bool)
                3? PurchaseTicket(>user,>ticket) /available==true
                        1 #WhetherTheTicketWasReserved(>ticket,<reserved:bool)[O:ticket]
        2? RemoveReservation(>ticket)[O:ticket] /reserved==true
                        3 Purchase(>user,>ticket)[C:BankService(:BankService)]^System@BankServer:www.bank.com
        3 GetPurchasedSeatsForUser(>user,<purchasedSeats:{}Seat)[O:user:getPurchasedSeats]
        4 Output(>purchasedSeats,<=O.PS)
```


User
I/O
PurchaseSeats
I.PS
input data
I.PS
output data
user
event
seats
Input
seat in seats
input data
event
seat
output data
ticket
FindTicket
input data
ticket
output data
purchased
HasTheTicketBeenPurchased
purchased==false
yes
input data
ticket
output data
reserved
WhetherTheTicketWasReserved
purchased==false AND reserved==true
yes
input data
ticket
user
output data
reservedByUser
WhetherTheTicketWasReservedByTheUser
input data
purchased
reserved
reservedByUser
output data
available
ReturnAvailability
available==true
yes
input data
ticket
output data
reserved
WhetherTheTicketWasReserved
reserved==true
yes
input data
ticket
output data
RemoveReservation
input data
user
ticket
output data
Purchase
input data
user
output data
purchasedSeats
GetPurchasedSeatsForUser
input data
purchasedSeats
output data
O.PS
Output
O.PS
User
I/O


Event
Ticket
Bank
Person
Class diagram
BoxOffice
Collection<Seat> PurchaseSeats(User user, Event event, Collection<Seat> seats)
Event
Ticket FindTicket(Seat seat)
BankService
Purchase(User user, Ticket ticket)
Ticket
bool HasTheTicketBeenPurchased()
bool WhetherTheTicketWasReserved()
bool WhetherTheTicketWasReservedByTheUser(User user)
RemoveReservation()
User
Collection<Seat> getPurchasedSeats()
Seat
= true]
ation(ticket)
ticket)
eats(user)
ats
Event
Ticket
Bank
Person


WhetherTheTicketWasReserved
Ticket
hetherTheTicketWasReserved(ticket)
reserved


Component diagram
BankService
«artifact»
boxOffice.jar
«manifest»
BoxOffice
«artifact»
users.jar
«manifest»
Users

Preliminary quantitative evaluation

1. Whether using AFD facilitates focusing on required logical checks in comparison to UML
2. Two groups of students, those who uses AFD (experimental), and UML (control)
3. Three experiments each one involving more than 100 students in total

Problem no.	Group	No. of students	Average grade on a problem (0-10)	Average grade during studies (6-10)
1	AFD	113	8.022	8.345
	UML	144	7.608	8.334
2	AFD	91	7.956	8.457
	UML	103	7.757	8.304
3	AFD	49	8.490	7.928
	UML	85	7.124	8.547

Preliminary quantitative evaluation

1. Two-way ANOVA test,
   - The main effect of the observation group factor is statistically significant (F (1, 579) = 16.007, P = 0.000)
   - The main effect of the observation problem factor is not statistically significant (F (2, 579) = 0.036, P = 0.964)
2. Non-parametric Kruskal-Wallis tests,
3. Experimental groups achieved higher average score than control groups.

Conclusion

- AFD is a text-based language based on computational thinking that enables stepwise refinement of a software product specification
- An approach that suggests using AFD language for software design (the first four levels) with an automatic mapping to UML diagrams for software development
- Preliminary study shows that AFD facilitates problem-solving in the domain of a logical design of information systems (the students who used AFD achieved higher average grades than those who used UML for solving the same problems)
- Plans for extending AFD and introducing new levels of decomposition in order to enable mapping with multiple UML diagrams

Thank you!

Zaharije Radivojevic