

Participatory Programming: Developing Programmable Bioinformatics Tools for End-Users

Catherine Letondal

▶ **To cite this version:**

Catherine Letondal. Participatory Programming: Developing Programmable Bioinformatics Tools for End-Users. Henry Lieberman; Fabio Paternò; Volker Wulf. End User Development, 9, Springer, pp.207-242, 2006, 978-1-4020-4220-1. 10.1007/1-4020-5386-X_10 . hal-01286956v2

HAL Id: hal-01286956

https://enac.hal.science/hal-01286956v2

Submitted on 6 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License

CATHERINE LETONDAL

PARTICIPATORY PROGRAMMING:

DEVELOPING PROGRAMMABLE BIOINFORMATICS TOOLS FOR END-USERS

Abstract

We describe participatory programming as a process that spans design, programming, use and tailoring of software. This process, that includes end-users at each stage, integrates participatory design and programmability. Programmability, as a property that relies on a reflective architecture, aims to let the end-users evolve the tools themselves according to their current, specific needs and to let them control better the way results are computed. We present an environment that results from this approach, called *biok*, developed for researchers in biology, which is both domain-oriented and open to full programming.

1 Introduction

This chapter describes what we call 'Participatory Programming', or how to integrate participatory design and programmability. We consider programming, not as a goal in itself, but rather as a potential feature, available if things go wrong in the context of use. We discuss how to better integrate the context of the user in the programming activity by both: a) letting the user participate to the design of the tools and b) providing access to programming via the user interface and from visible objects of interest, within a scaffolded software architecture. This approach applies to fields where users are both experts in their domain and able to develop basic programming skills to enhance their work.

Biology has seen a tremendous increase in the need for computing in recent years. Although biology labs may employ professional programmers and numerous ready-made tools are available, these are rarely sufficient to accommodate this fast-moving domain. Individual biologists must cope with increasing quantities of data, new algorithms and changing hypotheses. They have diverse, specialized computing needs which are strongly affected by their local work settings. This argues strongly for a better form of end-user development.

The problem is how best to provide access to programming for non-professional programmers. Can we determine, in advance, what kind of end-user development is required or how the software might evolve? Must we limit end-user development to specific well-defined features? Do end-users actually want to develop their own tools?

Our approach involves cooperative software development and co-evolution in two complementary ways: interviews and workshops with biologists to define their environments for data analysis, and software flexibility or *programmability*. This term refers to two main dimensions: (a) to let the end-users *evolve* these tools themselves according to their current specific needs; (b) to let the user better *control* the way results are computed.

In this chapter, we first describe some important characteristics of software development and evolution in biology, as well as situations where biologists who are not professional programmers may need to change the software they use. Next, we introduce our approach to help biologists better control their tools, the idea of *participatory programming*, and we provide a description of the participatory design process. We describe our prototype *biok* in section 4, followed by a section (5) that recounts uses of this prototype. The final section (6) provides a discussion of our choices, where we address the general aspects of software flexibility and open systems with respect to End-User Development.

2 Problem Description

Having installed scientific software for 8 years at the Institut Pasteur and having taught biologists how to use these scientific tools, I have observed that, in the past decade, the development of computing tools for biology and genomics has increased at a fast pace to deal with huge genomic data and the need of algorithms to discover their meaning. Daily work has also changed for the standard biologist: using computer systems to perform their biological analyses is hardly possible without some basic programming [Tis01]. Indeed, although there are already many ready-to-use tools, including Web-based tools and software packages for the micro-computer this is not really sufficient, even for usual tasks.

In order to justify our objectives and our approach, we need to describe the context of this work. In the following sections we describe the typical problems that have to be solved, the general idea of programming in scientific research and the more general issue of dealing with scientists as end-users. We have also performed several kinds of user studies that we describe in section 3.7.

2.1 Use of Computing at Institut Pasteur

We conducted a campus-wide survey in 1996, which consisted of 40 questions grouped in categories, regarding computing education, software and network use, access to technical documentation and types of technical problems encountered [Let99b]. Figure 1 shows the main groups that we identified through the analysis of the survey data (about 600 answers) plotted on two dimensions: level of programming autonomy and level of use of scientific computing:

- *Occasional users* were the largest group (36%) and had no direct use of scientific computer tools.
- *Non-Unix users* (15%) did not use the IT department training and technical

support, they had their own PC and mostly ran statistical software.

- *Young scientists* (15%) were interested in bioinformatics, and were able to program or at least build Web sites. They could read software documentation and were able to teach themselves.
- *Learners* (15%) were more-established scientists who had recently taken a course to improve their know-how of scientific software. This training was often conducted by the IT department.
- *Gurus* (6%) were heavily involved in computing and programming scientific software. They often acted as local consultants or *gardeners* [GN92].

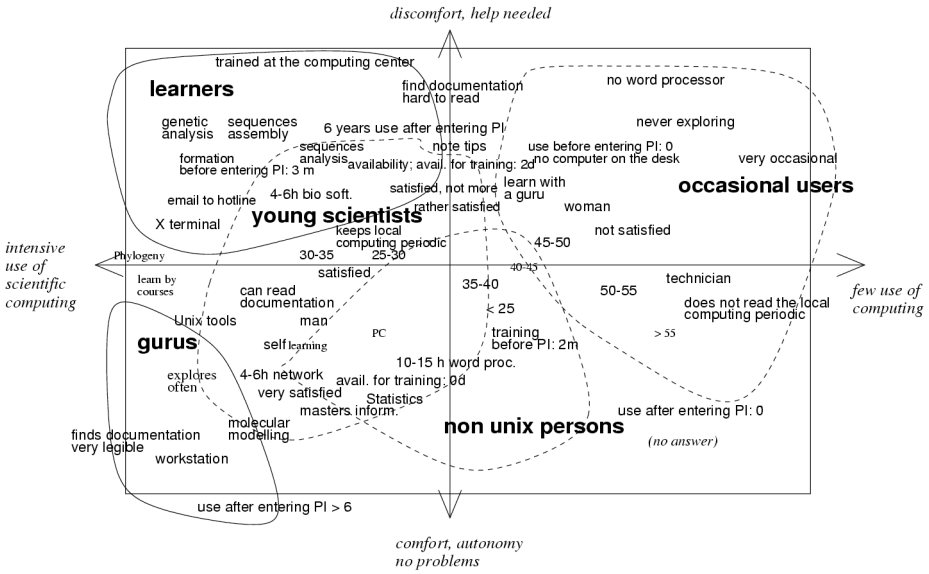


Figure 1: Campus-wide survey analysis of the use of computing

Both the computing skills and the available computer tools have changed greatly in the intervening years since this survey was taken. The Institut Pasteur has hired a significant number of bioinformaticians to act as local developers [GN92]. In various laboratories, young scientists (Ph.D.s and post-doctoral fellows) are now more likely to have had formal training in computing and the number of convenient software packages for biologists has increased, particularly via the Internet.

2.2 Typical Problems

In order to illustrate the need for end-user development in the context of biology and bioinformatics, let us show some typical examples (see also [CFL03]). Below is a

list of real programming situation examples, drawn from interviews with biologists, news forum, or technical desk. These situations happened when working with molecular sequences, i.e. either DNA or protein sequences (a sequence is a molecule that is very often represented by a character string, composed of either DNA letters – A, C, T and G – or amino-acid letters – 20 letters).

- scripting: search for a sequence pattern, *then* retrieve all the corresponding secondary structures in a database
- parsing: search for the best match in a database similarity search report *but relative to each subsection*
- formatting: renumber the positions of a sequence from -3000 to +500 instead of 0 to 3500
- variation: search for patterns in a sequence, *except repeated ones*
- finer control on the computation: control of the order in which multiple sequences are compared and aligned
- simple operations: search in a DNA sequence for the characters other than A, C, T and G

As illustrated by these examples, unexpected problems may arise at any time. However, these scenarios involve rather simple programmatic manipulations, without any algorithmic difficulty or complex design. An important reason why programming is needed here is that the function, although easy to program, or even already implemented somewhere inside the program, has not been *explicitly* featured in the user interface of the tool.

2.3 Programming in scientific research

Apart from these scenarios showing that everyday work leads to operations that involve some programming, there are fundamental reasons why scientific users, or at least a part of them, would need to program.

- *Sharing expertise*. Biologists, having accumulated a lot of knowledge through their academic and professional experience, in such a fast evolving area, are more able to know what kind of information is involved in solving scientific problems by a computational model. In her course on algorithmics for biologists [Sch03] [LS02], Schuerer explains that sharing expertise requires some computing skills on the side of biologists, in order for them to be aware of the tacit hypotheses that are sometimes hidden in computer abstractions.
- *Programs as evolving artifacts*. A computer program *evolves*, not only for maintenance reasons, but also as a refutable and empirical theory [Mor97]: thus, being able to modify the underlying algorithm to adapt a method to emerging facts or ideas could be, if not easily feasible, *at least anticipated* [LZ03].
- *Expression medium*. The field of bioinformatics and genomics is mostly

composed of tasks that are defined by computer artifacts. In these fields the expression medium [DiS99] for problem solving is encoded as strings, and problems are expressed as string operations (comparisons, counting, word search, etc...).

3 Approach: Participatory Programming

What kind of solutions could help biologists to get their work done?

3.1 More tools

One possibility is that biologists simply need more tools, with more parameters and more graphics. *This is maybe true*, but:

- Some features or needs, particularly in a fast evolving research field, where the researcher must be inventive, cannot be anticipated.
- Such software is complex: users must master many different tools, with specific syntax, behavior, constraints and underlying assumptions; furthermore, these tools must be combined, with parsers and format converters to handle heterogeneous data.

3.2 A programmer at hand

A second possibility could be for biologists to have a programmer at hand whenever they need to build or modify the programs. There are indeed many laboratories where one or more programmers are hired to perform programming tasks. This is however clearly not feasible for every biologist (for instance, the Institut Pasteur laboratories have about a dozen such local programmers, for more than 1500 biologists).

3.3 Programming

A third possibility involves programming: biologists just have to learn some basic programming skills, since programming is the most general solution to deal with unforeseen computational needs. In fact, many biologists program, and even release software. Most of the programs for data analysis are in fact programmed by biologists. We see two clear types of development:

- large-scale projects such as [SBB02], developments in important bioinformatics centers such as the US National Center for Biotechnology Information (NCBI) or the European Bioinformatics Institute (EBI), or research in algorithmics by researchers in computer science;
- local developments by biologists who have learned some programming but who are not professional developers, either to deal with everyday tasks for managing data and analysis results, or to model and test scientific ideas.

The two lines often merge, since biologists also contribute to open-source

projects and distribute the software they have programmed for their own research in public repositories.

