BRIEFINGS IN BIOINFORMATICS. VOL 13. NO 2. 244–257
Advance Access published on 29 July 2011 doi:10.1093/bib/bbr035

A toolbox for developing bioinformatics software

Kristian Rother, Wojciech Potrzebowski, Tomasz Puton, Magdalena Rother, Ewa Wywial and Janusz M. Bujnicki

Submitted: 23rd March 2011; Received (in revised form): 17th May 2011

Abstract

Creating useful software is a major activity of many scientists, including bioinformaticians. Nevertheless, software development in an academic setting is often unsystematic, which can lead to problems associated with maintenance and long-term availibility. Unfortunately, well-documented software development methodology is difficult to adopt, and technical measures that directly improve bioinformatic programming have not been described comprehensively. We have examined 22 software projects and have identified a set of practices for software development in an academic environment. We found them useful to plan a project, support the involvement of experts (e.g. experimentalists), and to promote higher quality and maintainability of the resulting programs. This article describes 12 techniques that facilitate a quick start into software engineering. We describe 3 of the 22 projects in detail and give many examples to illustrate the usage of particular techniques. We expect this toolbox to be useful for many bioinformatics programming projects and to the training of scientific programmers.

Keywords: *software development; programming; project management; software quality*

INTRODUCTION

Software is one of the most visible results of bioinformatics research and development. Several recent articles underline the importance of quality code for a real, long-lasting benefit for science: Hannay *et al*. [1] showed that although scientists generally appreciate a systematic approach to programming, there is a lack of formal training. Merali [2] concludes that although the time spent programming by scientists has increased, the practice to write and test software has not co-evolved resulting in problems with code transparency, maintainability and reproducibility, also see Figure 1A. But what can scientists actually do to improve their programming style?

Professional programmers have been applying methodologies for developing software, which have evolved considerably during recent years. Some take a formally rigid approach, where planning, implementation, and testing phases follow each other linearly, or where strict quality criteria and management processes are defined. These ideas are found in the waterfall model [3], the ISO 9000 family of standards [4], and others. On one hand, such processes are being applied in the software industry from small low-risk projects to projects involving hundreds of developers and severe financial liabilities or health damage in case of a software failure. On the other hand, waterfall-like models have been criticized as too clumsy in many situations.

Corresponding author. Kristian Rother, Laboratory of Structural Bioinformatics, Institute of Molecular Biology and Biotechnology, Collegium Biologicum, Adam Mickiewicz University, ul. Umultowska 89, 61-614 Poznan, Poland. Tel: +48 607095012; Fax: +48 22 5970715; E-mail: krother@genesilico.pl

Kristian Rother is a software developer, bioinformatics trainer and Certified Scrum Master at the AMU Poznan. He has also contributed to the Biopython and Pycogent software libraries.

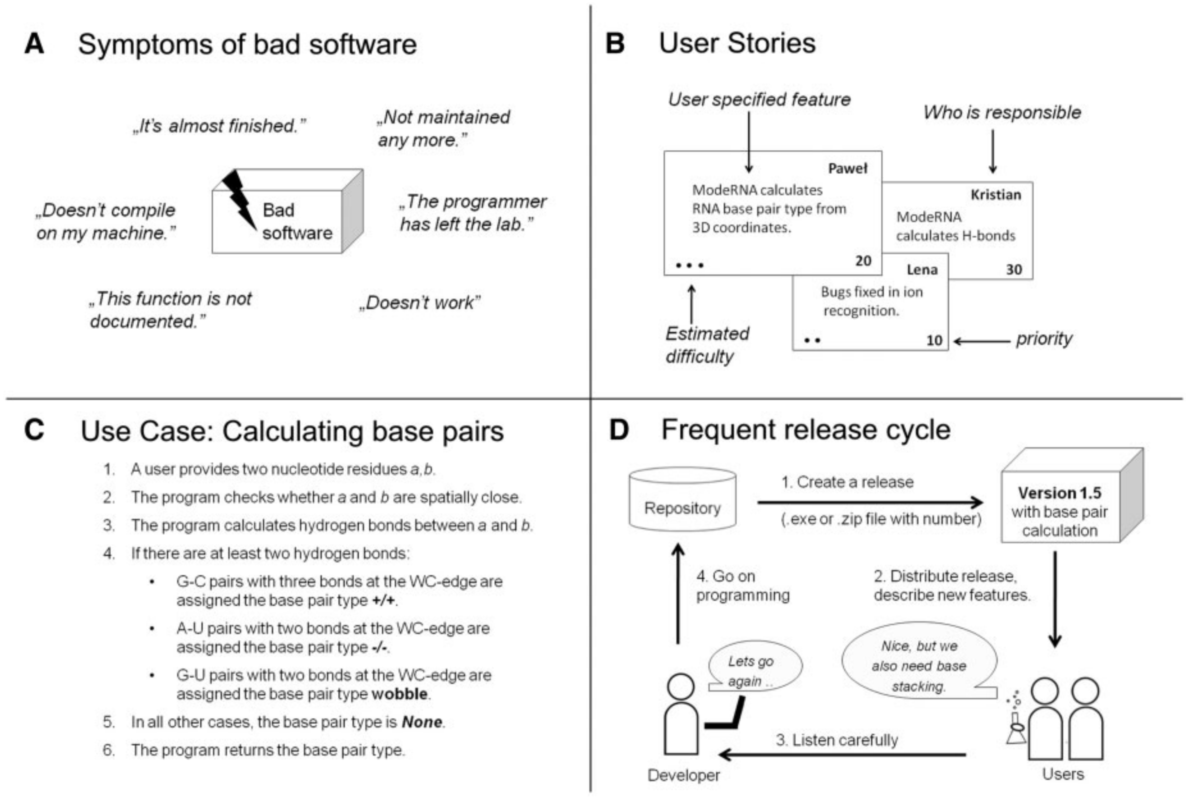
Wojciech Potrzebowski is a structural bioinformatician modeling macromolecular complexes at the IIMCB in Warsaw. He is developing algorithms working on 3D electron density maps.

Tomasz Puton is a structural bioinformatician working on procedures for RNA 2D structure prediction. He is the main developer of the software package for Statistical Geometry.

Magdalena Rother is the main developer of the ModeRNA software for RNA 3D structure modeling. She is also developing procedures for evaluating structural models.

Ewa Wywial is a structural bioinformatician working on homology modeling and phylogenetic analyses of protein structures.

Janusz M. Bujnicki is a research group leader of a combined theoretical/experimental laboratory. He has supervised numerous software development projects in structural bioinformatics of RNA and proteins.


