Using *i** to Describe Data Structures

Xavier Franch[0000-0001-9733-8830]

Universitat Politècnica de Catalunya, Barcelona
franch@essi.upc.edu

Abstract. This paper explores the use of the *i** language as a notation to describe data structures to be used in classical imperative programs written in e.g. Java or C#. Data structures are described at two levels of abstraction, their specification and their implementation (the data structure properly said). We analyze how iStar 2.0, enriched with both modularization and dependum specialization constructs, can be used in this context.

Keywords: iStar; iStar2.0; *i**; Data Structures; Software Selection.

1 Introduction

The study of data structures is almost as established as the study of programming and in particular, imperative programming. In the seventies and eighties, with the advent of imperative programming languages as C, Pascal, Ada and Modula, they were object of intensive analysis that resulted in classical textbooks by D.E. Knuth [15][16], A.V. Aho *et al.* [1] and N. Wirth [24], among others. These authors described at a great level of detail the basic types of data structures, from lists to trees to hash tables. Gradually, other authors proposed more sophisticated and specialized data structures (Fibonacci heaps, splay trees, etc.) with more specific purposes; most of them are very well-summarized in another classical textbook by T.H. Cormen *et al.* [5].

In the same period of time, as a response to the observation by J.B. Morris "[data] types are not sets" [21], another stream of research formulated the concept of abstract data type (ADT) [13]. A central concept formulated by the theory of ADTs is the distinction among the specification of an ADT (their operations and the properties they fulfil) and its implementation (the operationalization into code), linked by the so-called abstraction function [14]. Typically, the implementation of an ADT takes the form of a data structure. The most important consequence of this (at that time) revolutionary approach was that programmers had the opportunity to manipulate a data structure through its operations and not directly through its internal representation (in form of arrays, pointers, records, etc., offered by the programming language), improving then the reliability of the resulting code as well as its maintenance. In fact, ADTs together with object-orientation boosted modular development as the usual way to develop programs in imperative programming [17]. This approach was also reinforced by a consolidated portfolio of the most popular ADTs and data structures, promoting reuse as a fundamental principle to modular programming.

In the context of modular programming with reuse of ADTs and data structures, two questions arose when it is the moment to select data structures:

1) Which is the most appropriate ADT to solve the problem at hand? Typical ADTs are lists, mappings, graphs and trees. The answer to this question was the basis of component reuse by e.g. signature matching [26].
2) What is the data structure that best implements the chosen ADT? This second problem was addressed through the incorporation of ad hoc constructs in new languages like CLU [17] and SETL [23] that remained as research endeavours without industrial impact. In this direction, we also designed a language called NoFun [8][2] which we intensively used in the design of a comprehensive catalogue of ADTs implemented with popular data structures [7]. NoFun was an acronym for "Non-Functional", reflecting the fact that the main criteria to choose among alternative implementations of an ADT where related to non-functional requirements, in contrast to the problem of ADT/component reuse, based on the functional requirements.

Even if NoFun was a powerful notation to describe the specification of ADTs and the main characteristics of the data structures implementing them, it was quite low level. For instance, efficiency in time and space was measured by giving order of magnitude values using the asymptotic functions big-Oh and similar [15], which may be necessary to perform accurate analysis, but cumbersome when a more lightweight approach suffices, e.g. to summarize the main characteristics of the data structure in a teaching context.

This paper proposes to address the problem of data structure selection (thus, implicitly, of ADT selection) using the *i** framework [25] and in particular, the framework's language. We use iStar 2.0 [6] as such language (although the use of one version or another is not critical for this work) and, to emphasize the modularity of the approach, we will present the different models as modules that can be reused following the guidelines given in [11] and evaluated in [20]. Following the ideas in [9], the use of *i** is twofold, on the one hand for describing the data structures and on the other hand, for matching them to a particular situation. In this paper, we will focus on the first problem. Therefore, we may state the main goal of this paper as: to **analyse** *iStar 2.0* **for the purpose** of *understanding whether it helps describing data structures in imperative programs* **with respect to** *functional and non-functional characteristics* **from the point of view of** *programmers and educators* **in the context of** *modular software development.*

From this goal, according to the distinction among specification and implementation of an ADT, we derive two research questions:

RQ1: How can iStar 2.0 be used to describe the specification of ADTs?

