Journal of Computer Science 1 (1): 98-102, 2005
ISSN 1549-3636

A Note on the Role of Abstraction and Generality in Software Development

Pavol Návrat and Roman Filkorn
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies, Slovak University of Technology
Ilkovicova 3, SK-84216 Bratislava, Slovakia

Abstract: Although the evolving field of software engineering introduces many methods and modelling techniques, we conjecture that the concepts of abstraction and generality are among the fundamentals of each such methodology. This study proposed a formal representation of these two concepts, along with a two-dimensional space for the representation of their application. Based on the examples, we further elaborate and discuss the notion of abstraction and generalisation transformations in various domains of software development.

Key words: Abstraction, Generality, Software Development

INTRODUCTION

The field of software engineering gradually matures and proceeds towards one of its destiny–towards the mass production of quality software products. Many well-studied and broadly accepted principles and methodologies of software development are enhanced and arranged in entirely new ways. Although the multi-paradigm approaches introduce new and extended methods, techniques and models for various aspects of software development process, one of the assumptions of their successfulness is tight to their interpretation of common fundamentals–abstraction and generality.

We conjecture that abstraction and generality play one of the central roles among the fundamental concepts in software development. It is the notion of abstraction and generality, the form of their application and representation, which is an essential aspect that makes the paradigm usable and used.

For the purpose of our examples, we often use the word "concept" instead of the terms like "object" and "class", which are often tightly coupled with the semantics borrowed from object-oriented paradigm. We do not attempt to define exactly various meanings of word "concept"; we suppose that the reader is with the term familiar [1].

One of the most widely used architecture for web applications is an architectural pattern known as Model-View-Controller [2] (Hierarchical MVCI) [3]. It divides an application into three separate modules: the model contains data and problem-domain functionality, the view displays information to the user and the controller handles user input. Each module of the MVC architecture represents a distinct point of view–an abstraction, focusing on a particular concern of the application (concepts and relationships relevant for the concern, respectively). On the other side, the definitions of the modules are general enough to allow a set of specifying transformations into any special environment. We will take a closer look on both abstraction and generality as fundamental axes of concept transformations.

Abstraction and Generality in Software Development: One of the most important issues in software development is managing complexity. Each software development methodology or process proposes some mechanisms for such management. One fundamental concept, common in these methodologies, is the concept of abstraction. Another one, useful not only in reusing what has been developed before, is generality. Before further elaboration, we take a closer look at both concepts.

Abstraction: A software developer cannot deal with more than a few concepts and their relationships simultaneously. An abstraction allows suppressing details that are unimportant to him/her and emphasize the important information [4]. Abstraction helps manage the intellectual complexity of software [5]. Abstracting means that a higher-level concept disregards certain properties altogether [6, 7].

Abstraction can be undoubtedly viewed as one of the most fundamental principles that take into account (and are applied) in the software development process. It has been claimed that the process should be abstraction (de-abstraction, respectively) driven (similar to "stepwise refinement" originally presented by Dijkstra).

A lot of information about a system in development is collected in various types of models. Every model can be viewed as a formalised abstraction of the system, is concerning on a selected set of system's features and characteristics. In a simplified view, a model can be described by its intent and a set of predefined mechanisms to help fulfill the intent. The set defines representation formalisms, in other words, what type of

entities, concepts and relationships to stress out of the system. In a complementary definition, it describes what unimportant details are to be abstracted away.

As an attempt towards better formalization, we define for the purpose of the study operations of abstracting and concretising as follows.

Abstracting is a transformation that moves e (an entity in a model, e.g. class in class model, concept in conceptual model, or even whole model representation as an entity in software development process) to a more abstract e' by suppressing some details from e:

$\mathbf{A}_{details}(e) = e'$

Where, e' is e with some details omitted.

The inverse transformation to abstraction is concretising. Concretising is adding new details to an (more or less) abstract e:

$\mathbf{C}_{details}(e) = e'$

Where, e' is e with some details added.

By application of these operators, there may be distinct details omitted/added that are indicated as parameters of abstraction/concretisation. Details depend (not only) on the focus, the granularity of the entity they are applied on, the formalised representation of the entity and determine what of the entity should be abstracted/concretised. We further elaborate some examples of operators, their application on entities and possible distinct added/omitted details in this study.