A Symptoms of bad software
„It's almost finished."
„Not maintained any more."
„Doesn't compile on my machine."
Bad software
„The programmer has left the lab."
„This function is not documented."
„Doesn't work"
B User Stories
User specified feature
Who is responsible
Paweł
ModeRNA calculates RNA base pair type from 3D coordinates.
20
Kristian
ModeRNA calculates H-bonds
30
Lena
Bugs fixed in ion recognition.
10
Estimated difficulty
priority
C Use Case: Calculating base pairs
D Frequent release cycle
Repository
1. Create a release
(.exe or .zip file with number)
Version 1.5 with base pair calculation
4. Go on programming
2. Distribute release, describe new features.
Lets go again ..
Nice, but we also need base stacking.
3. Listen carefully
Developer
Users

1. A user provides two nucleotide residues *a*,*b*.
2. The program checks whether *a* and *b* are spatially close.
3. The program calculates hydrogen bonds between *a* and *b*.
4. If there are at least two hydrogen bonds:
    - G-C pairs with three bonds at the WC-edge are assigned the base pair type **+/+**.
    - A-U pairs with two bonds at the WC-edge are assigned the base pair type **-/-**.
    - G-U pairs with two bonds at the WC-edge are assigned the base pair type **wobble**.
5. In all other cases, the base pair type is ***None***.
6. The program returns the base pair type.

Figure 1: Problems and solutions in developing scientific software. (**A**) Typical statements indicating a unsystematic development approach. (**B**) User stories are paper cards that help to chop a programming project into smaller manageable tasks. (**C**) Use cases are a formal way of describing a step-by-step solution to a difficult problem. (**D**) A frequent release cycle provides users with incrementally improved versions of a program, allowing to react on user feedback quickly.

As an alternative, the Agile Manifesto has been developed, proposing a lighter software development approach [5]. In brief, it states that (i) working software is the most important, (ii) change is a constant in programming and plans need to be adjusted, (iii) face-to-face communication among developers and customers is crucial for success, and (iv) software can be developed incrementally, by creating working versions in short intervals. These principles have manifested itself in development models like Scrum [6], XP [7] and Crystal [8]. All Agile methods have in common that development practices are continuously adjusted to meet the demands of the team and project. A hands-on guide can be found at [9].

Agile methods have been successfully applied to develop biomedical software, e.g. InterPro [10] is being developed using Scrum. In an article by Kane *et al.* [11], there is a detailed analysis of the underlying processes and protocols. The authors conclude that an iterative approach is better suited for bioinformatics software development than a linear process.

However, recalling the initial problem of developing good software [2] (Figure 1A), these success stories are not representative. One reason is that adopting an Agile development model requires time dedicated to organizational matters. A number of best practices for scientific managers have been given by Baxter *et al.* [12], including valuable hints on project management, but telling little what can be done to improve a program. In practice, there is a plethora of techniques for planning, implementing and testing software, but literature is lacking that answers this question to researchers, students or group leaders without a background in computer science and software engineering.

Over the past 5 years, we needed to develop a series of scientific programs that were working, maintainable and easy to explain. Instead of adopting one methodology as a whole, we explored particular techniques one by one. In this article, we review 22 software projects and the engineering practices that we have been using.

BIOINFORMATICS SOFTWARE PROJECTS

Between 2006 and 2011, we have conducted a number of bioinformatic software projects. We have divided them into projects that have resulted in peer-reviewed publications (Table 1), and unpublished work (Table 2). The published work includes the Voronoia program for analyzing protein packing [13], the eMovie plugin for creating movies with PyMOL [14], the Protmap2D program for analyzing protein contact maps [15], the knotted2nested web server for removing RNA pseudoknots [16], the LaJolla program for structural alignment of RNA [17], the ModeRNA program for RNA 3D structure prediction [18], the Modomics database on RNA modifications [19], the REPAIRtoire database on DNA repair pathways [20], and the IGERS tool to for predicting DeltaG values of biological reactions [21]. In Table 1, we have listed the number of citations for these programs (including eventual earlier publications), although their number strongly depends on the focus of the project and publication date. The 12 unpublished projects in Table 2 include 5 projects where the software development has been largely or fully completed and manuscripts are currently being prepared for publication (I–V); three suspended projects (StatGeo, VI–VIII), which can be reactivated dependent upon funding availability and four additional (technically finished) projects (IX–XII), which, at present, are not expected to result in any follow-up work, because of project prioritization.

The 22 projects involve programming libraries, command-line tools, graphical applications, web databases, web servers (allowing submission of data and calculation), or a mixture of these. In total, 18 projects were implemented in Python and one in each Java, R and C++. Additionally, one project is a Python/Delphi hybrid. The duration of the projects was between one month and four years.

Table 1: Properties of published software projects

Project	Voronoia	Modomics	EMovie	ProtMap2D	K2N	LaJolla	IGERS	ModeRNA	REPAIR
Developers	**5**	12	4	2	4	5	5	**7**	9
Colocalized groups	**2/1/1/1**	6/4/1/1	3/1	2	2/1/1	3/1/1	4/1	**5/2**	5/2/1/1
Duration (months)	**24**	60	6	24	6	6	36	**42**	36
Releases	**3**	6	1	?	1	3	1	**8**	1
Language	**Del/Py**	Py	Py	Py	Py	Java	Py	**Py**	Py
License	**CS/PyL**	CS	GPL	CS	GPL	GPL	PyL	**GPL**	CS
Program library	○	●	○	○	●	○	●	●	○
Command-line app	●	○	●	○	○	●	●	●	○
GUI application	●	○	●	●	○	○	○	○	○
Web database	●	●	○	○	○	○	○	○	●
Web server	●	○	○	○	●	●	○	●	○
User Stories	○	●	○	○	○	○	○	●	●
Example data	●	○	○	●	●	●	●	●	○
CRC sheets	○	○	○	○	○	○	○	●	○
UML	○	●	○	○	○	○	○	●	●
Repository	○	●	○	●	●	●	●	●	●
Ticket system	○	●	○	●	○	○	○	○	●
Coding guidelines	○	○	○	○	●	○	○	○	○
Code reviews	○	●	●	○	●	○	○	●	●
Unit tests	○	●	○	○	●	●	●	●	○
TTD.	○	○	○	○	○	●	○	●	○
Cookbook	○	○	○	○	○	●	○	●	○
Release cycle	○	●	○	○	○	○	○	●	○
Completed	●	●	●	●	●	●	●	●	●
Actively developed	○	●	○	○	○	○	●	●	●
Would do it again	○	●	●	●	●	●	●	●	●
Reference	[13]	[19]	[14]	[15]	[16]	[17]	[21]	[18]	[20]
Published in	2003	2006	2007	2007	2007	2009	2010	2011	2011
PubMed citations	18	41	6	0	6	n.a.	0	0	0

REPAIR, REPAIRtoire database; Del, Delphi; Py, Python; CS, closed source; PyL, Python Licens; GPL, General Public License. 'Would do it again' refers to whether a majority of developers would like to work on a similar project in a similar way. The bold columns ('Voronoia' and 'ModeRNA') indicate two case studies discussed in detail in the text. The number of citations was extracted from PubMed, if available.