RQ2: How can iStar 2.0 be used to describe data structures that implement ADTs?

These two research questions are investigated in the next two sections.

2 Describing ADT Specifications using *i**

Fig. 1 and Fig. 2 show[1] the SD diagrams of two classical families of ADT, namely sequences and functions, respectively, encapsulated in modules. Dependencies that stem from the outside are in the form of goals to express high-level functionality required by the ADT; according to [11], missing actors allow to connect these dependencies to the right actors when the module is reused in a particular context. Fig. 1 shows three types of sequences: stacks, queues and lists. All of them share in common the goal of providing individual elements' addition and removal and the possibility to access one designated element, while lists in addition offer the functionality of traversing the full sequence (i.e., getting the elements in some order). Fig. 2 shows two types of functions: sets and mappings. Both of them provide functionalities of element addition and removal too, and the main difference comes from the access operation: while sets offer membership, mappings allow to look up the value associated to a given key. In fact, the existence of key is a requirement (in the form of a resource) from the ADT to the (unknown) depender.

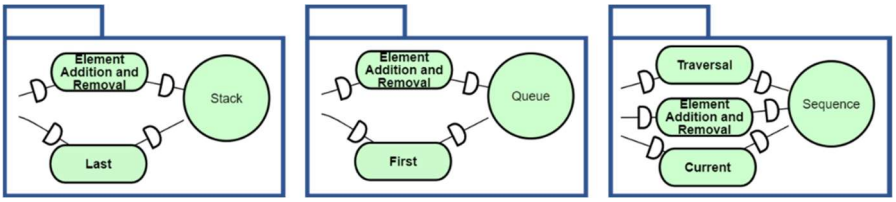

Fig. 1. Modules showing the SD diagram of sequence ADTs

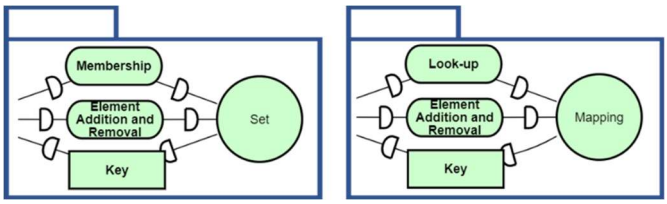

Fig. 2. Modules showing the SD diagram of function ADTs

Given that the ADTs of each type look similar, we can use specialization and arrange them into hierarchies of related modules. Fig. 3 and Fig. 4 show the two hierarchies for the two types of ADTs following the proposal by López et al. [19][2]. The (abstract) superactor `Sequence` defines the common traits to every sequence that are tailored by each of the three types, redefining the `Get` dependency conveniently and, in the case of lists, adding the traversal goal. The `Function` superactor has a similar situation. Going further, in Fig. 4 we also show how the concept may be used to defined more complex ADTs by specialization, e.g. sets offering further functionalities.

[1] All diagrams have been drawn with the piStar tool, https://www.cin.ufpe.br/~jhcp/pistar/, and manually modified to add modularity and specialization constructs not included in iStar 2.0.

[2] We also use colours to highlight the elements affected by the specialization.

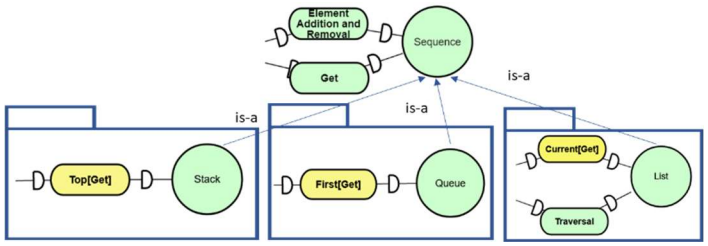

Fig. 3. Hierarchies of actors for sequences.