However, as programming is also known to be difficult, not every biologist wants to become a programmer. Most of the time, this change implies a total switch from the bench to the computer.

3.4 Programming *With* the User Interface

An intermediate step is End-User Programming (EUP) [Eis97], which gives biologists access to programming with a familiar language, i.e., the language of the user interface. Programming by demonstration (PBD) [Cyp93] [Lie00] lets the user program by using known functions of the tool: with some help from the system, the user can register procedures, automate repetitive tasks, or express specific models (styles and formats for word processors, patterns for visualization and discovery tools, ...). Visual programming languages, in contrast, offer a visual syntax for established programming concepts: programming with the user interface ideally means programming at the task level, which is more familiar to the end-user [Nar93].

Customization is related, but has a different goal: EUP provides tools or languages for *building* new artifacts, whereas customization enables to *change* the tool itself, usually from among a set of predefined choices or a composition of existing elements. Although these two different goals might be accomplished with similar techniques, this means that the level of the language is a base level in the case of EUP, whereas it is a meta level in the case of customization.

3.5 Programming *In* The User Interface

We introduce Programming *In* the User Interface as an approach that provides the end-user with a scaffolded access to general programming at use-time. We thus distinguish Programming *With* the User Interface from Programming *In* the User Interface. These approaches differ by essentially two aspects: 1) in our approach, programming is made available to the end-user, but the programming language is not necessarily the user interface language; 2) in this approach, programming includes customization, i.e modifying some parts of the software being used.

Related work also includes tailoring approaches which enable the user to change the software at use-time [FO02] [HK91] [MCLM90] [Mor97]. Similar approaches also include programmable tools where the user can add functionalities to the tool by accessing to an embedded programming language and environment [Eis95] [Mor97] [SU95]. Research such as [DE95] [WG01] focusing on methods to encourage the user to tailor the software by lowering a technical, cognitive or sociological barrier are very relevant as well, as explained in section 3.7.3.

Fischer's concept of MetaDesign [FS00] attempts to empower users by enabling them to act as designers at use-time. In this approach, software provides a domain-oriented language to the users and lets them re-design and adapt current features to

their own need. As explained in [FO02], user software artifacts can then re-seed the original design in a participatory way. Our approach is very similar: as described in section 3.7.2, we first let users participate to the initial design by conducting workshops and various user studies. Then, we either take their programming artifacts as input to prototyping workshops or, as described in section 5, we put their modifications in *biok* back into the original tool. The main difference in our approach lies in the programming language that is provided to the user. We chose a standard programming language, that the user can re-use in other contexts, like Eisenberg [Eis95] who offers the concept of programmable applications in which users of a drawing tool can add features by programming in Scheme. Thus, our tool does not include an EUP language: we indeed observed that using a standard general-purpose programming language is not the main obstacle in the actual access to programming, in the context of bioinformatic analyses (see sections 3.6 and 5).

Some approaches offer the technical possibility for the user to change the application during use by having access to the underlying programming language. MacLean et al [MCLM90] describe how to build a whole application by combining, cloning, and editing small components (buttons), associated to simple individual actions. This seminal work has greatly inspired our work, where graphical programmable objects (see section 4.2) form the basis of an application, and are a technical extension of buttons to more general programmable graphical objects. In this regard, our technical environment is closer to Self [SU95] or Boxer [DA89], except that we needed to use an open environment and a scripting language featured with extensive graphical and network libraries. As in Morch [Mor97], graphical objects provide an architecture where a mapping is provided between application units and programming units in order for the user to easily locate programming chunks of interest. As in our approach, the user interface helps the user with access to programming, *only when needed*. Most of the time, the focus of the user is the analysis of his or her data. However, as described in sections 3.7.2 and 3.7.3, our approach is not only technical, it relies on a participative approach at design-time, which helps determine how to build such an environment.

The following sections discuss one of the main aspect of our approach, which is to provide full access to programming (section 3.6), and explain how, by using contextual and participatory design methods, we address potential issues that could be raised by this access (section 3.7). Section 4 describes *biok*, our prototype tool for participatory programming, which, according to this approach, is *both* an analysis tool and a programming environment. This leads to an environment that is both highly domain-oriented and highly generic and open to full programming.

3.6 The problem of Programming

We decided to provide an access to a general programming language (see 4 for a short description of this language), as explained in the previous section, as discussed by [Eis95], and as opposed to EUP approaches. Let us discuss the choices we made:

- is programming really *too* difficult for end-users?

- is programming the *main* difficulty for end-users?
- is programming the problem at all?

Our thesis is that, focusing on the design of an end-user programming language and stressing programming difficulties, we do not progress toward a general solution regarding end-user development in biology or similar fields.

3.6.1 Difficulties of Programming

Programming is indeed technically difficult and raises cognitive issues, but this is not the main reason for biologists not to program when they need it. Nardi [Nar93] has shown that having to write textual commands, one of the most "visible" and discriminating aspect of classical programming, is not really the main explanation for the technical barrier: for instance, users are able to enter formula in a spreadsheet, for example, or to copy and modify HTML pages. If the language is centered on the task to perform, the user will be able to learn and use it.

We have also been running an annual intensive four-month course for research biologists to teach them various aspects of computing [LS02]. During this course, computer scientists and bioinformaticians from the IT department, as well as visiting professors, cover programming techniques, theoretical aspects (such as algorithm development, logic, problem modeling and design methods), and technical applications (databases and Web technologies) that are relevant for biologists. According to our experience during this course, reducing the difficulty of programming to difficulties with algorithms is too simple. The first reason is that there is not much algorithmic complexity in their everyday programming. The second reason is that, whereas biology students had good aptitude for programming (they had to program in Scheme, Java, perl or Python), and enough abstract reasoning for the required programming tasks, a significant part of them did not actually program after the course, even though they would need it. Why is that? This issue formed a basis for our reflection on both the technical and organizational context of the programming activity of biologists, that is illustrated by a case study described in section 5.2.

Software engineering aspects is a more significant barrier. The occasional user faces more problems with programming-in-the-large than with syntax or abstraction. The tools that are actually used for bioinfomatics analyses are often complex and large systems, rather than small software packages. Users can not build such systems by themselves. Can they at least participate in those parts that depend on their expertise? Finally, biologists want to do biology, not computer science. Even if they can program, and could overcome specific technical problems, they prefer to spend their time on biology. Therefore, both the technical context (software being used) and the use context (data analyses) should be taken into account when designing programming tools for such users.

3.6.2 What is Programming?

We believe that seeking for the perfect programming language for end-users is both

too simplistic and illusory. When I say to a colleague that "I am programming", he or she knows what I mean. This however does not lead to a definition of programming. There are indeed various and different definitions of programming : design of algorithms, automation, building of software, abstraction [Bla02], delegation [Rep93], design of the static representation of a dynamic process [LF95], problem definition, [Rep93][Nar93]. Thus, programming, being a polysemic term, that is not precisely defined, seems quite inappropriate for a specification to develop an end-user programming system [Let99a] [Let99c]. Even though programming claims to be a general solution and a general tool, it is also rather difficult to define programming *activity* without taking the sociological and professional context of this activity into account. A student learning programming to prepare for an exam and to enhance his or her reasoning capabilities is not pursuing the same objective as a software engineer building an application for a customer, and, more generally, one does not program the same way when programming for oneself than when programming for other people.

Furthermore, the definition of *what programming is* might benefit from the definition of *what programming is not*.

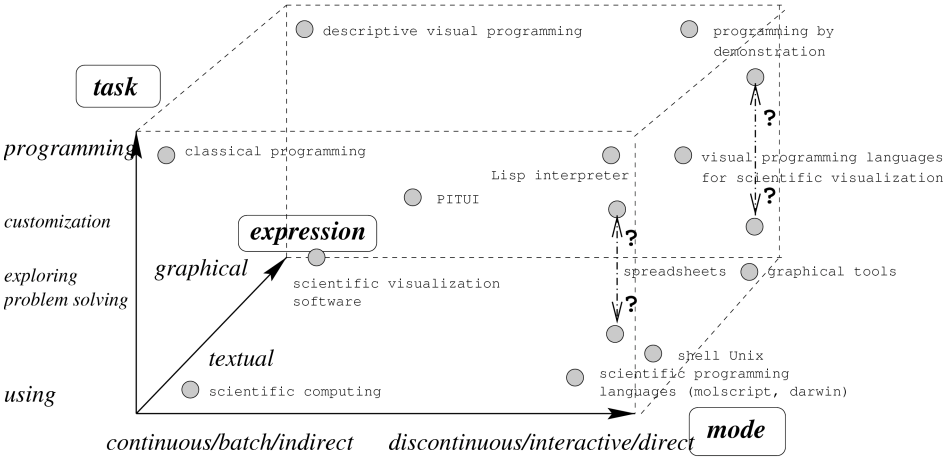


Figure 2: Dimensions to contrast programming and non-programming.

Figure 2 shows various concepts related to programming or non-programming opposed along 3 axes:

1. *the mode* (x axis): from batch, indirect and continuous to interactive, direct and discontinuous,
2. *the underlying task* (y axis): from using to programming,
3. *the form of the expression* (z axis): from textual to graphical.

This diagram is inspired by the classification given in [BHP94], where visualization software is classified along three axes: mode, expression and the

compatibility of the system with legacy software. The expression (z axis) and mode (x axis) axes have been thoroughly studied, and it is not our purpose here to study them further. For instance, the expression (z) axis describes the difference between *programming in the C language* and *programming in the Labview visual environment* [lab87]. But it also describes the difference between reading a textual program result, such as searching for similarities in sequences databases, visualizing hits in a 3D graphical plot. The mode axis (x axis) describes the difference between programming with a compiler and programming within an interactive interpreter. This axis also describes the difference between using an interactive tool and running long batch analyses that read their input at the beginning of the computation and produce their output at the end [BHP94].

We can observe from this diagram that, even though we build it on dimensions that, taken separately, contrast programming to non-programming, clearly-identified programming activities often belong to the non-programming side of each: while programming is often opposed to interaction, learning to program with a Lisp interpreter is on the *interactive* end of the x axis; building a Petri Net or programming in Labview belong to the graphical end of the z axis, and writing a script to build a visual model of a molecule [PL92] is on the *use* end of the y axis, since the goal is to understand and analyze a molecule, not to program. In fact, within these combined dimensions, programming activities fit within a continuum, which makes it difficult to rely only on a definition of programming to build an end-user programming environment.

In our diagram, we stress the importance of the *context of programming* as determined by the user's activity and goals: we use a task axis (y axis) instead of the compatibility axis from [BHP94], to describe the difference between *programming, building* a tool, and *using* it. This axis and the issues of why and when biologists need to program is the topic of the following section.