Table 2: Properties of unpublished software projects

Project	StatGeo	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII
Developers	4	4	7	4	4	5	26	4	4	4	2	3	2
Colocalized groups	3/1	3/1	3/3/1	3/1	2/2	4/1	20/6	2/2	4	2/1/1	2	2/1	2
Duration (months)	2	30	36	30	24	36	24	12	5	6	1	4	1
Releases	1	1	1	4	3	3	3	3	1	16	1	1	1
Language	Py	Py	Py	Py	Py	C++	Py	Py	Py	Py	Py	R	Py
License	GPL	CS	CS	CS	CS	CS	GPL	GPL	GPL	CS	GPL	CS	GPL
Program library	●	○	●	○	○	○	●	○	●	○	○	●	●
Command-line app	●	●	○	●	○	●	●	●	●	○	●	○	●
GUI application	○	○	○	○	●	●	○	○	○	●	○	○	○
Web database	○	○	○	○	○	○	○	○	○	○	○	○	○
Web server	○	●	●	●	○	○	○	○	○	○	○	○	○
User Stories	●	○	○	●	○	○	●	○	●	●	○	○	○
Example data	●	●	●	●	●	●	●	●	●	●	●	●	●
CRC sheets	○	○	○	○	○	○	○	●	○	○	○	○	○
UML	○	●	○	○	○	○	○	○	○	○	○	○	○
Repository	●	●	●	●	●	●	●	●	●	○	○	○	●
Ticket system	○	○	○	○	●	○	●	●	○	○	○	○	○
Coding guidelines	●	○	○	○	○	○	○	○	○	○	●	○	○
Code reviews	●	●	●	●	○	●	○	●	●	○	●	○	●
Unit tests	●	●	●	○	○	○	●	●	●	○	○	○	●
Test-driven development	●	●	●	○	○	○	○	○	○	○	○	○	●
Cookbook	○	○	○	○	○	○	●	○	○	○	○	○	○
Release cycle	○	○	○	○	●	●	●	○	○	●	○	○	○
Completed	●	○	●	●	●	○	○	○	●	●	●	●	●
Actively developed	○	●	●	●	●	●	○	○	○	○	○	○	○
Would do it again	●	○	●	●	●	●	○	○	●	○	○	○	●
Manuscript in prep	○	●	●	●	●	●	○	○	○	○	○	○	○

StatGeo, Statistical Geometry; Py, Python; R, R statistics package; CS, closed source; PyL, Python Licens; GPL, General Public License. 'Would do it again' refers to whether a majority of developers would like to work on a similar project in a similar way. The first column ('StatGeo') indicates the case study discussed in detail in the text. All projects except the first have been anonymized in order to protect unpublished work by the respective authors.

Around 2–12 people have been involved in each project, including cooperating experimentalists, other domain experts, undergraduate students and group leaders. Typically, some developers were more active than others, e.g. undergraduate students performing auxiliary tasks.

Below, we present three of the projects in detail. To highlight alternative outcomes, we have chosen one project that was scientifically successful but ran into engineering problems, one where we consider both the scientific and engineering part successful, and one that we consider well-engineered but was stopped for strategic reasons.

Case study 1: Voronoia—a tool for packing analysis

The Voronoia software is used to analyze packing densities and cavities in protein structures [13]. It consists of a core program for calculations written in Delphi, a graphical front-end written in Python, and a web interface created using Java. These three parts have been developed by one senior and two junior programmers in their favorite languages. The program itself [13] and several analyses using the program have been published [22–25], so it can be considered a scientific success.

Nevertheless, when we tried to adapt Voronoia to RNA structures, we ran into problems. For example, no developer had access to all of the source code, there were multiple gaps in the code documentation and no recent nor automated testing. Moreover, it was not certain which version of the source code included the most recent bugfixes and, finally, most of the initial developers had left the laboratory. These factors accumulated to a considerable impediment that made a further continuation of the project unlikely. This example prompted us to improve the way we write programs.

Case study 2: ModeRNA—a software for RNA comparative modeling

A few years later, our group developed the ModeRNA software for modeling of RNA 3D structures [18] based on the comparative modeling approach that has been successfully applied to proteins for many years. Our program was used to build more than 9000 tRNA models using template structures and target-template alignments. It includes a large library of structural fragments and can handle 115 different post-transcriptionally modified nucleotides. ModeRNA was written in Python and uses the PyCogent [26], BioPython [27] and NumPy libraries.

The implementation was conducted by two junior programmers coached by one senior programmer and supervised by a project leader. The junior programmers worked on the code independently, while the senior programmer concentrated on reviewing code, the architecture of the software, and writing tests. At the beginning, the project was decomposed into a set of elementary operations, e.g. 'add a methylation to a given nucleotide'. During the first half year, prototype scripts for individual tasks were written, based on the elementary tasks. Subsequently, a object-oriented architecture was created, using CRC sheets and Test-Driven Development (TDD). Tests were also added during debugging and adding further features. To date, ModeRNA contains 507 test functions, covering 90% of the overall code. In critical phases of development (e.g. debugging and refactoring stages, as well as for benchmarking), User Stories were collected and placed on a well-visible pin board to keep track of progress. Eight subsequent releases have been produced in two-month intervals. A scripting interface was designed, and all its functions documented online in a cookbook-style manual. ModeRNA is available as a programming library and executable at http://iimcb.genesilico.pl/moderna.

In our opinion the project is a major scientific and engineering success. A couple of follow-up publications have been written [[28], BiB (Rother M, Rother K, Puton T, Bujnicki JM, submitted for publication), Bioinformatics (Rother M, Milanowska K, Puton T, Jeleniewicz J, Rother K, Bujnicki JM, in press)]. Additionally, several lab members have started to customize ModeRNA functions for their own purposes or have joined the development team. The software reached a technical state where development can be taken over by another person with limited help from the original developers.

Case Study 3: a library for statistical geometry calculations

Our group has developed software in co-operation with The Centre for Integrative Bioinformatics at Vrije Universitet (IBIVU) in Amsterdam, implementing the statistical geometry in sequence space algorithm [29–31]. We wrote it as a replacement for the no-longer-maintained GEOMETRY program [32], and it was essential for projects being realized in the lab to have an extendable implementation. The program was implemented in the Python programming language. It intensively uses the PyCogent library [26], and the NumPy package for numerical calculations (http://numpy.scipy.org). The final implementation was validated with real-world data and cross-validated against the GEOMETRY program [32]. The entire implementation was conducted by one junior programmer with the supervision of two senior programmers—one of them in direct daily contact and the other in remote sporadic contact (i.e. checking the source code repository, making comments and minor changes in the source code). The TDD approach was used, i.e. tests were written prior to the code itself. The tests contain four times as many lines of code than the implementation itself and cover 76% of the code. Although the TDD approach initially required lots of time, it saved a large amount of debugging time and provided a solid base for effective code optimization. In fact, the final implementation of the algorithm was around 80 times faster than our first working version when evaluating a set of tRNA sequences.

The program was completed within 45 days. It reproduces data from the literature, and was used to conduct calculations for our own data sets. Although the project has not been published yet, we think it is successful from an engineering point of view: a reliable and transparent implementation has been created that can be used at a later time, and the entire project was brought to a quick and clean ending.

METHODS FOR DEVELOPING SOFTWARE

In the course of the 22 programming projects, we have applied a 'toolbox' of development techniques that helped us with planning our programming projects, improving software quality, delivering the outcome to users, and training junior programmers 'on the job'. Tables 1 and 2 summarizes which technique was used in which project. Below, 12 techniques for teams of fewer than 12 people are explained in detail.