A simple example of different levels of abstraction is in class-like diagram in Fig. 1. At the most abstract level, no details about class Person are present/modeled. At a less abstract level (after transformation $\mathbf{C}_{Personal}$), methods getName, addChild and getChildren model behaviour of the class and attributes name and children add some state to class objects. At the bottom, after application of $\mathbf{A}_{attributes}$ transformation, more abstract layer of class person is presented, with its attributes hidden. Such a class Person declares only its behaviour in the form of abstract methods.

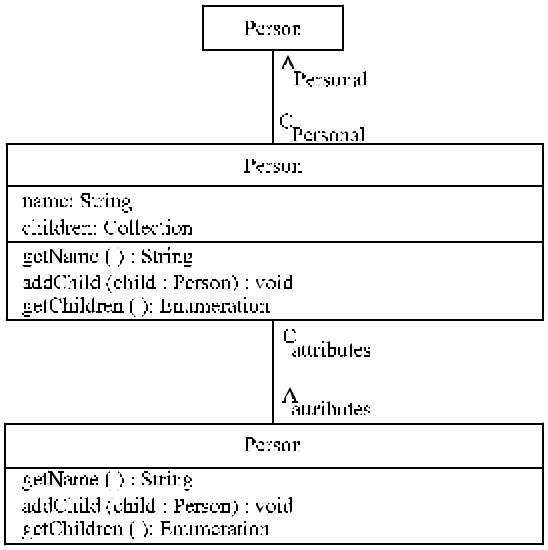

Fig. 1: Different Levels of Abstraction of Class Person

Generality: Generality can be characterized as a state or quality of being not limited to one particular case. Generalisation, as an inductive process, collects information about a number of particulars and presents it in a single description [1]. General description collects a set of individuals according to the features that are applicable to the whole as well as to every member of a set.

As noted in Navrat [7], generalisation moves things along the set-superset relation. Specialization, as an opposite operation, moves things in set-subset relation. Less general description determines a smaller set of things, while its transformation to a more general description will determine a superset of the original set. Every member of less general set is a member of more general superset.

For the purpose of the study, we define generalisation and specialising as follows:

Generalising is a transformation that moves entity e to a more general e' in such a way that all the features and dimensions of e' remains present in e, but at least some of the features or dimensions are less constrained in e';

$\mathbf{G}_{features}(e) = e'$

Where, e' is e with some features/dimensions less constrained.

Specialising is an inverse transformation to generalising. It produces a specific case e' as one of possible variations of e by constraining some features or dimensions of e;

$\mathbf{S}_{features}(e) = e'$

Where, e' is e with some features/dimensions more constrained.

Generality is achieved typically by introduction of some sort of loose definition to properties or characteristics. Such generality constitutes a variation point, which allows several distinct alternatives to occur at the pre-defined point (often referred to as parameter, or hot spot in [5]). Variation point consists of one or a combination of features, that are already present in the description (have been concretised in some previous transformation), but their values are not further specified. The property representation is less constrained or the value is abstracted at all.

As an example of specialisation, consider distinct kinds of Person depicted in Fig. 2. The most general notion of Person has three features: age, sex and position. Each of these features is a variation point and is candidate for further specialisation of Person. Feature age is constrained in the interval <0,150>, sex can obtain a