3.7 Studying the Context of Programming

Having explained in the previous section why the *context* of programming should be taken into account more deeply than a definition of programming, we describe in this section the studies and participatory activities that have been organized to understand this context [LM04].

3.7.1 Interviews

Among a total of 65 interviews that were conducted in the context of various projects over the past seven years, about 30 were dedicated to end-user programming. They were mainly intended to collect use scenarios, or to observe biologists using scientific software. Interviews were generally informal and open: we often just asked the biologists to act in front of us a scenario of a recent bioinformatic analysis. Some of the interviews have been videotaped or recorded, and annotated.

Several of these interviews enabled us to observe biologists programming, either

by using standard programming environments and languages such as C or C++, or, very often, scripting languages such as awk to parse large text files, perl to write simple Web applications, or Python to build scripts for analyzing structural properties of proteins. We also observed uses of visual programming environments such as HyperCard or even visual programming languages. Khoros [RASW90] for image analysis or Labview [lab87], for instance, are used in some laboratories, mostly due to their good libraries for driving hardware devices, and image or signal processing routines. We also observed various people using spreadsheets for performing simple sequence analyses.

During these interviews, we made several observations:

- *Re-use of knowledge*. Most of the time, biologists prefer using a technique or a language that they already know, rather than a language that is more appropriate for the task at hand, which is referred to as the assimilation bias by [CR87]. A researcher had learnt HyperCard to make games for his children, and used it in the laboratory for data analysis, even though nobody else knew it and thus was able to provide help. But the result was efficient and he almost never had to ask to the IT Center for help to find or install simple tools. Another researcher wrote small scripts in the only scripting language she knew: awk, although perl that is now often used and taught in biology would have been much more efficient. In summary, as long as the result is obtained, it does not matter how you get it. Similarly, a researcher tends to use a spreadsheet instead of learning to write simple scripts that would be be more suitable to the task.
- *Opportunistic behavior*. Generally and as described in [Mac91b], biologists, even if they can program, will not do so, unless they feel that the result will be obtained really much faster by programming. If this is not the case, they prefer to switch to a non-programming methods, such as doing a repetitive task within a word processor or performing an experiment at the bench. There is no requirement nor any scientific reward for writing programs. They are only used as a means to an end, building hypotheses.
- *Simple programming problems*. During his or her everyday work, a biologist may encounter various situations where some programming is needed, such as simple formatting or scripting (for extracting gene names from the result of an analysis program and use them for a subsequent database search) and parsing, or simple operations, not provided in the user interface, such a searching for characters other than A, C, G or T in a DNA sequence.
- *Need for modifying tools rather than building from scratch*. A frequent need for programming that we observed is to make a variant or add a function to an existing tool. Designing variants for standard published bench protocols is often needed in a biology laboratory. For instance, when constructing a primer for hybridization [1], it is often needed to adapt the number of

[1]A primer is a short DNA sequence used to generate the complementary DNA of a given sequence

washings according to the required length and composition of the primer, or to the product that is used. With software tools, this is however unfortunately seldom feasible, but it would be highly valuable since there are already many existing tools that perform helpful tasks, and biologists rarely want to build a tool from scratch.

- *Exploratory use of tools*. There is a plethora of tools, including new tools, for the everyday task of biologists, and these tools are often specialized for a specific type of data. This leads to a very interactive and exploratory use of computing tools [OAKB01]. For instance, an observed scenario started by the search of a protein pattern published in a recent paper. The user was looking for other proteins than those referred to in this paper and that also contained this motif. After an unsuccessful attempt - the results were too numerous for an interactive analysis - the researcher decided to use another program. This attempt failed again because his pattern was too short for the setting of this specific program. He then decided to extend it by adding another one, also belonging to the set of proteins mentioned in the paper. In the end, this enabled a final iterative analysis of each result. This is a brief summary that stands for many scenarios we have observed, often resulting in many failures due to a problem with the program, or with the format of the data.

  This typical behavior might both be a barrier to and a reason for programming. It can be a barrier by preventing a user to think of a more efficient way to get a result (leading to an "active" user behavior as described by [CR87]). However, at the same time, it can be a ground for programming since programming could help to rationalize, *a posteriori*, such an exploratory behavior. This, however, involves some kind of anticipation: for instance, it might be a good place for programming instruments such as history and macro recording.

3.7.2 Workshops

biok, that we describe in section 4, has involved a series of video brainstorming and prototyping workshops over several years from 1996 to 2004. We drew prototyping themes from brainstorming sessions (Figure 3) and from use scenarios, which based on interviews and observation. Each workshop involved from 5 to 30 people, with participants from the Institut Pasteur or other biological research laboratories, as well as biology researchers who were students in our programming course.

Finding potential dimensions for evolution

From the very beginning of the design process, it is important to consider the potential dimensions along which features may evolve. Interviews with users help inform concrete use scenarios, whereas brainstorming and future workshops create a design space within which design options can be explore. As Trigg [Tri92], Kjaer [KM95], [SKW97] or [Kah96] suggest, participatory design helps identify which

areas in a system are stable and which are suitable for variation. Stable parts require functionality to be available directly, without any programming, whereas variable parts must be subject to tailoring.

For example, the visual alignment tool in *biok* vertically displays corresponding letters in multiple related sequences (Figure 6, back window). Initial observations of biologists using this type of tool [Let01b] revealed that they were rarely flexible enough: biologists preferred spreadsheets or text editors to manually adjust alignments, add styles and highlight specific parts. It became clear that this functionality was an area requiring explicit tailoring support.

Design of meta-techniques

Scenarios and workshops are important to effectively design meta-level features. Scenarios sometimes reveal programming areas as side issues. The goal is not to describe the programming activity per se, but rather to create an analogy between the task, how to perform it, and the relevant programming techniques. We identified several types of end-user programming scenarios:

- *Programming with examples*: One workshop participant suggested that the system learn new tags from examples (tags are visualization functions, see 4.4). Another proposed a system that infers regular expressions from a set of DNA sequences. These led to a design similar to SWYN [Bla00].
- *Scripting*: one participant explained that text annotations, associated with data, can act as a "to do" list, which can be managed with small scripts associated with the data.
- *Command history*: a brainstorming session focusing on data versioning suggested the complementary idea of command history.

The *biok* tag editor design (Figure 6, front window) had to consider the following issues: Must programming be available in a special editor? Must it require a simplified programming interface? Should the user interface be interactive? Should it be accessible via graphical user interface menus?

We found prototyping workshops invaluable for addressing such issues: they help explore which interaction techniques best trigger programming actions and determine the level of complexity of a programming tool. For example, one group in an alignment sequence workshop built a pattern-search mockup including syntax for constraints and errors (Figure 3).

One of the participatory design workshops was organized in the Winter of 2001 with five biologists to work on the *biok* prototype. Among the participants, four had followed a programming course, and programmed from time to time, but not in Tcl, except one who had programmed in Visual Tcl. Before the workshop, interviews had been conducted with discussions about the prototype, and participants were sent a small Tcl tutorial by email. The aim of the workshop was to experiment the prototype and get familiar with it through a scenario (instructions were provided on a Web page). They had to play with graphical objects, and define a simple tag. The issues that would arise during this part would then be discussed and re-prototyped

during a second part. The scenario had also spontaneously been "tested" by one of the participants who brought some feedback about it. Although the workshop was not directly aimed at properly testing the prototype, the participants behaved as if it was, and this actually brought some insights on the design of the prototype - briefly and among the most important ones:

- The participants were somewhat disturbed by a too large number of programming areas: formula box, shell, method editor, ...
- They had trouble to understand, at a first sight, the various elements of the user interface and how they interact.
- They had the feeling that understanding the underlying model would help.

One of the the tasks of the scenario was to define a tag. The only tool that the participants had for this was an enhanced text editor, only providing templates and interactive choosers for the graphical attributes. This tool proved completely unusable and participants got lost. The tool was indeed too programmer-centered, and difficult to use, and there was no unified view of the tag definition. This led to another workshop shortly after this one, and after a long brainstorming session, one participant built a paper-and-transparencies prototype. We created an A3-size storyboard mockup and walked through the tag creation scenario with the biologists. The tag editor currently implemented in *biok* is a direct result of these workshops (see section 4.4).

Participatory approaches are also helpful when designing language syntax [PRM01, dCdS03] or deciding on the *granularity* of code modification. As observed during the previously described workshop, the object-oriented architecture and the method definition task apparently did not disturb users that much. In a previous workshop that we started by displaying a video prototype showing the main features of *biok*, participants tended to adopt the words "object" and "method" that were used in the video. Interestingly, one of them used the term: "frame" all along the workshop in place of object, probably because objects in *biok* (and in the video prototype) are most often represented by graphical windows. In object-oriented programming terms, we found however that biologists are more likely to need new methods than new classes. Since defining new classes is a skilled modeling activity, we designed *biok* so that user modifications at the user level do not require sub-classing. User-edited methods are performed within the current method's class (see section 4.5) and are saved in directories that are loaded after the system. However, visualizing tags required the user to create new classes, which lead us to provide this as a mechanism in the user interface.



Figure 3: Prototyping a pattern-search and an annotation scenario.

Setting a design context for tailoring situations

Our observations of biologists showed that most programming situations correspond with breakdowns: particular events cause users to reflect on their activities and trigger a switch to programming [Mac91a]. Programming is not the focus of interest, but rather a means of fixing a problem. It is a distant, reflexive and detached "mode", as described in [Win95], [SUC92] or [Gre93]. While end-user programming tools may seek to blur the border between use and programming [DLL03], it is important to take this disruptive aspect into account, by enabling programming *in context*. Developing and playing through scenarios are particularly useful for identifying breakdowns and visualizing how users would like to work around them.

3.7.3 Participatory Design and Participatory Programming

Participatory programming integrates participatory design and end-user programming. Participatory design is used for the design of an end-user programmable tool, yet biologists programming artifacts also participate in the making of tools. These artifacts are either produced by local developers, observed during interviews, or they can be produced by end-users using *biok* (section 5).

Henderson and Kyng [HK91] and Fischer [Fis03] also discuss how to extend end-user participation in design to use-time. Our approach is very similar except that it includes programming participation and not only domain-level design artifact. Likewise, Stiemerling et al [SKW97] or [Kah96] provide a detailed description illustrated with various examples of a participative development process for designing tailorable applications. Yet, in both cases, the tools that offer document search and access right management features, do not include programming.