User stories

When starting a programming project, the complexity of the task may at first seem overwhelming, especially for an inexperienced programmer. What helps is to divide the project into smaller tasks. To do this, we wrote down required features, also called User Stories (see Figure 1B). Each User Story describes something that has an added value for users; it should be brief enough to fit on a note pad (the card should not be bigger than $3'' \times 5''$—if the description does not fit one needs to start over again [9]). The description does not need to include technical details, but the list should be complete. User Stories that depend on many unknown factors should be minimally discussed. This way of documenting is not to be confused with writing up Use Cases, which are much more detailed text documents.

When using User Stories, we kept them permanently visible (e.g. on a pin board). Completed tasks were moved to a separate ‘done’ section, allowing to grasp progress immediately. User Stories have the advantage that they can be prepared in almost no time, and provide a ‘promise for conversation’ [9]—a starting point for further discussion about technical details, priorities, or responsibilities.

Collecting example data

When developing a verbal definition of program features into an implementation, it is essential to know exactly what kind of data the program will use. A practical approach is to gather explicit input files that the program can be run with. It is easier to use simplified examples for which the prospective output is known than big real-life data sets. Such example input/output pairs define the functionality from User Stories, task cards or tickets more accurately. This is a good occasion to decide on input/output formats of the program, and cooperation with potential users is very useful at this point. As the program grows, border cases like ambiguous input, big files or wrong data can be added to the examples. Having explicit input and output data helps to break down the problem into smaller parts and to design main functions of a program by developing intermediate input data for each of them. One can then describe detailed program steps between a given input/output pair as a Use Case (see Figure 1C) to break down a complex problem further. Eventually, example input/output data is a valuable base for creating automated tests (in particular acceptance tests, see section on testing below).

A prominent use of example data is the FR3D program for calculation of RNA base pairs, its website essentially being a showcase of exemplars [33]. Biopython [27] includes a set of files for each major component. In ModeRNA, many of the programs functions were first developed for the PDB structure 1EHZ that we almost knew by heart after a while. Also, we are using hundreds of example files covering as many situations as possible. In all of these scenarios, example data is a valuable resource to describe a program’s *functionality*, but not its *architecture*, which is covered in the next section.

Class-responsibility-collaboration cards

After dividing the functionality of a program into smaller units, the same can be done for a program’s architecture. In practice this means to define components (e.g. classes, modules, packages, etc.) and to assign responsibilities to them. One way to document the architecture as it develops is to write class-responsibility-collaboration (CRC) cards, which consist of a single page for each component. Every CRC card includes: the name of the component (i.e. its title) at the top of the sheet; on the left side the function of that component; and on the right side the names of other components, which the titled component depends on to perform its function.

We found CRC cards useful to define a few central components in order to write the first prototype and adjusting details later, rather than writing CRC cards for the entire software in advance. Also, we wrote CRC cards during cleanup stages to check whether for each User Story there is at least one responsible component and whether there are any duplicate responsibilities. Finally, one can verify whether the implementation is in accordance with the CRC cards and the cards should be updated accordingly, not the implementation [34].

Unified modelling language diagrams

The unified modelling language (UML) is a sophisticated instrument used in the software industry to formalize a technical system. It is capable of illustrating complex program subsystems graphically. Of these, we have used only Use Case and Class diagrams. Use Case diagrams represent ‘What a system does’, using actors, their goals (Use Cases), and dependencies between them. The diagram helps discussing a program outline among programmers or with the principal investigator. We have used Use

Case diagrams early in a project to explicitly check whether each User Story was represented in our draft of the system. Class diagrams represent classes, their attributes, methods and relationships between them. Similar to CRC sheets, they facilitate the discovery of circular dependencies and redundant responsibilities between program components. Apart from that, we have used Class diagrams in place of Entity–Relationship diagrams for database design, essentially because the Django framework used in the Modomics [19] and REPAIRtoire [20] projects implements the data model as Python classes. According to [35] these two components of the UML language are used the most, and Use Case diagrams scored highest in usefulness for communication with a client (e.g. when there is no program to show yet). We found that both kinds of diagrams can be explained easily to other bioinformaticians. Other elements of UML may be useful, but despite their benefits there is a risk of over-specifying a program via UML, termed as 'Death by UML Fever' [36].

Repositories

A code repository (also termed version control) manages files that are being used and modified by different persons. Programmers can use it to exchange program code and other files systematically. The code repository keeps track of which is the most recent version of a particular file and notices whenever two persons try to change the same version of a file and prevents changes that overwrite each other. Moreover, a project can be reverted to any earlier state. Multiple independent subprojects called branches can be managed simultaneously [37]. The files in the repository are not restricted to program code, but can include example input/output data, planning documents or related publications as well. Therefore, a code repository can be useful for any research project, even if no programming is involved.

To produce reproducible scientific results using software, it is required to test, protocol what has been done, and to recognize and track program bugs [12]. A repository is crucial for controlling exactly which version of a software, which set of input files, and which parameters were used to produce a specific set of results, and at what time. When a bug is found—especially if it is several years old—it is practically impossible to rebuild a previous state of the program without a repository.

Commonly used code repositories are Concurrent Versions System (CVS), Subversion (SVN) and more recently GIT and Mercurial. Space for publicly accessible repositories is frequently offered for free. A code repository also helps with data backup—instead of maintaining backups of the working copies of all users, it is sufficient to backup the repository.

Ticketing systems

Planning results in a number of well-defined tasks also referred to as tickets. To keep track of the progress in a project, the tickets can be displayed in a place visible for all developers. Ticketing systems allow programmers to assign priorities and the person in charge to each ticket. In the course of a project, further tickets can be created (e.g. bug reports). The main purpose of a ticketing system is to prevent important tasks from being forgotten, which becomes critical especially as the number of people involved in the development increases. A ticketing system can be as simple as a Wiki page or a pin board. There is a number of applications, for example TRAC or TestTrack that manage tickets electronically and automatically e-mail reports to assigned developers.

According to our experience, an online ticketing system is most useful when there is a large physical distance between developers. One of our projects was dominated by use of TRAC, which allowed the two main developers to maintain a well-structured collaboration over 2 years despite the fact that they met in person only twice. Another situation where tickets paid off was in the Modomics project [19], when feature requests and tasks for data curators accumulated, but for some time it was not clear who would take care of them.

Coding guidelines

To enhance the readability of source code and avoid common coding errors, a document with coding guidelines can be used. Such guidelines define a convention for naming variables, functions, classes and files, where and how to write comments and exclude certain language constructs. These constraints go beyond the mere syntactical rules of a programming language. Applied consequently, guidelines help keeping code readable for others (or yourself at a later date), because they prevent developers from overadjusting code to their personal preferences.

We have applied the PEP8 coding guidelines for Python, and the pylint tool that evaluates conformance of source code with PEP8 on a scale of up to 10 points. This introduces a competitive aspect that makes the otherwise boring task of cleaning up code more attractive. Another effect of the PEP8 guidelines is that they discourage using complex language constructs, keeping in mind a quote by Brian Kernigan: ‘Everyone knows that debugging is twice as hard as writing a program in the first place. So if you’re as clever as you can be when you write it, how will you ever debug it?’.