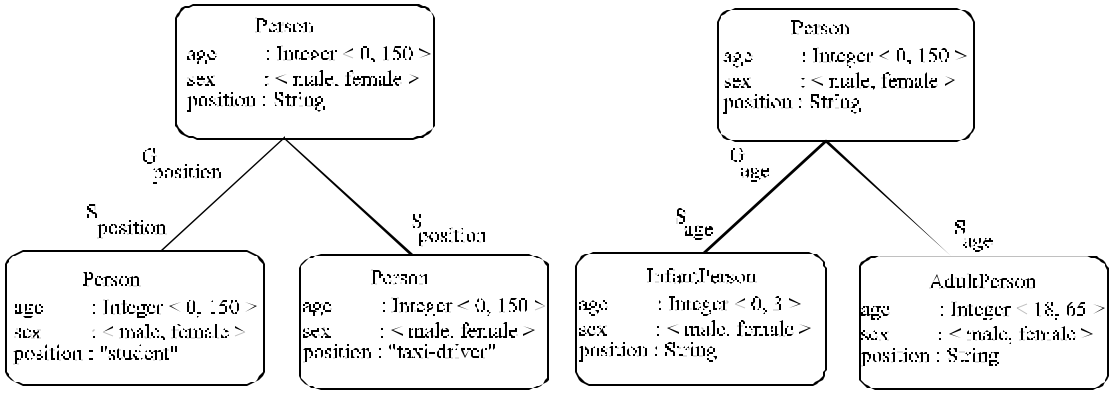

Fig. 2: Specialization of Features of Concept Person

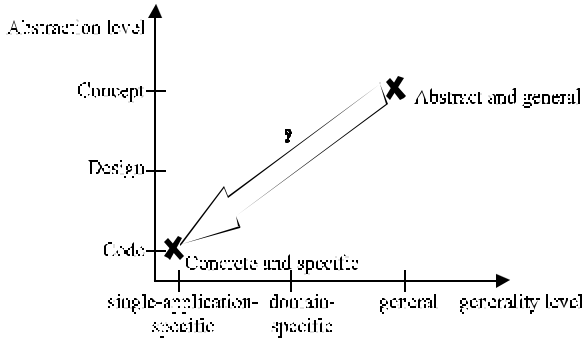

Fig. 3: Space of Abstraction and Generality [10]

value from enumeration <male, female> and position is of type String, with no value assigned. On the left side of the diagram, specialisation of feature position ($\mathbf{S}_{position}$) is applied, resulting in two special cases of Person- "student" and "taxi-driver". On the right side of the diagram, variation point age is further constrained ($\mathbf{S}_{age}$), resulting in two less general cases of Person. If the age specialisation plays important role in the problem domain, these new classes obtain unique names, InfantPerson and AdultPerson. Another point of possible discussion in the interval defined for adultness of person–exact boundaries (18, 21, or else) are the subject of problem domain, too.

Generality is an inevitable characteristic of reusable assets. Things get reusable only if they are general and allow turning to specific in a clear and straightforward manner. A difficult part of finding appropriate generalisations is to find a balance between a natural tendency to over-generalise (which may result in an introduction of unnecessary variation points and thus superfluous complexity) and a risk that some of the relevant variation points will not be identified [1].

There are usually several candidates for variation points which can be specified in the next step of entity's transformation. Although there is not any exact order defined for specialisation of these variation points, the order itself might play a significant role. It is often some sort of knowledge that captures proposed order of transformations and forms the content of a software development process definition. Such knowledge is mostly in a form of natural language sentences and a lot of research is put in its further formalisation [8, 9] for concept of design pattern [3].

Abstraction vs. Generality: Since it is very common to confuse abstraction and generality, we believe the issue is worth discussion. Although the entire complexity of the relation between abstraction and generality is beyond the scope of this article, we present one simple view on it in the problem domain of previous examples. Abstraction and generality are undoubtedly two distinct concepts: while abstraction is concerned with the amount of details present in an entity, generality makes things usable in a wider context. Abstraction disregards certain properties altogether, generality groups individuals according to the features they all share. Concretisation selects important features from the problem domain into the model, while specialisation helps classification of entities according to those selected (concretised) properties.

One possible reason of the confusion between abstraction and generality may be bound with the specialisation. By specialising, there usually arise new opportunities to include additional details into the more specific entity.

Let us consider an example of further transformation of the concept Person. It may be specialised into AdultPerson by restricting age to be a value in interval <18,65>. Once we have this, it is possible to find further characteristics that are common for the AdultPerson (and not for the Person) such as (collection of) its children or id-card number. We can add them by concretising as a new detail to AdultPerson. Usually, both transformations are joined into a single step, resulting in same set of features, but without a clear realisation of both the involved transformations.

Space of Abstraction and Generality: For the purpose of clarity and simplicity, let us reduce software development process to one special case–stepwise