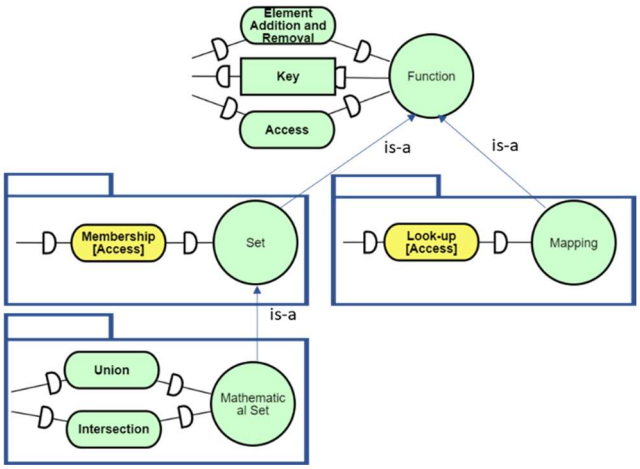

Fig. 4. Hierarchies of actors for functions.

3 Describing ADT Implementations by Data Structures using *i**

While the description of ADT specification is related to the functionalities offered (or sought, depending on the perspective), when it comes to data structures, non-functional requirements come into play, which calls for the use of iStar 2.0 qualities. Not only this, but also given the lower level of abstraction, tasks are more likely to appear. To focus the discussion, we select the most typical non-functional concept that is explored in data structure design, namely time efficiency, although others as space efficiency, maintainability or understandability could be also of interest.

Fig. 5 shows the simplest expression of SD diagrams for two popular data structures that implement mappings, namely hash tables and AVL trees. Both of them share in common the capability of providing fast access to look-up operations, and the main difference is on what they require in order to be properly implemented. Please note also that we do not include the information already provided in the corresponding specification. Hash tables need to know an approximate size of the data structure (i.e., number of elements to be stored), and in addition a hash function. Although the modularity approach proposed in [11] does not allow to distinguish roles outside the module, we can expect that the actors informing about the size and implementing the hash function will not be the same (or at least, they are not required to); we have tried to enforce this

statement in the drawing, placing the dependencies pointing to different places. Concerning AVL trees, they do not need an estimation of size but have another important requirement: the ability to order the elements, which we represent with the `Greater or equal than` task. As it happened with the specification, we could make use of inheritance to capture the commonalities that every implementation of mappings should have, in this case fast element look-up. This is shown in Fig. 6, which also shows the use of a *participates-in* link to represent explicitly the link among an implementation and its corresponding specification.

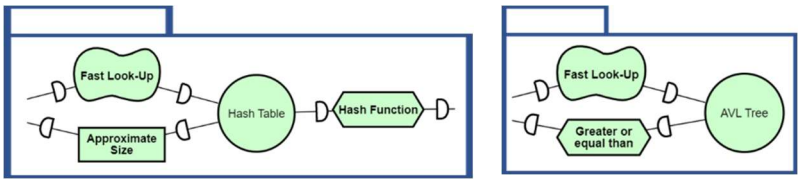

Fig. 5. Two implementations for mappings: hash tables and AVL trees.

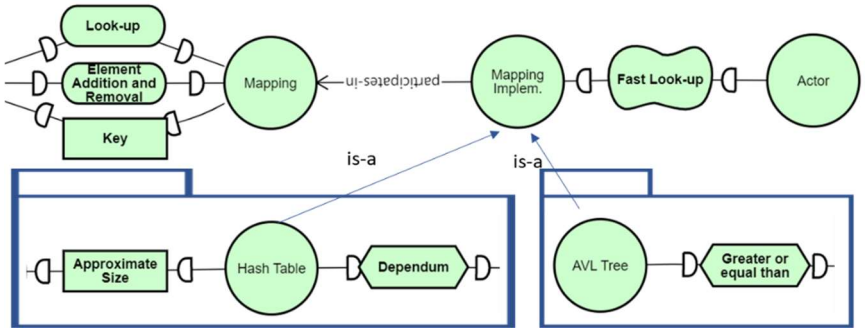

Fig. 6. Structuring the implementation of mappings.

