Chaste: using agile programming techniques to
develop computational biology software

BY JOE PITT-FRANCIS1,*, MIGUEL O. BERNABEU1, JONATHAN COOPER1,
ALAN GARNY2, LEE MOMTAHAN1, JAMES OSBORNE1,
PRAS PATHMANATHAN1, BLANCA RODRIGUEZ1, JONATHAN P. WHITELEY1
AND DAVID J. GAVAGHAN1

1Oxford University Computing Laboratory, Wolfson Building,
University of Oxford, Parks Road, Oxford OX1 3QD, UK

2Department of Physiology, Anatomy and Genetics, Sherrington Building,
University of Oxford, Parks Road, Oxford OX1 3PT, UK

Cardiac modelling is the area of physiome modelling where the available simulation
software is perhaps most mature, and it therefore provides an excellent starting point for
considering the software requirements for the wider physiome community. In this paper,
we will begin by introducing some of the most advanced existing software packages for
simulating cardiac electrical activity. We consider the software development methods
used in producing codes of this type, and discuss their use of numerical algorithms,
relative computational efficiency, usability, robustness and extensibility.

We then go on to describe a class of software development methodologies known as test-
driven agile methods and argue that such methods are more suitable for scientific software
development than the traditional academic approaches. A case study is a project of our
own, Cancer, Heart and Soft Tissue Environment, which is a library of computational
biology software that began as an experiment in the use of agile programming methods.
We present our experiences with a review of our progress thus far, focusing on the
advantages and disadvantages of this new approach compared with the development
methods used in some existing packages.

We conclude by considering whether the likely wider needs of the cardiac modelling
community are currently being met and suggest that, in order to respond effectively to
changing requirements, it is essential that these codes should be more malleable. Such
codes will allow for reliable extensions to include both detailed mathematical models—of
the heart and other organs—and more efficient numerical techniques that are currently
being developed by many research groups worldwide.

Keywords: cardiac electrophysiological modelling; high-performance computing;
software engineering

* Author for correspondence (joe.pitt-francis@comlab.ox.ac.uk).

One contribution of 12 to a Theme Issue 'The virtual physiological human: building a framework
for computational biomedicine'.

3111 This journal is © 2008 The Royal Society
Mathematical modelling of cardiac physiology, and, more particularly, cardiac electrophysiology, is a very mature discipline with the first model of a cardiac cell action potential having been developed in 1962 (Noble 1962). Current models of cardiac electrophysiology span the range from models of single ion channels (for instance, those of Capener et al. 2002), through very complex models of individual cardiac cells (see Rudy & Silva (2006) for a review), to models of the electrical activity in geometrically and anatomically detailed whole ventricles (see review by Kerckhoffs et al. 2006).

Several advanced software packages for modelling cardiac electrophysiology have been developed by different groups around the world. These packages include CARP (http://carp.meduni-graz.at/), CMISS (http://www.cmiss.org/), CONTINUITY (http://www.continuity.ucsd.edu/), MEMFEM and SCAM among others. CARP is a finite-element solver of the cardiac bidomain equations (see §2), which has been developed by researchers at the University of Calgary and the Graz University, and is the most mature code in terms of linear solvers and parallel efficiency on distributed-memory clusters. CMISS is a long-standing general simulation tool developed in the Bioengineering Institute at the University of Auckland. CONTINUITY is another general multiphysics simulation package produced by the University of California San Diego, which is able to run on a wide variety of computer architectures and operating systems. MEMFEM is a package that solves the cardiac bidomain equations using finite-element methods and is now maintained at the Johns Hopkins University. SCAM is a monodomain simulation package for studying re-entry and fibrillation developed at the University of Sheffield.

These codes have achieved sufficient efficiency to allow simulation of cardiac activity using large anatomically based atrial and ventricular meshes (Nielsen et al. 1991; Vetter & McCulloch 1998). Through the use of these software packages, a large number of computational studies have provided significant insight into cardiac electrophysiological behaviour in health and disease, including investigations on ventricular and atrial arrhythmias and anti-arrhythmia therapy among other applications. Such studies include Fenton et al. (2005), Jacquemet et al. (2005), Rodríguez et al. (2005, 2006), Tranquillo et al. (2005), Potse et al. (2006), Seemann et al. (2006), ten Tusscher et al. (2007) and Vigmond et al. (2008).

Although these packages have proven critical to our understanding of cardiac behaviour, they share one or more of the following limitations: (i) they are not generic—they have been developed for specific applications based on the particular scientific interests of the developers, (ii) they have not been developed using state-of-the-art software engineering methods and therefore have not been completely tested and validated or, at least, results are not publicly available, (iii) they do not achieve maximum efficiency on high-performance computing platforms because they have not been designed with such platforms in mind, (iv) they are not freely available to the scientific community but only to direct collaborators, and (v) they do not include state-of-the-art numerical and computational techniques.