4 Biok: Biological Interactive Object Kit

biok is a prototype of a programmable graphical application for biologists written in XOtcl [NZ00] and the Tk graphical toolkit [Ous98]. XOtcl is an object extension of Tcl, inspired by Otcl [WL95], a dynamic object-oriented language. Tcl is not an end-user programming language: on the contrary, it is a general purpose language, and moreover, it is not simple. Like [WP02], our experience teaching programming to biologists, show that languages such as Python are much easier to learn. However, we chose Tcl XOtcl for:

1. its incremental programming feature, i.e. the possibility to define or redefine methods for a class at any time, even when instances have been created, which is very convenient for enabling programming in the user interface,
2. its introspection tools that are generally available in Tcl software components and that are mandatory to get a reflective system (see 6);

3. its dynamic tools such as filters and mixins, that enabled us to build a debugging environment and some algorithmic flexibility features [LZ03].

The purpose of *biok* is twofold: to analyze biological data such as DNA, protein sequences or multiple alignments, and to support tailorability and extensions by the end user through an integrated programming environment.

4.1 Biological data analyses

The first purpose of biok is to provide an environment for biologists to analyze biological data. Currently, it includes a tool to compare and manually align several related sequences or display alignments that are computed by one of the numerous alignment programs available [Let01a]. A molecular 3D viewer is also provided [TNQL03] (Figure 4). Thanks to this toolkit, biologists can compare similar objects in various representations, simultaneously highlighting specific features in the data: the alignment tool stresses common features among homologous sequences, whereas the 3D viewer shows their structural location, which provides useful hints regarding the potential function in the cell.

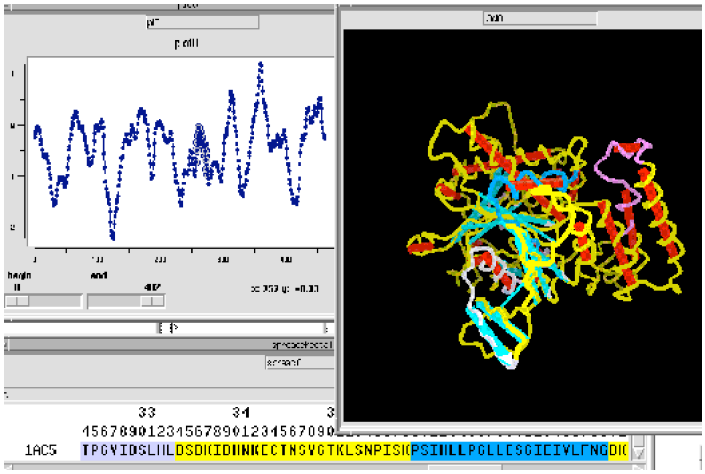


Figure 4: Analyzing biological data: a plot displays the hydrophobicity curve of a protein sequence, that is simultaneously displayed in a 3D viewer. Structural properties of the protein, namely transmembrane segments, are simultaneously highlighted in the two protein representations. The user, by selecting parts of the curve, can check to which hydrophobicity peaks these segments might correspond.

4.2 Graphical programmable objects

The main unit both for using *and programming* in *biok* is what we call a "graphical object". Objects are indeed "visible" through a window having a name, or alias, that the user may use as a global Tcl variable (Figure 4 show three such graphical objects). Graphical objects are programmable in several senses:

1. their "content" may be defined by a *formula*,
2. their *methods* may be edited, modified, copied to define new methods
3. their graphical components are accessible as *program objects*
4. graphical attributes may be defined to implement *visualization functions*

Graphical objects also have a command "shell", to either directly call the object's methods or to configure the Tk widget via a special "%w" command, as illustrated in Figure 5 where the plot's title and curve legend have been directly set up by Tk commands. Commands are stored in a class-specific editable history file as soon as they have been issued, so they can be instantaneously reused on another object of the same class.

4.3 Dataflow and Spreadsheet models

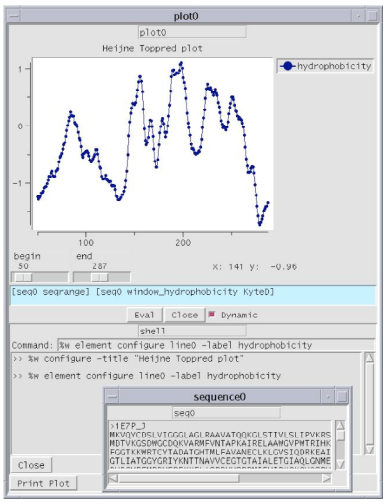


Figure 5: Graphical objects, formulas and shell.

Given the wide use of spreadsheets and the pervasiness of its related dataflow computing paradigm, we have designed the content of graphical objects as being optionally defined by a formula involving one or more source objects. Figure 5 shows two graphical objects: one (front window) is a "Sequence" object, called "seq0", containing the letters representing the amino-acids of a protein. The other is a "Plot" object ("plot0"). Its formula returns two lists of floating point numbers: the first one represents the ticks of the x axis and the second the values of the curve, that displays the hydrophobicity peaks of the "seq0" object. Each time the sequence is modified, the curve is redisplayed, unless the "Dynamic" switch has been turned off. The "Plot" object is composed of a BLT graph, two sliders to select the displayed portion of the curve, and two fields showing the x and y values selected by the user.

4.4 Programmable Visualization Features

One of the central tools of *biok* is a spreadsheet specialized in displaying and editing sequences alignments (Figure 6, back window). As in many other scientific research areas, visualization is critical in biology. Several graphical functions, or *tags*, are available in *biok*. A tag is a mechanism that highlights some parts of an object. For instance, segments of protein sequences showing a specific property, such as a high hydrophobicity, can be highlighted in the spreadsheet. Table 1 shows that a tag is composed of two relations:

1. between graphical attributes (such as colors, relief or fonts) and predefined tag values,
2. between these values and positions in the annotated object.

Graphical attributes	*Tag values*	*Locations*
blue	certain	36 .. 55
dark green	putative	137 .. 157
red	AXA	13 .. 15

Table 1: Tag relations

New tags may also be defined with a dedicated editor (Figure 6 (front window)), where the user has to define a method to associate tag values to data positions in a script. This method is typically either a small script for simple tags, or an invocation to more complex methods that run analyses, notably from a Web Server [Let01b].

biok comes with several pre-defined tags, which can be modified, or the user can create new ones with a set of super-classes of tags, that define e.g. row tags, column tags, cell tags, or sequence tags. For example, a tag class Toppred, named after a published algorithm for detecting transmembrane segments in a protein sequence [Hei92], is available in *biok*. This tag was implemented by a biology student during her internship (section 5.2). Section 5.4 reports a concrete case of a tag extension which highlights non-conventional patterns.

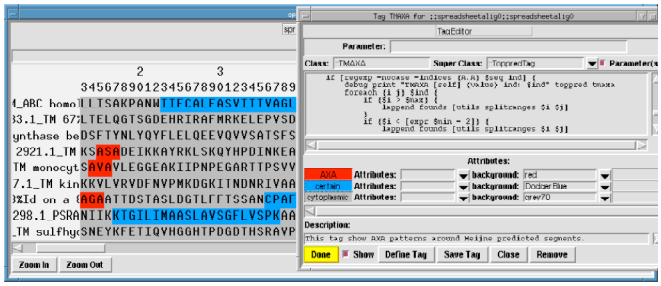


Figure 6: Tag editor (front) and alignment editor (back). In the tag editor, the top

part displays a field to enter parameters for this tag, if needed. The part below contains a superclass chooser and an editor for the code to define tag values. The middle part is a tool to associate graphical attributes to tag values. The bottom part displays a short description of the tag. The positions associated with the various tag values are highlighted in the alignment editor. In this example, a tag showing transmembrane segments (in blue) is extended by a sub-tag highlighting small patterns around them (in red).

The spreadsheet tool, the tag editor and the 3D molecular visualization widget [TNQL03] (Figure 4), have been the subject of numerous workshops. Among these workshops, the workshop dedicated to the design of the tag editor has been described in section 3.7.2.

4.5 Programming Environment

A method editor lets users redefine any method, save it, print it, search its code, and try it with different parameters (which just makes the editor a form to run the code), to set breakpoints, and to ask for trace.

User-created or re-defined methods and classes are saved in a user directory, on a per-class and per-method basis. This enables the user to go back to the system's version by removing unwanted methods files, and to manage the code outside of *biok*, with the system file commands and the preferred code editor. This double access and the fact that the result of user's programming work is not hidden in a mysterious format or database, where source code is stored as records organized in a way that the users are not able to understand, is very important to us.

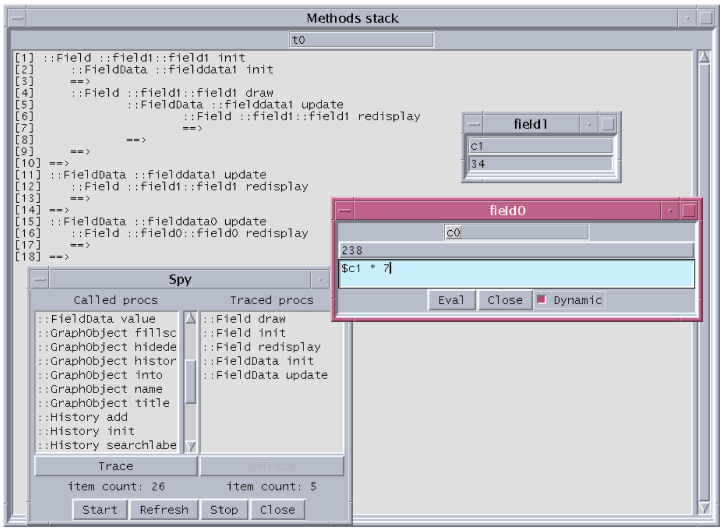


Figure 7: Spying and Tracing method calls. Object "c0" uses "c1" in its formula.

The "Methods stacks" window displays methods called when a change occurs in "c1". The "Spy" window enables to start/stop spying and to select methods for being traced.

biok features several navigational tools that enable the user to locate source code directly from the user interface, whenever needed during standard use of the tool:

1. methods browser and inspector available from any graphical objects: source code is available at the method level, on a per-object basis,
2. search tools for classes and methods that enable to search the name, body or documentation,
3. a classes browser.

The methods browser can be triggered from any graphical object, by a menu that only displays methods relevant to this object. In this menu, "Data analysis" methods are distinguished from methods dealing with "Graphical Display", the former being often more of interest to the biologists than the latter.

biok also has various debugging tools, that we were able to develop quite easily thanks to the dynamic introspection tools provided by the XOtcl language. Users can print keyed debug messages, put/remove breakpoints on method calls by simply clicking on a button in the method editor, and put breakpoints inside the code.