Journal of Computer Science 1 (1): 98-102, 2005
ISSN 1549-3636

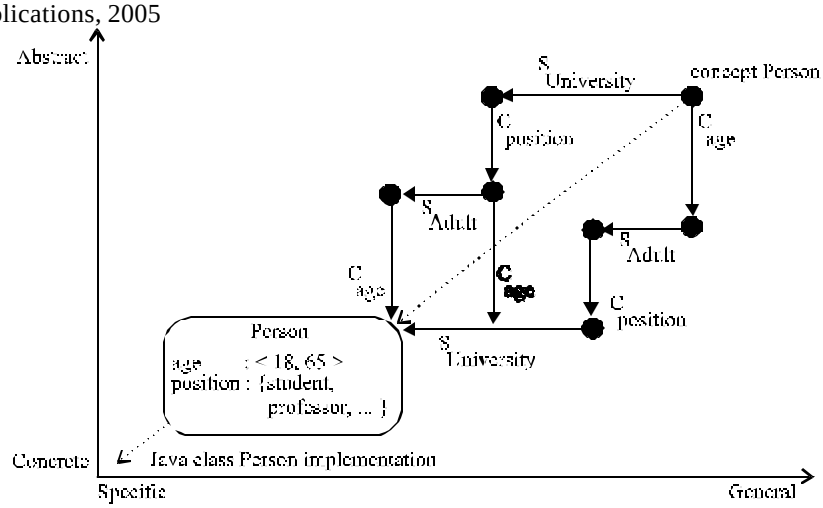

Fig. 4: Two-dimensional Space of Abstraction and Generality

development from domain modeling to class implementation in object-oriented manner. In such a process, each concept from problem domain is a subject to a transformation that essentially alters its level of abstraction or generality. The course of application of specialisation and concretising operators transforms the concept representation to mechanisms defined for solution domain.

In Fig. 3, we showed the transformation space as a simplified, two-dimensional space of abstraction and generality. Abstraction levels vary along the vertical axis with concept at the high-level abstraction level and code listed at the low-level end of the axis. Generality level that is indicated by the horizontal axis ranges from general to single-application-specific.

In this space, we have depicted two significant states: a general and abstract one in the top right corner and a concrete and specific one in the bottom left corner. Abstract and general state corresponds to a general definition of a concept in problem domain. Concrete and specific state represents its class/object representation, which is embedded in a software system's implementation.

Concept implementation is a process of transforming its general and abstract representation (valid in problem domain) to a specific and concrete representation (based in solution domain). However, it does not proceed as straightforwardly as the arrow shown in Fig. 3 might suggest. It is a systematic step-by-step process, composed of a series of transformations, most of them directed towards concrete and specific.

Possible concretising and specialisation steps for the concept Person are shown in Fig. 4. Starting with abstract and general representation, various transformations along one of the axes are possible. In the first case, we start with specialisation of the concept towards the person domain of university positions. For this new specialisation (e.g. PersonAtUniversity), new detail is concretised-attribute named position values of which are constrained according to the problem domain (e.g. Student, Professor, etc.). Again, many possible transformations arrive in this state of the process, what gives us responsibility to choose among (e.g. specialisation to AdultPersonAtUniversity, or concretize by adding new attribute named age and constrain it in a later transformation). Similar approach, different in the order of applied transformations, is shown as the lower path in our example. Although both paths end in the same state in our example, it shall not be taken as a general rule. The order and dedicated priority of transformations may lead to very different representations (e.g. concretizing to different programming languages). Although being in different states, the developed systems might fulfill the requirements criteria and both pass verification and validation tests.

Abstraction and Generality in Recent Works: Abstraction plays a significant role in the field of reverse engineering. In Egyed [11], abstraction transformations are used for reducing complexity of class models of large systems. The subject entity for transformation is the whole graph representation of the class model of reverse engineered system. Abstraction transformation is represented as a transformation rule between two sub-graphs. In the class model graph, abstraction patterns are recognized and transformed to a more abstract subgraph representation. A tool was developed to help manage and apply the abstraction transformations and was successfully tested on real software systems.

One of the common consequences of using general concepts in development is the risk of reduced performance. In Schultz *et al*. [12], clear representation of specialisation transformation for selected design patterns is proposed. It is captured in a specialisation pattern. Specialisation pattern holds information about possible specialisation transformation of the program code in the context of application of the pattern.

Specialised program code executes with significant performance improvement. Because of such automated specialisation might lead to code/complexity explosion, the process should be overlooked by a human developer-driven.

One of the characteristics of a programming paradigm may be the central abstraction it deals with [13]. In a simplified view, multi-paradigm approaches to software development offer various combinations of these verified abstraction concepts. Trying to solve these constrains often leads to better representation of used techniques (e.g. the aspect as a separation and an in-one-place representation of crosscutting concepts, previously solved for example by macro and precompiler techniques). The common concept that remains behind is the abstraction itself. In another point of view, the abstraction concepts differ according to level of granularity and the domain/concern they are applied in. For example, Vranic [14] presents a feature model of concepts in the solution domain of AspectJ paradigm. Such a model represents the set of the higher-level abstractions, along with their variation-points. Further classification according to the discrimination of the abstraction and the generality transformations would probably lead to a more clear and exhaustive understanding of these concepts and will be subject of our further work.