Code reviews

Typically, the level of experience in a programming team varies. We grouped junior and senior programmers working together in pairs. One technique that we have used in almost every such coach–trainee pair is internal review of the code. The junior programmers would implement a program. When the program approached a first working version, a senior colleague (or co-developer) would inspect the code and return it with comments written into the code itself. The review pointed out inefficient implementation, poor readability or structure and praise things done well. Cohen [38] gave helpful recommendations for code reviews. For instance, no more than 300–500 lines of code should be reviewed per hour. Comments written by the author of the code facilitate following the code during the review. For the review to be successful, a prompt and constructive reply is more important than an in-depth analysis of the entire code. On one hand, reviewing code written by junior level programmers can be an important component of their training. On the other hand, reviews by an experienced programmer for another may stimulate fruitful discussion and help solving difficult situations. Additionally, an external person might be able to provide more independent feedback than a teammate.

Write test code

Tests are necessary to check whether a given implementation is correct. Insufficient testing hampers development of many programs. According to Jacob Kaplan-Moss, one of the main authors of the successful Django web framework [39], ‘code without tests is broken by design’. Moreover, manual testing is prone to errors and the time needed to manually test the code of large programs usually exceeds the time needed to write it. The solution is to automate the tests.

There are various ways of testing code automatically and there exist dedicated testing utilities. Usually smaller units like functions or classes are tested on their own. This approach is known as unit testing. There are many unit-testing utilities in different programming languages (including JUnit in Java and the unit-test module in Python). Another way of testing is writing and running acceptance tests. These tests check whether a program as a whole works ‘as advertised’. Acceptance tests compare program runs on sample input and check whether the output meets certain criteria. Very often acceptance tests are based on User Stories and there can be many acceptance tests to ensure that the program is working as expected. Running test sets gives developers an objective measure of progress (i.e. how much of the program works). It is important to understand that both the program and the test code may contain bugs that cause an automatic test to fail. Fortunately, the test code is often not very complex, and it is therefore easier to assure that the test works correctly. Testing is not only limited to testing classes and functions—there exist various frameworks capable of testing user interfaces of both web and graphical applications. One of them is Selenium, which allows automatic testing of web applications. For testing GUI applications, one can use for example Abbot—a JAVA GUI testing framework. It is even possible to test random number generators by controlling seed values in the test environment. An extensive list of testing frameworks can be found at [40].

Tests are also crucial when a given implementation needs to be cleaned up (refactored) or its speed is to be optimized. They allow other developers to check the status of the code quickly and define the expected features of a program in the most precise way possible. Over time, automatic testing allows for faster development, because repeated manual testing of already existing features is replaced by the much faster automatic procedure (see Figure 2). Large test suites can be found for example in the ModeRNA program [18] and the PyCogent [26] and BioPython [27] libraries.

TDD

Intuitively, one would first implement a program and then test it. This design is called the ‘test-after’ approach [41]. The inverse method, when a programmer writes tests first and then the implementation, is known as TDD. In TDD a test function is

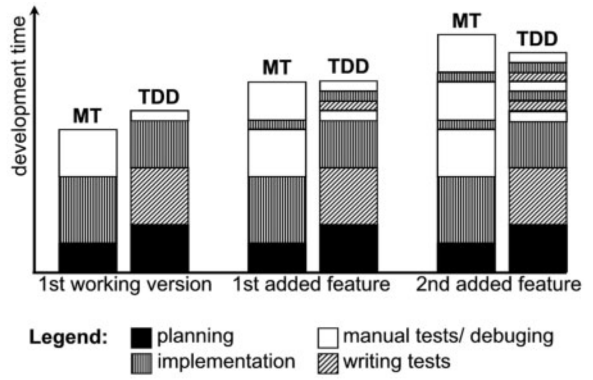


Figure 2: Saving time by writing tests. The left columns in each section indicate the rough relative amount of time spent in the manual testing/debugging approach (MT), right columns indicate the time using automatic tests or TDD.

written only for the part of the code that is going to be written next. Using TDD the design of an entire program can be guided in a divide-and-conquer schema (see Figure 3). When completed, tests usually contain sufficient information to start coding. If not, larger or more complicated parts should be divided into smaller units and additional tests should be written. An advantage of TDD is that it creates a strong motivating force for programmers; in order to write a test, they need to understand the entire problem. Since writing tests requires fully understanding the specification, developers are better prepared and more focused during the implementation [42]. It has been reported that TDD is slower, but produces more reliable code [43].

While TDD is understood mainly as a design approach, we found the concept of writing tests first valuable for debugging as well. Each time a bug was found, we added a test function to check whether the bug has been fixed. This way, the code could incrementally be made more reliable without having to be concerned about the same bug twice.

Releasing the program frequently

One of the most crucial steps is the delivery of a program to users. The trade-off between releasing a software early and deferral to enhance functionality or improve quality has been extensively studied [44]. Still, many imperfections are noticed only after the release, e.g. when real-world input data turns out to be more problematic than the examples used for testing.

The Agile Manifesto [5] recommends a simple yet effective solution: produce multiple incrementally improved versions of the program instead of trying to get everything right the first time (see Figure 1D). The releases of the program can be given to users allowing to collect feedback and include suggestions in subsequent releases. We have applied both regular release cycles of 2–12 weeks and irregular ones, where the release was made when the developers agreed it was time. Although building the release produced some overhead, we found it beneficial because it allowed to keep a project focused on goals instead of moving along a blind alley.

There are many configuration management tools like make or ant available that help with preparing releases, although some depend on the programming language, such as distutils (Python), or Maven (Java). This process can be automated by Continuous Integration tools that build a new release every time code changes are sent to a repository [45].

Cookbook documentation

A software cookbook is a collection of practical usage examples of a software library or scripting environment. Instead of lengthy descriptions, it contains code using the library that can be copied and executed directly. For instance, the secondary structure of a tRNA can be calculated by ModeRNA with the instructions:

```
from moderna import *
m = load_model('1QF6.pdb', 'B')
print get_secstruc(m)
```

We made many such examples readily available, and they consist the main part of the documentation. Cookbooks are used to document the Biopython [27], BioPerl [46], BioJava [47] and Pycogent libraries [26]. The latter uses Sphinx, a tool that can automatically test whether the code examples work in the intended way. As users of these libraries, the cookbooks allowed us to find out quickly what the software does and how to use it.

It needs to be noted that in order to avoid excessive code duplication in the examples, the program library needs a well-designed programming interface (API). Using design patterns, in particular the Façade pattern, helps to create an usable interface. As developers, we found creating and maintaining cookbook documentation less work than a full text manual.