Figure 7 shows spy and trace tools. A button in the editor enables the user to trace method calls: this opens a "Methods stack" window displaying method calls with parameters and return values. In a way similar to direct activation [WG01], a concept to encourage tailoring, the user can spy the execution during a short period, and select specific methods from the list of those called (see "Spy" window in figure 7). Thus, these tools are useful for both source code navigation and comprehension: combining browsers, spy or trace tools, and breakpoints is a good mean to find the location of a specific problem, or to understand the way the program works.

4.6 Programming errors

A frequent objection to our approach is that opening the source code to end-users might make programming errors or break the system. By providing programming and debugging tools, and by designing the environment for incremental changes, we sought a way to minimize the impact of errors, not to remove them.

Breaking the system is indeed feasible at the user level, in the same way as with an open-source software that the user can download, install and modify locally. However, whenever the system incidentally breaks, there is a simple way to go back to a previous state: user source is saved as simple text files organized in easy to manage directories. If a new method breaks the class, for example, the user only has to remove its corresponding file from its class directory. There is no specific

mechanism implemented yet in the prototype to organize source code management, for this can easily be implemented with standard versioning tools such as CVS.

We observed, though, that biologists are not really interested in challenging software robustness! On the contrary, as we observed during student projects (section 5), they are very cautious, and generally only focus on the part of the code they feel they have control over. According to [WG01] or [Mac91a], tailoring has even rather to be encouraged than prevented.

4.7 Design rationale summary

Our focus in *biok* is not on programming, but rather on the programming *context*. We provide however a full access to programming *as a key feature* of the tool. *biok* contextualizes the task of programming with respect to the biologist's scientific tasks and motivations. We stress:

1. *Focus*: the focus of the tool is on biology or bioinformatics tasks; coding software is possible, but not obligatory.
2. *Incremental programming*: programming from scratch is difficult: whereas modifying an existing, working program, especially if it includes domain-centered examples, is motivating and helpful. The PITUI (Programming In The User Interface) approach enables incremental programming and progressive learning [CSBA90].
3. *Integration of the graphical and coding levels*: an important aspect of a programmable application is to integrate code objects of interest and graphical representation of these objects [WG01]. Integration should work both from the graphical object to the code object, and from the code to the graphical object. In [Eis95], some variables describe the user interface state, and some commands for modifying this state are available at the scripting level. The notion of graphical objects gets close to the extreme integration of graphical and coding elements that is provided in Boxer [DA89] or Self [SMU95]. Following the spreadsheet paradigm, and whenever possible, graphical objects of interest, such as sub-areas of objects, or tags, are available from the code.
4. *A good programming environment*: motivation for the user to program, although existing, is probably discouraged by common standard environments [LF95].

5 Reports on the uses of the prototype

This section reports various uses of the prototype over the last years. It has three purposes: to show how the prototype can be used as a programming environment, either to tailor or to create simple functions; this also illustrates an aspect of the participatory programming process, where programming artifacts produced by end-users can be incorporated back into the tool to be shared with other users (sections 5.2 and 5.3); to show that domain orientation is obviously important to sustain

programming activity (5.2) and to provide tailoring affordances (5.4); this shows however that this does not exclude encoding with a standard programming language as discussed in section 3.6; to illustrate the use of the prototype for design or re-design purposes (section 5.5).

5.1 Methods of the Evaluation Studies

Even though *biok* is not specifically aimed at students, it has mostly been used by them. The main reason for this is that it is a prototype and this is why I have preferred not to distribute it, for the moment. These students, however, were biologists with bioinformatics training, including a first exposure to programming (from a few hours to several weeks). An important part of bioinformatics work, and this is not only true in the Institut Pasteur, is performed by students, this makes them significant users of the tool. Moreover, none of these students were obliged to use the prototype. As a matter of fact, some of them did not, but used more standard programming tools such as a text editor and the Tcl interpreter instead. More established scientists indirectly used biok's although not alone, because it is not finished and stable enough, hence they needed my help. They were interested by its incremental programming features particularly for visualization. For one of them, the possibility to superimpose several visualizations of patterns was interesting (see a description in section 5.4). Other scientists reported their need to use the prototype and urged me to finish it. For one of them, an alignment editor was a pivotal tool in bioinformatics, since it helps produce data that lead to a considerable quantity of other analysis tools. Added to this, she said that it was essential to be able to add functions, because it is impossible to have every function provided for in a single tool. Other scientists stressed the value of Web server tools integration [Let00].

Student internships brought various types of information. None of projects that are described here can be considered as a proper user study with controlled setting. Our approach seeks to explore a complex problem space by conducting user studies and workshops, rather than to evaluate specific solutions. However, we did some user testing (section 3.7.2) and since the prototype - that was still at development stage - has been used on a daily basis during the internships, the students' projects were an opportunity to informally observe how the environment and the task-centered tools could help.

Generally speaking, most of the students used a great deal the environment either for locating examples of code by using the navigation and search tools, for debugging their code and understanding interactions between objects, or just for modifying simple methods, for instance either by adding a parameter to a method that calculates the hydrophobicity of a protein, or by adding a default value to a parameter, or by adding a branch in a conditional, in a function that select the appropriate hydrophobicity scale for a given protein. The following sections provide a more detailed description of some specific projects using *biok*.

5.2 Learning to program

A first two months project (Spring 2001) involved a student in biology having learnt some programming at the university, in a different language (Python). She first had a few days training in Tcl and *biok*, either from the documentation provided on a Web page, from Tcl books or with assistance from me. Then she had to implement a published algorithm [Hei92] to predict transmembrane segments in proteins.

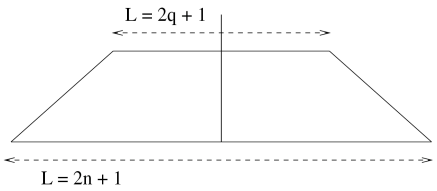


Figure 8: Heijne algorithm

The algorithm consists in the computation of a weighted sum: $h_i * w_i$ on a sliding window (figure 8), with:

h_i = amino-acid hydrophobicity values

w_i = i/S for $1<=i<=n-q+1$; $(n-q+1)/S$ for $(n-q+1)<i<(n+q+1)$; $(2n+2-i)/S$ for $(n+q+1)<=i<=2n+1)$

with a normalization factor: $S=(1+n)2-q2$ to get:

$$\text{Sum}_{i=1}^{2n+1} w_i = 1$$

At first, the student encountered a lot of problems in programming, since the course she had was too short and too difficult. At the beginning, she had no or very few understanding of computing or programming concepts, such as functions, loops, parameter passing, etc... She was really discouraged.

The human environment helped a lot: several computer scientists of the team gave her advice, and she could at any time ask for information. We believe that *biok* helped her mainly by bringing a *real motivation*. We observed her positive reaction the first time she obtained a display of the curve she had to compute for the algorithm - a hydrophobicity curve (figure 5). From this moment, she progressed much faster and explored spontaneously various alternatives, new computations, etc... She was also able to find out - with little help - how to add a graphical component to the plot object (field displaying the pointed location on the curve). Besides, she confirmed in interviews that programming with objects of interest

makes a real change.

She also helped a lot to enhance *biok*. Convenient although very simple features, such as the "Print" button in the method editor or in the Plot object were added. She was always doing a cut-and paste of single methods code into the emacs editor just to get a printed copy. The formula-level history also originated from seeing her copy-and-pasting the very same formula for the very same kind of objects.

However, she almost never used the debugging tools, although we did a quick demo. The reason for her not using the trace tool was probably that she had a dozen methods to program, where the order of method calls was always the same. The only tool that could have been helpful is the keyed print statement, to visualize variables' values, but this mechanism was too complex, compared to a simple print that one can interactively comment out. Furthermore, the breakpoint mechanism was not ready at the time of her internship.

We observed that implementing this type of algorithm (about 300 lines of code, divided in about 10 functions, with some simple embedded loops), is a current practice among bioinformaticians. Even though the program corresponding to the published algorithm is generally available from the authors, researchers might need to apply it to a variant set of data, or to take only a part of it for a similar problem. However, even though the code she has developed is now included in *biok*, and is the basis of the visualization tag described in section 4.4, the implementation represented for her only exercise. Why are we reporting about this project? It is to show that that cognitive problems raised by programming decrease in a domain-oriented environment, *even if the programming language is a general-purpose language*. This supports the hypothesis described in section 3.6.

5.3 Adding new features in an existing component and connecting objects

Another bioinformatics student, who had learnt programming for a few months, spent 6 months (Spring 2002) on a project where she had to refine a graphical object defined in *biok*, on top of a Tk widget for visualizing a molecule. She also had to link this object to the alignment editor, in order for features common to both representations to be simultaneously displayed by the mean of tags (section 4.4) by using a simple protocol that is provided in *biok* to enable the user to synchronize selections (see section 6.3.2 and figure 4). This student was of course able to program, although as a beginner, for she had just learnt programming. The main benefit of this project for our approach was to provide a test-bench for our environment, that she used all the time. In particular, she used the debugging tools (section 4.5) that proved quite useful to program interactions between graphical objects. She also used the method editor all the time, even though it is not a full featured editor, and although the use of another external editor such as emacs is possible in the environment. Several technical aspects we focused on in the design proved to be really useful for her: as a biologist, she especially liked the focus on the task. The incremental programming idea, and the direct connection between

graphical objects and code enabled her to better control the appropriate parts of the program.

5.4 Tailoring a visualization function

We illustrate here a situation where a biologist came to me because he knew about the programming features of biok and he knew that it was the only mean to get his peptide features visualized in a reasonable time. This scientist wanted to search in a large set of protein sequences for a signal peptide recognized by non-conventional secretion enzymes. For this purpose, he had to check whether a short pattern, composed of 3 letters and corresponding to the following regular expression: *A.A*, also occurred in his sequences, either before the first predicted transmembrane segment, or after the last one. Defining a new tag for this purpose, required:

- to define a new tag in the tag editor as a sub-class of the Toppred tag (a menu to select a base-class is provided),
- to add about 20 lines of code to:
    - search for the positions of the first and last segment in a list of segments,
    - search for an occurrence of the *A.A* pattern before the first position or after the last one,
- associate a color (red) to this new tag (Figure 6).

It is worth noting that such a specific problem could not have been anticipated in a general-purpose bioinformatic tool. This newly created tag, once saved, is then available to the user for the current and next sessions. It can be sent to a colleague as a separate and easy to locate file to be copied in his or her *biok* classes directory.

5.5 A prototyping tool for design

One convenient aspect of the sequences alignment editor is its flexibility and, thanks to its adequation to users domain and work practices, it also proved to be a quite powerful tool to explore design ideas.

Unexpectedly using the alignment editor to align analyses

Figure 9 shows the visualization of three independent analyses of the same sequence. Although the possibility to compare analyses was not really anticipated, it could be quickly developed during the preparation of a workshop where visualization issues were addressed.