CONCLUSION

The application of abstraction and generalisation transformations is undoubtedly one of the fundamental mechanisms in the field of software development. We proposed representation that is more formal with respect to abstraction and generalisation operators. Based on our simple examples, we discussed the possibilities and consequences of these transformations. For the purpose of a model space for transformations, we used two-dimensional space for abstraction and generalization. Application of an operator moves an entity along one of the axes, mainly towards the concretisation and specialisation. The ambition of such representation is not to define the exact order of transformations that shall be taken, but to help in a clearer representation and separation of taken steps.

Of course, it is difficult if not even impossible to define the exact order for the application of the transformations or the path that has to be traveled in the two-dimensional space for even the simplest developed systems. However, the overall direction and some prediction might be captured to successive level in some sort of development knowledge. The knowledge may be helpful in decision-making based on (not exhaustive list consisting of) the requirements, the problem domain, solution domain mechanisms to be used, development methodology, etc. Although the sequence of modeling, design and development decisions may lead to different paths and destinations in the two-dimensional space, the results in the form of developed systems might still (and of course, should) fulfill the requirements and successively model the problem domain. Our future work is concerned with a definition that is more exact and elaboration of the notion of abstraction and generality in the field of software development. We would like to define a set of transformations, that would form a knowledge base for designing web applications for various problem domains. Based on the specific experience, we would like to progress in building more formal and clear knowledge for the broad field of software development.

ACKNOWLEDGEMENTS

The work reported here was partially supported by Slovak Scientific Agency, project No. VG 1/0162/03.

REFERENCES

1. Czarnecki, K. and U.W. Eisenecker, 2000. Generative Programming. Methods, Tools and Application. Addison Wesley.
2. Buschmann, F., R. Menier, H. Rohnert, P. Sommerland and M. Stal, 1996. Pattern-Oriented Software Architecture, A System of Patterns. John Wiley.
3. Cai, J., R. Kapila and G. Pal, 2000. HMVC: The layered pattern for developing strong client tiers, in JavaWorld (http://www.javaworld.com).
4. Krueger, Ch. W., 1992. Software Reuse. ACM Computing Surveys, 24: 131-83.
5. Pree, W., 1994. Design Patterns for Object-Oriented Software Development. Addison-Wesley.
6. Návrat, P., 1994. Hierarchies of programming concepts. Abstraction, generality and beyond. ACM SIGCSE Bulletin, 26: 17-21, 28.
7. Návrat, P., 1996. A Closer Look at Programming Expertise. Critical Survey of Some Methodological Issues. Information and Software Technol., 1: 37-46.
8. Smolárová, M., P. Návrat and M. Bieliková, 1998. Abstracting and Generalising with Design Patterns. In: 13th Int. Symp. Computer and Information Sci., Oct. 26-28, Belek-Antalya, Turkey, pp: 551-558.
9. Smolárová, M. and P. Návrat, 2000. Reuse with Design Patterns: Towards Pattern-Based Design. Proc. of Conference on Software: Theory and Practice. 16th World Computer Cogress, 21.15.8.2000 Beijing, China. Eds. Z. Feng-D. Notkin-M.C. Gaudel, Publ.House of Electr. Industry, pp: 232-235.
10. Návrat, P. and M. Smolárová, 2000. Pattern-Supported Software Development. Role of Abstraction and Generality. Technical report, STU Bratislava.
11. Egyed, A., 2002. Automated Abstraction of Class Diagrams. ACM Transactions on Software Engineering and Methodology, 11: 449-491
12. Schultz, U.P., J.L. Lawall and Ch. Consel, 2000. Specialisation Patterns, Proceedings of ASE.
13. Vranic, V., 2002. Towards Multi-Paradigm Software Development. J. Computing and Information Technol.-CIT 10: 133-147.
14. Vranic, V., 2001. AspectJ Paradigm Model: A Basis for Multi-Paradigm Design for AspectJ. In Proceedings of Generative and Component-Based Software Engineering, GCSE (Ed. Jan Bosch), Springer Verlag