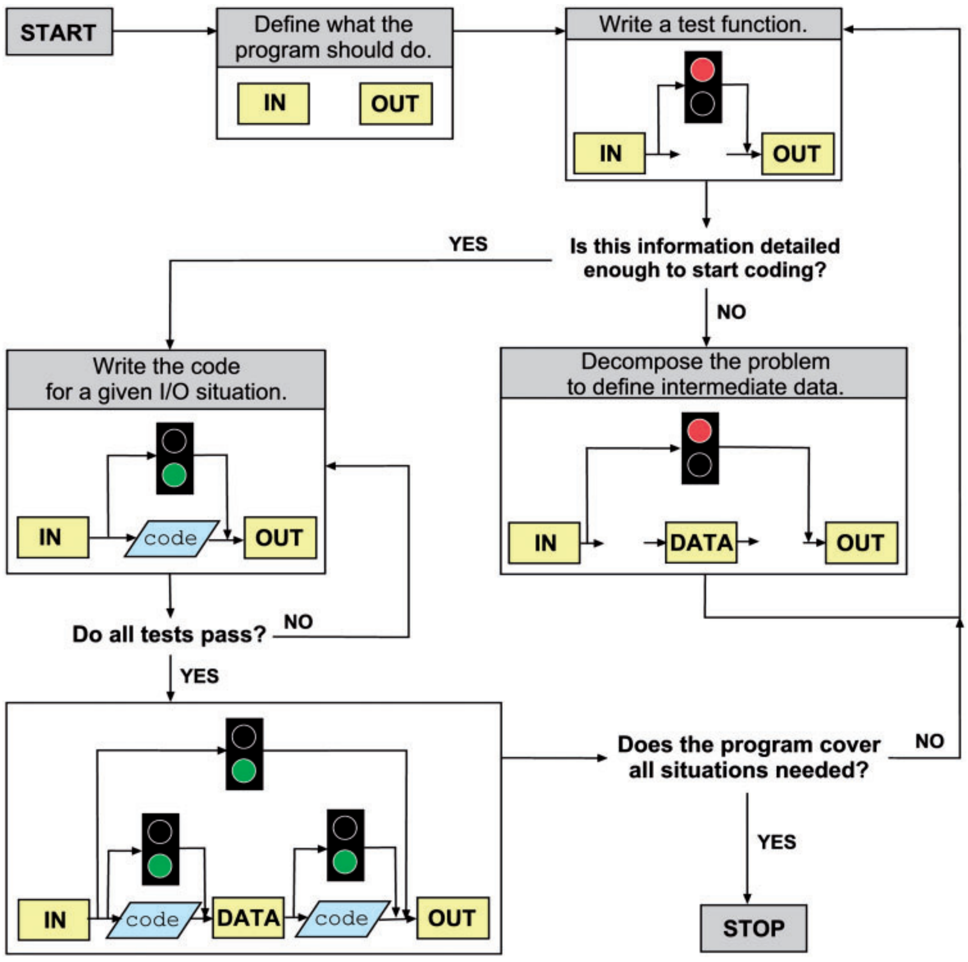

START
Define what the program should do.
IN
OUT
Write a test function.
IN
OUT
Is this information detailed enough to start coding?
YES
NO
Write the code for a given I/O situation.
IN
code
OUT
Decompose the problem to define intermediate data.
IN
DATA
OUT
Do all tests pass?
NO
YES
IN
code
DATA
code
OUT
Does the program cover all situations needed?
NO
YES
STOP

Figure 3: Schema of TDD. The idea of TDD is to write a program in a number of iterations, where test functions are written first. The tests set the frame in which each component can be developed independently. If a particular task is difficult, the problem should be divided, and new tests defined for the subcomponents. The charm of TDD is that at each step the test functions allow to monitor the process objectively.

Roles and responsibilities in a development team

It is crucial to recognize that good collaboration of the people involved is a prerequisite for success. Divergent points of view are important for creative work, but they can also lead to misconceptions, interpersonal friction and delays in the development. To avoid such troubles, we maintained in many projects a document that answered the questions ‘what is the project about?’ and ‘who is doing what?’. This project plan was kept either as wiki page, Google document or a file in the repository. An example project plan containing a brief project summary, and a list of tasks with initials of assigned persons is available as a supplementary file.

We found several roles appear repeatedly in different teams. The supervisor usually defined the scope of a programming project (features), decided on the product form (e.g. distribution, operating system, etc.) and managed the development team (see also Figure 4A). Involved users helped to communicate requirements to the programmers (e.g. as User stories), provided real-world data for testing, tested preliminary versions of the program and reported bugs (see also Figure 4B). Our observation was that user participation compels programmers to express themselves understandably that helped the project to stay on track. When one programmer was more experienced than the other, both often collaborated in a teacher–trainee setup (see Figure 4C). The coach programmer dissected tasks into smaller units for implementation, reviewed code, recommended algorithmic or architectural solutions and eventually wrote test functions.

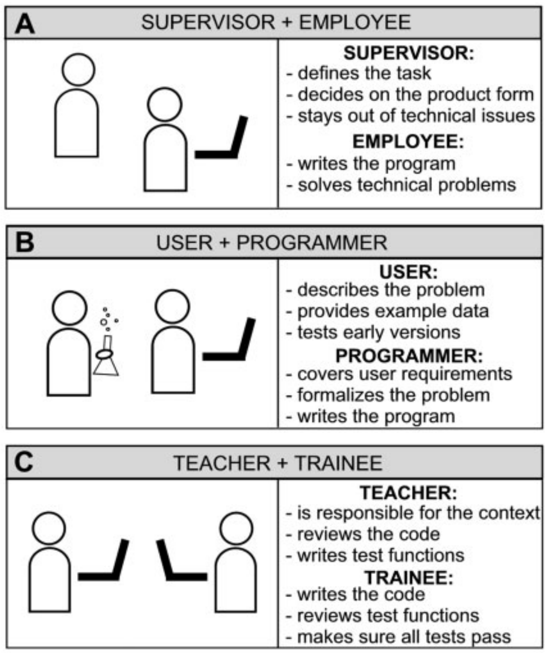


Figure 4: Responsibilities of people developing software in archetypical constellations. (**A**) supervisor/employee (**B**) user/programmer (**C**) programming in a teacher/trainee setup.

The trainee in turn was in full responsibility of the code. Often, the coach was also responsible to identify impediments affecting the project as a whole as well as for shaping constructive interaction of all team members. This function corresponds to that of Scrum Master in Scrum [6], while the supervisors role is also termed Product Owner in that context. In our experience, these roles served as a point of reference that helped team members orient themselves. We did not stick to them all the time, but it was clear when further communication was necessary, e.g. when the coach was about to edit a trainees' code.

DISCUSSION

In the 22 programming projects presented initially, we have applied 12 techniques. The projects in Table 1 have been successful in that they have resulted in peer-reviewed publications. Despite their differences in project duration, type of software and team size they had two things in common: first, the team composition with one or two core developers as in Case studies 1–3 is representative for all projects. Such a scenario has been described as a 'surgical programming team' [48], and we found it worked for 2–12 people involved. For larger teams, coordination quickly gets more complicated [8]. In fact, in one of our programming projects (VI from Table 2), temporarily 26 people were involved, but the number of active persons dropped quickly, and in the end progress stopped completely.