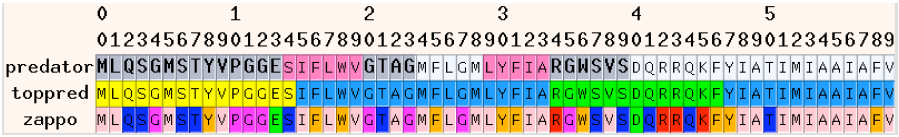


Figure 9: Comparing analyses: predator *is a secondary structure analysis, showing helices and sheets,* Toppred *is a transmembrane segments predictor, another kind of secondary structure specific to membrane proteins, and* zappo *is a physico-chemical analysis, showing hydrophobic amino-acids, very often found in helices and transmembrane segments*

Exploring algorithmic problems

biok has been used in several occasions as an environment to explore new ideas regarding algorithmic issues. Section 6.1 reports on an attempt to open unexpected points of interactions within an algorithm. In this situation, we were again able to quickly prototype the idea, before re-implementing it in a more efficient programming language.

Demonstrating ideas for participatory workshops

Although we have not directly used *biok* during workshops, as in [BG91], the tool has often been used before participatory workshops to demonstrate some ideas and open the design space.

6 Between End User Programming and Open Systems: A final reflexion

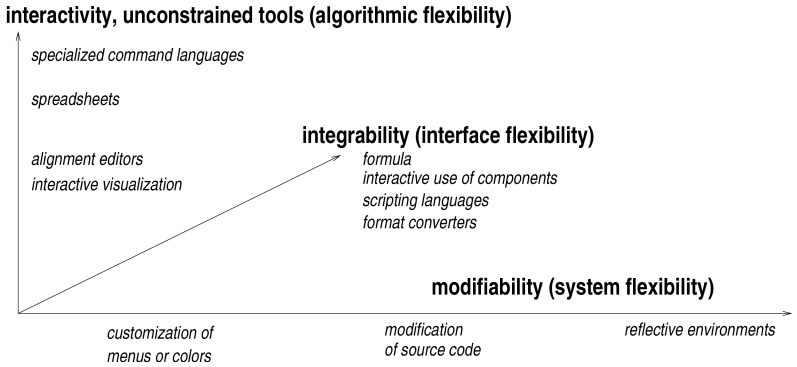
We have analyzed the programming practices among biologists and observed the context of programming, which often consists in adapting software to numerous specific and diverse situations. We wanted, as far as possible, to better deal with *unanticipated* software evolution [LZ03] and adaptation, so it appeared important to consider general software flexibility issues. In [vR99] Rossum advocates for the access to programming for everybody through computing education, development of better programming tools and the building of a community of users. In this approach, access to programming not only enables end-users to build small programs or to customize their tools, but also to *modify* them. The approach also explicitly relies on the use of a general-purpose programming language, such as Python, that is easy for beginners to learn and yet is suited for building real-world professional applications. Thus it goes beyond End-User Programming as described in [Lie00]. We agree with Rossum that, following the open source movement, such a desirable development of end-user programming will change the nature of the software development process. In our approach however, even though our objectives are very similar, we believe that more results from the EUD field should be taken into account to enhance access to programming. Powerful programming tools, such

as enhanced Undo or program visualization tools are envisioned in [vR99]. But these tools are still quite programmer-oriented and lack data visualization features or lack links to the domain-oriented concepts; this proved critical in the success of *biok* as a programming environment. Moreover, we believe that programmability requires powerful mechanisms such as introspection, that are lacking in Python, as well as powerful concepts such as meta-level interfaces, hence we will describe which general principles should be applied, related to software flexibility, as described by the Open Systems approach [KdRB91][Dou96]. This section, by describing how these principles could be applied in EUD, is an attempt at bridging the gap between EUP, CP4E[2] and Open Systems. We first report on the specific software flexibility issues we observed in our user studies (section 6.1). We then relate these problem to the work on reflective architectures and open system protocols (section 6.2). Finally (section 6.3), we describe how these principles have been applied in the *biok* architecture and how they could be adapted to End-User Programming.

6.1 Dimensions of flexibility

Object-oriented and component technology are generally acknowledged as an all purpose solution to achieve software flexibility, and this is why *biok* is based on object-oriented constructs. Yet, can we entirely rely on these technologies to address unanticipated software changes by the end-user? [Boy98] [SBB02] or [CC00] are examples of this approach in the field of bioinformatics, and at least for the latter ones, are extensively used even by novice programmers. However, we have observed during our user studies that this approach has some limitations, compared to the flexibility that biologists need, such as:

1. components should be modifiable (and they are often not),
2. components often do not easily adapt,
3. the vast majority of tools are monolithic applications,
4. flexibility during the computation of a scientific result is often required.



[2] CP4E: Computer Programming For Everybody

Figure 10: Dimensions of flexibility

Figure 10 shows the important flexibility dimensions that emerged during the user studies (section 3.7).

1. Open systems or *system flexibility* addresses the possibilities to change the system, from simple customization to reflective systems.
2. Integrability or *interface flexibility* refers to the possibility to easily combine components. Bioinformatics is a fast evolving field: changes often occur at the component interface level. Typical solutions include dataflow systems [Biz00], wrappers, API [SBB02] and Web services [WL02]. Along this dimension, explicit and interactive interface adaptation features by the end-user could be defined.
3. Interactivity or *algorithmic flexibility* describes systems that give a significant control on the computation to the user, from interactive visualization tools to domain-oriented programming languages such as Darwin [GHKB00]. As observed by Repenning [Rep93], the whole field of HCI aims at building systems that provide control to the user at the appropriate moment. In this view, a programmable software enables the user to control the computation better. Typically, the user can provide hints to the heuristic of the algorithm. In a multiple alignment of several sequences, the user could control the order in which the sequences are compared. Interestingly, opening unforeseen points of control in a tool does not lead to more programming but to *more interaction*. At one end of this spectrum there are *unconstrained tools*, such as spreadsheets and word processors which according to [NJ94], lead to a level of flexibility necessary for visualizing scientific data (see an example in section 5.5). The more a system can be progressively adjusted with parameters in a rich dialog between the user and the tool, the more flexible and adaptable it becomes [BHP94]. One could even say that the most adjustable systems are these unconstrained tools we have just mentioned, (i.e.) systems whose "result" are entirely decided by the user. In bioinformatics, several tools already enable the user to interactively steer the computation, or even change results. Several tools [Let01b] allow the manual change of the output of algorithms that compute multiple alignments. However, changing the results can provoque mistakes. In [LZ03], we describe an attempt to open new unexpected points of control in the algorithm: this mixed-initiative approach enhances the final result and prevents such mistakes.

6.2 Reflective and Open Architectures for Unanticipated Changes

There are systems that anticipate their own modification: this starts from customizable systems, up to meta and reflective systems.

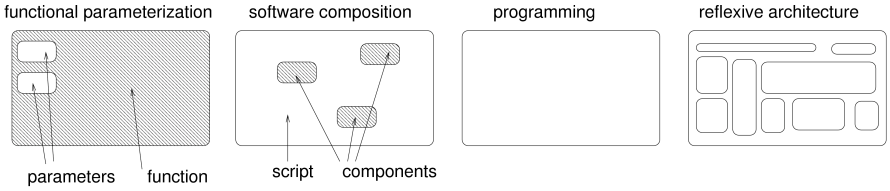
Nierstrasz [NT95] identifies three main levels of software flexibility (three first levels of Figure 11):

1. *functional parameterization*, where the parameters may be values or functions, and where the system is composed of functions;
2. *software composition*, where the parameterized part includes the scripting to glue software components;
3. *programming*, where the user input is the program code, and the system is the programming language.

The third level is thus the most flexible, but too much freedom does not help the end-user, who needs scaffoldings. This is why a fourth level is needed: *reflexive architectures*, that both provides full programming and explicit structures for parameterization.

6.2.1 Flexibility and Reflexivity

Our goal is to achieve a general flexibility, similar to that in programming. In figure 11, the fourth level shows that everything is "open" to the user, everything in the system becomes a parameter. Yet, as opposed to free programming (third level in figure 11), here there is a scaffolding. This scaffolding is composed of an existing program, with components, structures, examples, as well as an environment to help use these objects. We put this approach in a coherent way related to free software and open systems, but this freedom does not prevent an inexperienced user to be involved.



	param. fonct.	software composition	programming	reflexive architecture
parameters	*value, function*	*script, graphical assembly*	*program*	*program*
system	*program, function*	*function, component*	*prog. language*	*prog. language + explicit struct.*

Figure 11: Software levels of flexibility: the blank part is the "free" part for user input, and the grey part the system.

As demonstrated by [Rep93], one can consider that *the more a system makes its internal objects* - structure, functions, values, data structures - *explicit, the more flexible it is*. This principle is indeed made systematic in the reflective systems approach, which uses internal representation for standard computation and behavior [Mae87]. The principle that we have followed in *biok*, is both to provide a structured underlying architecture and framework to help the understanding of the code (see section 6.3), and to provide dynamic navigation tools to help locate source code

involved in a specific feature (section 4.5 and figure 7).

6.2.2 Meta-objects protocols

Providing a reflective architecture requires a specific internal design, where internal components are available as an additional meta-level interface, potentially subject to modifications. The meta-object protocol (MOP) technology [KdRB91] gives a model of an explicit modification protocol for systems where changes of the system are anticipated. MOP was originally intended for object-oriented languages, to let the user change the way object, classes, inheritance, etc... behave. Figure 12 illustrates that this approach can be transposed to standard applications, as was done by Rao [Rao91] for a window system or by Dourish [Dou96] for a CSCW toolkit.

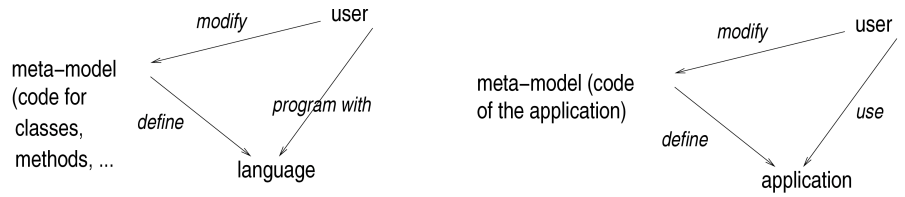


Figure 12: From Meta-Object Protocol to Meta-Application Protocol

6.3 Flexibility for the User

A reflective architecture does not only require an additional interface. Giving a non-specialist that many possibilities to perform complex actions raises a usability issue. As explained by [Mor97], or as modeled by [dCdS03], the more the user interface offers programmability, the less usable the system is, since the user interface language departs from the user's task.