4 Discussion

The main contribution of this paper is to provide a high-level view to the concept of data structure using *i**. This provides the basis for an instantiation of the classical software component selection problem [3], as we have already tackled in the past with *i** in the domain of commercial software packages [4] together with the use of metrics for informing this selection [10][12]; this is part of our next steps. It could be argued that, while selection of software packages is still a contemporary challenge, it is not the case for data structures. This is partially true, since a trend in modern programming languages like Python is to embed some basic data structures. However, wide communities of popular programming languages like Java or C# still need to integrate data structures in their programs. In addition, the increased specialization on software demands results in the need of creating data structures responding to specific challenges. For instance, G. Navarro proposes a catalogue of "compact data structures", aimed at storing in a compressed form large quantity of data still allowing to access and query them in compact form [22]. Another specialized field of research is the need of managing concurrent data structures [18], which raises specific goals and qualities related to performance.

Adopting a high-level, goal-oriented approach as presented in this paper could help these communities to organize their data structures in a more structured form.

Other aspects have not been addressed in this paper. For instance, the possibility of refining the very general qualities, as `Fast Look-up`, into more specific ones. In this case, typically statements about the asymptotic cost of the operations [15]. We could represent these statements as qualities too, e.g. `Big-Oh is log(n)`, related through contribution links to the former ones, or else we could define an extension of iStar 2.0 introducing these concepts that are specific to data structure analysis.

References

1. Aho, A., Hopcroft, J., Ullman, J.: *Data Structures and Algorithms*. Addison-Wesley, 1983.
2. Botella, P., Burgués, X., *et al.*: Modeling Non-Functional Requirements. JIRA 2001.
3. Carvallo, J.P., Franch, X., Quer, C.: Determining Criteria for Selecting Software Components: Lessons Learned. *IEEE Software* 24(3), 2007.
4. Carvallo, J.P., Franch, X., Quer, C.: Defining a Quality Model for Mail Servers. ICCBSS 2003.
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: *Introduction to Algorithms*. The MIT Press, 1990.
6. Dalpiaz, F., Franch, X., Horkoff, J.: iStar 2.0 Language Guide. arXiv preprint arXiv:1605.07767, 2016.
7. Franch, X.: *Estructuras de Datos: Especificación, Diseño e Implementación*. Ed. UPC, 1993.
8. Franch, X.: Systematic Formulation of Non-Functional Characteristics of Software. ICRE 98.
9. Franch, X.: On the Lightweight Use of Goal-Oriented Models for Software Package Selection. CAiSE 2005.
10. Franch, X.: A Method for the Definition of Metrics over *i** Models. CAiSE 2009.
11. Franch, X.: Incorporating Modules into the *i** Framework. CAiSE 2010.
12. Franch, X., Grau, G., Quer, C.: A Framework for the Definition of Metrics for Actor-Dependency Models. RE 2004.
13. Goguen, J.A., Thatcher, J.W., Wagner, E.G.: An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In *Current Trends in Programming Methodology*, *Vol. IV*, Prentice-Hall, 1978.
14. Hoare, C.A.R.: Proof of Correctness of Data Representation. *Acta Informatica* 1(1), 1972.
15. Knuth, D.E.: *The Art of Computer Programming, Vol. 1*. Addison-Wesley, 1968.
16. Knuth, D.E.: *The Art of Computer Programming, Vol. 3*. Addison-Wesley, 1973.
17. Liskov, B.H., Guttag, J.V. *Abstraction and Specification in Program Development*. MIT, 1986.
18. Liu, Z., Calciu, I., Herlihy, M., Mutlu, O.: Concurrent Data Structures for Near-Memory Computing. SPAA 2017.
19. López, L., Franch, X., Marco, J.: Specialization in the iStar2.0 Language. *IEEE Access* 7, 2019.
20. Maté, A., Trujillo, J., Franch, X.: Adding Semantic Modules to improve Goal-Oriented Analysis of Data Warehouses using I-star. *Journal of Systems and Software* 88, 2014.
21. Morris, J.B.: Types are not Sets. PoPL 1973.
22. Navarro, G.: *Compact Data Structures*. Cambridge University Press, 2016
23. Schwartz, J.T., Dewar, R.B.K., Dubinsky, E., Schonberg, E.: *Programming with Sets: An Introduction to SETL*. Springer, 1986.
24. Wirth, N.: *Algorithms and Data Structures*. Prentice-Hall, 1983.
25. Yu, E.: Modeling Organisations for Information Systems Requirements Engineering. ISRE 93.
26. Zaremski, A.M., Wing, J.A.: Signature Matching: A Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology* 4(2), 1995.