The second common trait in the 22 projects was that most of them were written in Python. Nevertheless, the methods described above can be applied with any programming language. Only when specific software tools are involved, one needs to identify their equivalents in ones' favorite language (e.g. Maven is a build tool that cooperates well with Java, while make is more common with C/C++ developers). One lesson we took from the Voronoia project is that using more than one programming language in a project can become a liability. However, there are examples where this is justified, namely applications such as PyMOL, which uses C for the time-critical parts and Python for the user interface [49].

Summarizing our observations, we think the techniques were most useful in three situations:

(1) When starting a project, formalizing tasks on cards, gathering example data or writing a short project summary helps developers to agree on a common set of ideas. This can prevent people running into different directions. CRC sheets and simple UML diagrams facilitate initial design decisions that shorten the path to the first working prototype. They leave however room for changes in the program architecture, which may be necessary as the project develops.
(2) During an ongoing project, consequent usage of a repository, automatic tests, frequent releases, code reviews or coding guidelines help maintaining high-quality code. Automatic tests rationalize time spent on testing the same things over and over. Repositories and releases eliminate the problem of finding the right program version or single files. Reviews and code analysis prevent the program from becoming a black box. Together, all these techniques propagate code that is more reliable, reusable and readable.
(3) When training bioinformaticians, a repertoire of development techniques can be used to give helpful and precise yet simple instructions. Advice like 'Could you add some tests to module X?', 'Does the class Y do the same as on the CRC sheet?' or 'Let's do a code review of Z' guides a student toward problem solving (and

keeps the smart ones from reinventing the wheel). For a supervisor, checking a repository or test set is a valuable tool to gain awareness of students' progress. Working in coach–trainee pairs (even if occasionally) is a key to teach good programming style.

Thus, should one try and include all described methods into the next best project at once? The data in Tables 1 and 2 do not contain a project that uses all methods. We are not convinced that one should generally try to use all tools available in the toolbox, because some of them may depend on personal preference. Also, changing too much in ones' development process at once may not be beneficial. We believe it is more reasonable to pick and introduce one method at a time that promises the biggest benefit in a given project. This allows to evaluate whether it works for ones project, and further changes can be added gradually, rather 'as much as necessary' than 'as many as possible'.

The same idea permeates the Agile school of thought. The evolutive nature of programming projects requires the techniques to co-evolve. One advantage of the techniques described here is that they can be used pragmatically as required, without knowing much about Agile. Nevertheless, we think that well-described methodological frameworks like Scrum and XP may allow for more efficient development than what we have written here. In fact, Scrum is being used by the developers of InterPro [10], but we have not tried it ourselves. One practice that we found particularly challenging to adapt is working in defined time-boxes (iterations). To familiarize with the concept of iterations, we recommend the Scrum Lego City game [50]. In this exercise, a team is to build a brick city within four time boxes of 5 min. Instructions on task cards are used, and each iteration is accompanied by a short planning and evaluation session.

Agile contains at least two more techniques that are worth mentioning but that we have not used extensively: one of them is pair programming, in which two developers share a single workstation to program. It has been reported that this helps to identify potential problems early and the resulting code is more reliable, and the overall productivity is not lower than without pair programming [8]. The other is daily Scrum or daily standup meetings. In a daily Scrum, the entire development team meets at a regular time to briefly discuss the status of the project. This helps to keep all developers informed and solve problems together.

A few important aspects of programming have not been covered here: development tools (editors, debuggers, testing frameworks, etc.) are mostly specific for a given programming language. A detailed discussion of Design Patterns [51] would exceed the scope of this text, although their importance for good program design cannot be ignored. Examples relevant for bioinformatics can be found in Biopython (Bio.PDB.Entity implements the Composite pattern), in ModeRNA (the API uses the Facade pattern), and in most web frameworks (the MCV pattern).

Another issue not touched is licensing. Barnes claims that it is imperative to make program code available in order to make its improvement possible [52]. On the other hand, there are arguments against, such as potential commercialization or cooperation with companies. We think the licensing was beneficial for the 12 open source projects in Tables 1 and 2, but the strategic decision about which license to use depends on many factors.

Finally, it would be worthwhile to extend the survey of development techniques to a larger body of bioinformatics projects and examine their long-term effects on software quality, availability and usage.

CONCLUSION

We have used software development techniques in 22 projects to support planning, improve code quality and help in training situations. The 12 methods described in this article follow a lightweight, pragmatic approach to writing software that is independent of the programming language or platform used. We therefore think this article may be useful as a toolbox to improve software production for other developers of bioinformatics software and team leaders with limited knowledge of software engineering.

SUPPLEMENTARY DATA

Supplementary data are available online at http://bib.oxfordjournals.org/.

FUNDING

Polish Ministry of Science and Higher Education (POIG.02.03.00-00-003/09, 188/N-DFG/2008/0);

the 7th Framework Programme of the European Commission (HEALTHPROT 229676); the German Academic Exchange Service (D/09/42768 to K.R.); the National Institutes of Health (1R01GM081680-01 to W.P.); the European Molecular Biology Organization (ASTF 274-2008 to T.P.); the Faculty of Biology at the Adam Mickiewicz University (PBWB-03/2009 to M.R.); the European Research Council (RNA + P = 123D to J.M.B.); and the Foundation for Polish Science ('Ideas for Poland' to J.M.B.].

Acknowledgements
We thank Paweł Skiba, Stanisław Dunin-Horkawicz, Grzegorz Papaj, Marcin Feder, Jan Kosiński, Łukasz Kozłowski, Andrzej Kamiński, Marcin Pawłowski, Jan Kogut, Michał Piętal, Maciej Fijałkowski, Michał Gajda, Tomasz Jarzynka, Ewa Tkalińska, Karolina Tkaczuk, Łukasz Kościński, Joanna M. Kasprzak, Kaja Milanowska, Katarzyna H. Kamińska, Tomasz Osiński, Marcin Domagalski, Jan Kaczyński, Małgorzata Figiel, Irina Tuszyńska, Raphael A. Bauer, Peter Hildebrand, Andrean Goede, Björn Grüning, Sabrina Hofmann, Sascha Bulik, Andreas Hoppe and Herrmann-Georg Holzhütter for their participation in the programming, Coding Sprints and testing software. We would also like to acknowledge Sandra Smit, Rob Knight and Gavin Huttley for their support in the Statistical Geometry project, and fruitful discussions about development.