To avoid user confusion, a compromise must be found to deal with these different representations. How can we help the user understand which part of the source code corresponds to such and such user interface components? How can we articulate both languages by using an intermediate representation level? How can we structure the code in small enough components that correspond to the user's domain task units? In other words, internal architecture has to be handled and designed as an explicit, although additional, user interface. In section 3.7, we explained that this design was greatly influenced by the observations we made during interviews and the ideas that emerged in participatory workshops.

6.3.1 Explicit Intermediate Meta-level Interfaces

In the context of EUP, several approaches exist to manage intermediate levels of access from the user interface to the code. Morch [Mor94] suggests a design where tailoring can occur at three levels: customization, integration, or extension. Extension can be performed through three kinds of interfaces: the standard user

interface, design rationales or source code. The design rationale fills the gap between the source code and the user interface. It is a pedagogical go-between which explains the behavior of the application units. For instance, a diagram can explain how the code deals with mouse movements in a drawing application.

More generally, our approach draws from the MAP model (figure 12). Not only must this interface be explicit, it *must also belong to the user's working environment* so that the user does not have to switch from his or her work environment to an encoding environment.

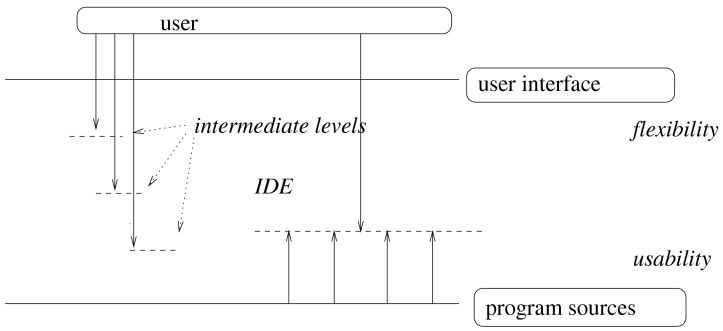


Figure 13: Adding intermediate programming levels and improving the programming environment usability: two complementary approaches

In order to achieve this, we have built two kinds of intermediate interfaces that are directly accessible from the application: intermediate programming levels and a programming interface (Figure 13). Programming levels and intermediate meta-level user interfaces include:

1. a formula spreadsheet-like level (section 4.3),
2. a tag programming level to visualize biological functions (section 4.4),
3. a scripting level to facilitate program composition,
4. an object-oriented implementation level with a full-fledged integrated programming environment (section 4.5).

6.3.2 Internal Framework: Explicit Elements of a Protocol

In addition, we not only structure the software in order to make it "source-navigable", we also borrow from the MOP approach the idea of having the source code follow a documented and explicit framework. In order to do this, we need well-known method names where the user can learn to go [3] . In *biok*, graphical objects

[3]Notice however that we are not looking for a framework-based approach. *biok* is not an abstract set of classes that first need to be instantiated. Yet, there is a

define a set of protocols:

- *Graphical display*. Graphical objects define `draw` and `redisplay` methods for graphical components to be created, initialized and refreshed. If, for example, fields were missing in the tool for displaying curves, the user would just have to edit the `draw` method. This is what happened with a student who wanted to add the `x` and `y` fields in the initial tool (Figure 5).
- *Synchronized selections*. A simple protocol has been defined with another biology student as a set of three generic methods. These methods define which selections in objects should be synchronized (`interact`); what to do when a synchronization should occur (`propagate`); how to enforce selection in the target object (`highlight`).
- *Persistence*. Two methods deal with object contents (`value`) and persistence (`save`).

6.4 Concluding Remarks on Flexibility

In this section, we have tried to show that general software flexibility is desirable for educated end-users, as long as explicit tools are designed for it, and that this scaffolded flexibility is feasible through reflective mechanisms. We preferred to adopt a reflective architecture rather than a more explicit meta-descriptive system. The first reason is that the latter solution is more costly: we have chosen to reuse descriptive constructs (classes and methods) instead of rewriting a language, and to have them available and modifiable through introspection. Secondly, true reflective mechanisms ensure a causal connection between the running system and its source code [Mae87]. Finally, an explicit meta-descriptive system requires an additional abstraction level. Bentley et al [BD95] have demonstrated that computer abstractions, unlike mathematical ones, are often compromises, leaving potentially important aspects out of its scope. Hence, instead of being a positive tool to structure application, they become a barrier.

7 Conclusion

We built *biok* as an environment that enables biologists to conduct data analyses, combine them and visualize the results with graphical tools. In this environment, and according to their needs, biologists can locate and modify the code of methods, and create new ones. Through familiar entities such as programmable graphical objects corresponding to domain concepts, *biok* makes programming more accessible, but it still requires a basic knowledge of programming, as in [vR99] or [Eis95]. Being embedded within a running application, the programming meta-level has to rely on a well designed internal architecture [KdRB91], where flexibility dimensions carefully correspond to the users needs. Participatory programming, a

---

documented framework within the running application. In *biok*, programming is possible, not required.

process that integrates participatory design and End-User Programming, leads to enhanced flexibility in the right places [Tri92][SKW97][KM95]. Our work consists more in the exploration of the problem space: we wanted to investigate on the *context* of the programming tasks rather than programming itself, by addressing the following issues: *what* do users want or need to program? *when* do users want or need to program? The main outcome was that biologists preferred to program in the context of normal software use, or even that *they preferred not to program at all*. An important consequence is that a software with programming facilities should, through a careful design, both *maximize* the available programming features, and *minimize* the programming needs. This is why a better cooperation should take place in the building of software. Indeed, we discovered that problems arising when biologists need to program lie in the way common software is built rather than in the difficulty of the programming activity itself. This is why we shifted the problem focus from programming to flexibility, in order to take into account the fact that programming, in our context, is neither the goal nor the main difficulty for biology researchers.

Acknowledgements

My thanks to the many biologists, programmers and bioinformaticians who participate to the interviews and workshops. Special thanks to Volker Wulf, Wendy Mackay, Michel Beaudouin-Lafon, Katja Schuerer, Alexandre Dehne Garcia, Fabienne Dulin, Albane Le Roch, Alexis Gambis, Marie-France Sagot, Thierry Rose, Pierre Tuffery, Victoria Dominguez, Francois Huetz, Lionel Frangeul, Bertrand Neron, Pierre Dehoux and Stephane Bortzmeyer. Many thanks to Andrew Farr for his helpful assistance on the English.

References

[BD95] R. Bentley and P. Dourish. Medium versus mechanism: Supporting collaboration through customization. In: Proceedings of ECSCW'95, 1995, pp. 133–148.

[BG91] S. Bodker and K. Gronbaek. Design in action: From prototyping by demonstration to cooperative prototyping. In: Design at Work: Cooperative Design of Computer Systems. Hillsdale, New Jersey Lawrence Erlbaum Associates, 1991, pp. 197–218.

[BHP+94] M. M. Burnett, R. Hossli, T. Pulliam, B. VanVoorst, and X. Yang. Toward visual programming languages for steering in scientific visualization: a taxonomy. In: IEEE Computational Science and Engineering, 1994, pp. 44–62.

[Biz00] J.W. Bizzaro. Distributing scientific applications with Gnu Piper. Technical report, Bioinformatics.org, 2000. http://bioinformatics.org/piper.

[Bla00] A. F. Blackwell. Swyn: A visual representation for regular expressions. In: Your Wish is My Command: Giving Users the Power to

Instruct their Software. Morgan Kaufmann, 2000, pp. 245-270.

[Bla02] A. F. Blackwell. What is programming? In: Proceedings of PPIG 2002, pp. 204–218, 2002.

[Boy98] J. Boyle. A visual environment for the manipulation and integration of java beans. In: Bioinformatics, Volume 14, Issue 8, September 1998, September 1998, pp. 739–748.

[CC00] B. Chapman and J. Chang. Biopython: Python tools for computation biology. In: ACM-SIGBIO Newsletter, August 2000.

[CFL+03] M.F. Costabile, D. Fogli, C. Letondal, P. Mussio, and A. Piccinno. Domain-expert users and their needs of software development. In: Proceedings of the HCI 2003 End User Development Session, 2003.

[CR87] J. M. Carroll and M. B. Rosson. The paradox of the active user. In: Interfacing Thought: Cognitive Aspects of Human-Computer Interaction. J.M. Carroll, Ed. Cambridge, Mass: MIT Press, 1987, pp. 80–111.

[CSBA90] J. M. Carroll, J. A. Singer, R. K. E. Bellamy, and S. R. Alpert. A view matcher for learning Smalltalk. In: Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems, ACM Press, 1990, pp. 431 – 437.

[Cyp93] A. Cypher. Watch What I Do. Programming by Demonstration. MIT Press, 1993.

[DA89] A. DiSessa and H. Abelson. Boxer: a reconstructible computational medium. In: Studying the Novice Programmer, Lawrence Elbaum Associates, 1989, pp. 467–481.

[dCdS03] C. K. V. da Cunha and C. S. de Souza. Toward a culture of end-user programming: understanding communication about extending applications. In: Proceedings of the CHI'03 Workshop on End-User Development, April 2003.

[DE95] C. DiGiano and M. Eisenberg. Self-disclosing design tools: a gentle introduction to end-user programming. In: G. Olson and S. Schuon, editors, In: Proceedings of DIS'95 Symposium on action Systems, ACM Press, Ann Arbor, Michigan, 1995, pp. 189–197.

[DiS99] A. DiSessa. Changing Minds: Computers, Learning, and Literacy. MIT Press, 1999.

[DLL03] Y. Dittrich, L. Lundberg, and O. Lindeberg. End user development by tailoring. Blurring the border between use and development. In: Proceedings of the CHI'03 Workshop on End-User Development, April 2003.

[Dou96] P. Dourish. Open Implementation and Flexibility in CSCW Toolkits. PhD thesis, Dept of Computer Science, University College, London, 1996.

[Eis95] M. Eisenberg. Programmable applications: Interpreter meets interface.

ACM SIGCHI Bulletin, 27(2), April 1995, pp. 68–93.

[Eis97] M. Eisenberg. End-user programming. In: Handbook of Human Computer Interaction, second, completely revised edition. North-Holland, 1997, pp. 1127–1146.

[Fis03] G. Fischer. Meta-design: Beyond user-centered and participatory design. In: Proceedings of HCI International 2003, Constantine Stephanidis (ed.), Crete, Greece, June 2003, pp. 78–82.

[FO02] G. Fischer and J. Ostwald. Seeding, evolutionary growth, and reseeding: Enriching participatory design with informed participation. In: Proceedings of the Participatory Design Conference (PDC'02), T. Binder, J. Gregory, I. Wagner (Eds.), Malm_ University, Sweden, June 2002, CPSR, P.O. Box 717, Palo Alto, CA 94302, 2002, pp. 135–143.

[FS00] G. Fischer and E. Scharff. Meta-design–design for designers. In: Proceedings the 3rd International Conference on Designing Interactive Systems (DIS 2000); D. Boyarski and W. Kellogg, Eds, New York City, ACM, August 2000, pp. 396–405.

[GHKB00] H.H. Gonnet, M.T. Hallett, C. Korostensky, and L. Bernardin. Darwin v. 2.0: an interpreted computer language for the biosciences. In: Bioinformatics, 16(2), 2000, pp.101–103.

[GN92] M. Gantt and B. A. Nardi. Gardeners and gurus: patterns of cooperation among CAD users. In: ACM conference on Human Factors in Computing Systems (Proceedings) (CHI '92), ACM Press, 1992, pp. 107–117.

[Gre93] J. Greenbaum. PD, a personal statement. In: CACM, 36(6), June 1993, p. 47.

[Hei92] G. von Heijne. Membrane protein structure prediction. hydrophobicity analysis and the positive-inside rule. In: Journal of Molecular Biology, 225(2), 1992, pp. 487–494.

[HK91] A. Henderson and M. Kyng. There's no place like home: Continuing Design in Use. In: Design at Work: Cooperative Design of Computer Systems. J. Greenbaum and M. Kyng,, Eds. Hillsdale, New Jersey Lawrence Erlbaum Associates, Publishers, 1991, pp. 219–240.

[Kah96] H. Kahler. Developing groupware with evolution and participation. a case study. In: Proceedings of the Participatory Design Conference 1996 (PDC'96), Cambridge, MA, 1996, pp. 173–182.

[KdRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. The Art of the Meta-Object Protocol. MIT Press, Cambridge (MA), USA, 1991.

[KM95] A. Kjaer and K.H. Madsen. Participatory analysis of flexibility. In: CACM, 38(5), May 1995, pp. 53–60.

[lab87] Labview: a demonstration. unpublished, 1987.

[Let99a] C. Letondal. A practical and empirical approach for biologists who almost program. In: CHI'99 Workshop on End-User Programming and

Blended-User Programming, May 1999. http://www.pasteur.fr/ letondal/Papers/chi_pp.html.

[Let99b] C. Letondal. Résultats de l'enquête sur l'utilisation de l'informatique à l'institut pasteur. Technical report, Institut Pasteur, Paris, April 1999.

[Let99c] C. Letondal. Une approche pragmatique de la programmation pour des biologistes qui programment presque. In: Actes Onzième confèrence francophone sur l'Interaction Homme Machine, IHM'99, Montpellier (France), tome II, November 1999, pp. 5–8. http://www.pasteur.fr/ letondal/Papers/pc_ihm99.ps.gz.

[Let00] C. Letondal. A web interface generator for molecular biology programs in UNIX. Bioinformatics, 17(1), 2000), pp. 73–82.

[Let01a] C. Letondal. Programmation et interaction. PhD thesis, Université de Paris XI, Orsay, 2001.

[Let01b] C. Letondal. Software review: alignment edition, visualization and presentation. Technical report, Institut Pasteur, Paris, France, may 2001. http://bioweb.pasteur.fr/cgi-bin/seqanal/review-edital.pl.

[LF95] H. Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In: Proceedings of ACM conference on Human Factors in Computing Systems (Summary, Demonstrations) (CHI '95). ACM Press, 1995, pp. 480–486.

[Lie00] H. Lieberman, Ed. Your Wish is My Command: Giving Users the Power to Instruct their Software. Morgan Kaufmann, 2000.

[LM04] C. Letondal and W. E. Mackay. Participatory programming and the scope of mutual responsibility: Balancing scientific, design and software commitment. In: Proceedings of the eighth biennial Participatory Design Conference (PDC 2004), Toronto, Canada, July 2004.

[LS02] C. Letondal and K. Schuerer. Course in informatics for biology. Technical report, Institut Pasteur, Paris, 2002. http://www.pasteur.fr/formation/infobio.

[LZ03] C. Letondal and U. Zdun. Anticipating scientific software evolution as a combined technological and design approach. In: USE2003: Proceedings of the Second International Workshop on Unanticipated Software Evolution, 2003.

[Mac91a] W. E. Mackay. Triggers and barriers to customizing software. In: Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems. ACM Press, 1991, pp. 153–160.

[Mac91b] W. E. Mackay. Users and Customizable Software: A Co-Adaptive Phenomenon. PhD thesis, Massachusetts Institute of Technology, 1991.

[Mae87] P. Maes. Concepts and experiments in computational reflection. In: Proceedings of the OOPSLA'87: Conference on Object-Oriented Programming Systems, Languages and Applications, Orlando, FL, 1987, pp. 147–155.

[MCLM90] A. MacLean, K. Carter, L. Lovstrand, and T. Moran. User-tailorable systems: Pressing the issues with buttons. In: Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems. ACM Press, 1990, pp. 175–182.

[Mor94] A. Morch. Designing for radical tailorability: Coupling artifact and rationale. In: Knowledge-Based Systems, 7(4), December 1994, pp. 253–264.

[Mor97] A. Morch. Method and Tools for Tailoring of Object-oriented Applications: An Evolving Artifacts Approach. PhD thesis, Department of Informatics, University of Oslo, April 1997.

[Nar93] B. A. Nardi. A small matter of programming: perspectives on end user computing. MIT Press, 1993. 162 pages.

[NJ94] B. A. Nardi and Jeff A. Johnson. User preferences for task specific vs. generic application software. In: ACM conference on Human Factors in Computing Systems (Proceedings) (CHI '94). ACM Press, 1994, pp. 392–398.

[NT95] O. Nierstrasz and D. Tsichritzis, Eds. Object-Oriented Software Composition. Prentice Hall, 1995. 361 pages.

[NZ00] G. Neumann and U. Zdun. Xotcl, an object-oriented scripting language. In: Proceedings of 7th Usenix Tcl/Tk Conference (Tcl2k), Austin, Texas, Feb. 14-18, 2000.

[OAKB01] V. L. O'Day, A. Adler, A. Kuchinsky, and A. Bouch. When worlds collide: Molecular biology as interdisciplinary col laboration. In: Proceedings of ECSCW'01, 2001, pp. 399–418.

[Ous98] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. In: IEEE Computer, 31(3), 1998, pp. 23–30.

[PL92] D. Ploger and E. Lay. The structure of programs and molecules. In: Journal of Educational Computing Research, 8(3), 1992, pp. 347–364.

[PRM01] J.F. Pane, C.A. Ratanamahatana, and B. Myers. Studying the language and structure in non-programmers' solutions to programming problems. In: International Journal of Human-Computer Studies, 54(2), February 2001, pp. 237–264.

[Rao91] R. Rao. Implementational reflection in silica. In: ECOOP '91 (LNCS 512), ACM Press, July 1991, pp. 251–267.

[RASW90] J. Rasure, D. Argiro, T. Sauer, and C. S. Williams. A visual language and software development environment for image processing. In: International Journal of Imaging Systems and Technology, 2, 1990, pp. 183–199.

[Rep93] A. Repenning. Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments. PhD thesis, University of Colorado at Boulder, 1993.

[SBB+02] J. E. Stajich, D. Block, K. Boulez, S. E. Brenner, S. A. Chervitz, C. Dagdigian, G. Fuellen, J. G.R. Gilbert, I. Korf, H. Lapp, H. Lehvaslaiho, C. Matsalla, C. J. Mungall, B. I. Osborne, M. R. Pocock, P. Schattner, M. Senger, L. D. Stein, E. Stupka, M. D. Wilkinson, and E. Birney. The bioperl toolkit: Perl modules for the life sciences. In: Genome Research, 12(10), 2002, pp. 1611–1618.

[Sch03] K. Schuerer. Course in informatics for biology: Introduction to Algorithmics. Technical report, Institut Pasteur, Paris, France, 2003. http://www.pasteur.fr/formation/infobio/algo/Introduction.pdf.

[SKW97] O. Stiemerling, H. Kahler, and V. Wulf. How to make software softer - designing tailorable applications. In: Proceedings of DIS'97 (Amsterdam), 1997, pp. 365–376.

[SMU95] R. B. Smith, J. Maloney, and D. Ungar. The Self-4.0 user interface: Manifesting a system-wide vision of concreteness, uniformity, and flexibility. In: Proceedings of OOPSLA '95, 1995, pp. 47–60.

[SU95] R. B. Smith and D. Ungar. Programming as an experience: The inspiration for Self. In: Proceedings of ECOOP '95, 1995, pp. 303-330.

[SUC92] R. B. Smith, D. Ungar, and B-W. Chang. The use-mention perspective on programming for the interface. In: Languages for Developing User Interfaces. Jones and Bartlett, Publishers, Boston, 1992, pp. 79-89.

[Tis01] J. Tisdall. Why biologists want to program computers. Technical report, O'Reilly, October 2001. http://www.oreilly.com/news/perlbio_1001.html.

[TNQL03] P. Tuffery, B. Neron, M. Quang, and C. Letondal. i3DMol: Molecular visualization. Technical report, Institut Pasteur, Paris, France, 2003. http://www.pasteur.fr/ letondal/biok/i3DMol.html.

[Tri92] R. H. Trigg. Participatory design meets the MOP: Informing the design of tailorable computer systems. In: Proceedings of the 15th IRIS (Information Systems Research seminar In Scandinavia), G. Bjerknes, T. Bratteteig and K. Kautz, Eds, August 1992, Larkollen, Norway, 1992, pp. 643-646.

[vR99] G. van Rossum. Computer programming for everybody. Technical report, CNRI: Corporation for National Research Initiatives, 1999.

[WG01] V. Wulf and B. Golombek. Direct activation: A concept to encourage tailoring activities. In: Behaviour and Information Technology, 20(4), 2001, pp. 249 – 263.

[Win95] T. Winograd. From programming environments to environments for designing. In: CACM, 38(6), June 1995, pp. 65 – 74.

[WL95] D. Wetherall and C. J. Lindblad. Extending Tcl for dynamic object-oriented programming. In: Proceedings of the Tck/Tk Workshop 95, Toronto, Ontario, July 1995, 1995.

[WL02] M. D. Wilkinson and M. Links. Biomoby: an open-source biological web services proposal. Briefings in Bioinformatics, 3(4), December 2002, pp. 331–341.

[WP02] L. Wang and P. Pfeiffer. A qualitative analysis of the usability of Perl, Python, and Tcl. In: Proceedings of The Tenth International Python Conference, 2002.