References

1. Hannay JE, MacLeod C, Singer J, *et al.* How do scientists develop and use scientific software, *ICSE Workshop on Software Engineering for Computational Science and Engineering*, 2009, Vancouver, Canada, pp. 1–8.
2. Merali Z. Computational science: . . . Error. *Nature* 2010; **467**:775–777.
3. Benington HD. Production of large computer programs. *IEEE Annals of the History of Computing* 1983;**5**:12.
4. Poksinska B, Dahlgaard JJ, Antoni M. The state of ISO 9000 certification: a study of Swedish organizations. *TQM Magazine* 2002;**14**:10.
5. Beck K, Beedle M, Bennekum A, *et al.* Manifesto for Agile Software Development. *Agile Alliance* 2001.
6. Takeuchi H, Nonaka I. The new new product development game. *Harv Bus Rev* 1986. Jan 01. Online journal.
7. Beck K. *Extreme Programming Explained: Embrace Change*. Boston: Addison-Wesley Professional, 1999.
8. Cockburn A. *Agile Software Development*. Boston: Addison-Wesley Professional, 2001.
9. Amfahr J, Bustamante A, Rome P. Exploring Agile: The Seapine Agile Expedition. http://downloads.seapine.com/pub/ebooks/SeapineAgileExpedition.pdf (7 May 2011, date last accessed).
10. Hunter S, Apweiler R, Attwood TK, *et al.* InterPro: the integrative protein signature database. *Nucleic Acids Res* 2009;**37**:D211–5.
11. Kane DW, Hohman MM, Cerami EG, *et al.* Agile methods in biomedical software development: a multi-site experience report. *BMC Bioinformatics* 2006;**7**:273.
12. Baxter SM, Day SW, Fetrow JS, *et al.* Scientific software development is not an oxymoron. *PLoS Comput Biol* 2006;**2**: e87.
13. Rother K, Hildebrand PW, Goede A, *et al.* Voronoia: analyzing packing in protein structures. *Nucleic Acids Res* 2009;**37**:D393–5.
14. Hodis E, Schreiber G, Rother K, *et al.* eMovie: a storyboard-based tool for making molecular movies. *Trends Biochem Sci* 2007;**32**:199–204.
15. Pietal MJ, Tuszynska I, Bujnicki JM. PROTMAP2D: visualization, comparison, and analysis of 2D maps of protein structure. *Bioinformatics* 2007;**23**:1429–30.
16. Smit S, Rother K, Heringa J, *et al.* From knotted to nested RNA structures: a variety of computational methods for pseudoknot removal. *RNA* 2008;**14**:410–416.
17. Bauer RA, Rother K, Moor P, *et al.* Fast structural alignment of biomolecules using a hash table, N-grams and string descriptors. *Algorithms* 2009;**2**:17.
18. Rother M, Rother K, Puton T, *et al.* ModeRNA: a tool for comparative modeling of RNA 3D structure. *Nucleic Acids Res* 2011;**39**:4007–22.
19. Czerwoniec A, Dunin-Horkawicz S, Purta E, *et al.* MODOMICS: a database of RNA modification pathways. 2008 update. *Nucleic Acids Res* 2009;**37**:D118–21.
20. Milanowska K, Krwawicz J, Papaj G, *et al.* REPAIRtoire–a database of DNA repair pathways. *Nucleic Acids Res* 2011;**39**: D788–92.
21. Rother K, Hoffmann S, Bulik S, *et al.* IGERS: inferring Gibbs energy changes of biochemical reactions from reaction similarities. *Biophys J* 2010;**98**:2478–86.
22. Rother K, Preissner R, Goede A, *et al.* Inhomogeneous molecular density: reference packing densities and distribution of cavities within proteins. *Bioinformatics* 2003;**19**: 2112–21.
23. Hildebrand PW, Rother K, Goede A, *et al.* Molecular packing and packing defects in helical membrane proteins. *Biophys J* 2005;**88**:1970–77.
24. Hildebrand PW, Gunther S, Goede A, *et al.* Hydrogen-bonding and packing features of membrane proteins: functional implications. *Biophys J* 2008;**94**:1945–53.
25. Elshemey WM, Elfiky AA, Gawad WA. Correlation to protein conformation of wide-angle X-ray scatter parameters. *Protein J* 2010;**29**:545–50.
26. Knight R, Maxwell P, Birmingham A, *et al.* PyCogent: a toolkit for making sense from sequence. *Genome Biol* 2007; **8**:R171.
27. Cock PJ, Antao T, Chang JT, *et al.* Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics* 2009;**25**:1422–3.
28. Rother K, Rother M, Boniecki M, *et al.* RNA and protein 3D structure modeling: similarities and differences. *J Mol Model* 2011. Jan 22 [Epub ahead of print].
29. Eigen M, Winkler-Oswatitsch R, Dress A. Statistical geometry in sequence space: a method of quantitative comparative sequence analysis. *Proc Natl Acad Sci USA* 1988;**85**:5913–7.
30. Eigen M, Lindemann BF, Tietze M, *et al.* How old is the genetic code? Statistical geometry of tRNA provides an answer. *Science* 1989;**244**:673–9.

31. Nieselt-Struwe K. Graphs in sequence spaces: a review of statistical geometry. *Biophys Chem* 1997;**66**:111–31.
32. Kuznetsov I, Morozov P. GEOMETRY: a software package for nucleotide sequence analysis using statistical geometry in sequence space. *Comput Appl Biosci* 1996;**12**:297–301.
33. Sarver M, Zirbel CL, Stombaugh J, *et al*. FR3D: finding local and composite recurrent structural motifs in RNA 3D structures. *J Math Biol* 2008;**56**:215–52.
34. Wilkinson NM. *Using CRC Cards: An Informal Approach to Object-Oriented Development*. Cambridge, UK: Cambridge University Press, 1998.
35. Larman C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Upper Saddle River, JJ, USA: Prentice Hall PTR, 2001.
36. Bell AE. Death by UML fever. *ACM Queue* 2004;**2**:9.
37. O'Sullivan B. Making sense of revision-control systems. *Commun ACM* 2009;**52**:7.
38. Cohen J. Best Kept Secrets of Peer Code Review. Smartbearssoftware.com, 2006.
39. Foundation DS. Django Framework. www.djangoproject.com (10 May 2011, date last accessed).
40. Testing Framework. http://c2.com/cgi/wiki?TestingFramework (10 May 2011, date last accessed).
41. Erdogmus H, Morisio M, Torchianp M. On the effectiveness of the test-first approach to programming. *Software Engineering, IEEE Transactions* 2005;**31**:12.
42. Beck K. *Test Driven Development: By Example*. Boston: Addison-Wesley Professional, 2002.
43. Müller MM, Padberg F. About the Return on Investment on Test-Driven Development. Universität Karlsruhe, Germany.
44. Sassenburg H. Design of a Methodology to Support Software Release Decisions. Do the Numbers Really Matter?. SE-CURE AG 2005. http://dissertations.ub.rug.nl/faculties/eco/2006/j.a.sassenburg/ (13 June 2011, date last accessed).
45. Humble J. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Boston: Addison-Wesley Professional, 2010.
46. Stajich JE, Block D, Boulez K, *et al*. The Bioperl toolkit: Perl modules for the life sciences. *Genome Res* 2002;**12**:1611–8.
47. Holland RC, Down TA, Pocock M, *et al*. BioJava: an open-source framework for bioinformatics. *Bioinformatics* 2008;**24**:2096–7.
48. Brooks FP. *The Mythical Man-Month: Essays on Software Engineering*. Boston: Addison-Wesley Professional, 1995.
49. DeLano WL. The PyMOL Molecular Graphics System 2002.
50. agile42. Scrum Lego City. http://www.agile42.com/cms/pages/lego_city_game/ (06 May 2011, date last accessed).
51. Gamma E, Helm R, Johnson R, *et al*. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley Professional, 1994.
52. Barnes N. Publish your computer code: it is good enough. *Nature* 2010;**467**:753.