It is probable that higher quality software (which exhibits fewer of the limitations listed above) may be written by adopting professional software engineering practices. Academic software is usually written by a small and fluid
workforce who inherit source code without necessarily fully understanding how it works. A new doctoral student (or postdoctoral researcher) will typically spend some of his or her time learning about the current state of the software from an established student (who is simultaneously writing up his or her thesis). Similar problems in the commercial sector have spawned a plethora of programming paradigms, and so, after struggling with this type of development for several years, we decided to investigate whether one class of these paradigms, namely agile methods, was appropriate for software development in the academic environment.

In order to test this hypothesis, we embarked upon the Chaste project. The idea behind Chaste was to bring together a large number of academic developers from a range of backgrounds in order to collectively produce a library of code using agile methods, using the wide range of skills offered by these workers. Chaste stands for ‘Cancer, Heart and Soft Tissue Environment’ and is largely maintained in two problem domains: cardiac electrophysiology and cancer modelling. The code is being developed with and by researchers from both areas with as much code modularity and reuse as possible.

This paper documents our experiences with agile methods in the Chaste project, and thus presents both a new application area for agile methods and a potential new development methodology for computational biology software. Throughout this paper, particular emphasis is placed on the cardiac modelling thread since modelling in this area is most mature. In the next section, we outline the main mathematical models used in the cardiac simulation and then explain in §3 that there are particular issues that need to be tackled when developing software for this field. The following section (§4) describes the agile method (as used in the commercial sector) and the adaptations that we have made in order to use it in academia. Section 5 gives an indication of the current status of the Chaste project and evaluates our experience of using agile techniques. Finally, we conclude with some more general discussion on the merits of using agile techniques in this problem field in §6.

2. Introduction to heart modelling

In order to lay out the main problems faced when developing software for computational biology in §3, we introduce the main problem domain of interest: cardiac electrophysiology.

In modelling at the tissue-to-whole-organ level, the most commonly used models of cardiac electrical activity are the monodomain equations or the bidomain equations. These equations model in detail the manner in which electrical activity spreads through the heart as a wave. The bidomain equations take the form of a coupled system of partial differential equations (PDEs) and ordinary differential equations (ODEs) given by

\[
\chi \left( C_m \frac{\partial V_m}{\partial t} + I_{\text{on}}(V_m, u) \right) - \nabla \cdot (\sigma_i \nabla (V_m + \phi_e)) = 0, \tag{2.1}
\]

\[
\nabla \cdot ((\sigma_i + \sigma_e) \nabla \phi_e + \sigma_i \nabla V_m) = 0, \tag{2.2}
\]

\[
\frac{\partial u}{\partial t} = f(u, V_m), \tag{2.3}
\]

*Phil. Trans. R. Soc. A* (2008)
where $V_m$ is the transmembrane potential and the primary variable of interest; $\phi_e$ is the extracellular potential; $u$ is a vector of dependent variables containing gating variables and ionic concentrations; $\chi$ is the surface-to-volume ratio; $C_m$ is the membrane capacitance per unit area; $\sigma_i$ is the intracellular conductivity tensor; $\sigma_e$ is the extracellular conductivity tensor; $I_{\text{ion}}$ is the total ionic current; and $f$ is a prescribed vector-valued function. Both $I_{\text{ion}}$ and $f$ are determined from a cellular ionic model that typically takes the form of a large (e.g. several tens of equations in ten Tusscher et al. 2004) system of nonlinear ODEs modelling the movement of ions across the cell membrane (see the CellML website (http://www.cellml.org/) for examples of this type of model). Suitable boundary conditions for equations (2.1) and (2.2) are

$$(\sigma_i \nabla (V_m + \phi_e)) \cdot n = I_{\text{i}};$$

$$(\sigma_e \nabla \phi_e) \cdot n = I_{\text{e}},$$

where $I_{\text{i}}$ is the external stimulus applied to the intracellular boundary per unit area; $I_{\text{e}}$ is the external stimulus applied to the extracellular boundary per unit area; and $n$ is the outward-pointing normal vector to the boundary. Equation (2.3) does not need any boundary conditions as there are no spatial derivatives appearing in this equation. Finally, we need to specify initial conditions for the dependent variables $V_m$, $\phi_e$ and $u$. When a bath is modelled, it is treated as an extension of the interstitial fluid (Vigmond et al. 2008).

Under certain conditions, we may eliminate $\phi_e$ from the bidomain equations, and equations (2.1) and (2.2) may be replaced by a single equation

$$\chi \left( C_m \frac{\partial V_m}{\partial t} + I_{\text{ion}}(V_m, u) \right) - \nabla \cdot (\sigma \nabla V_m) = 0,$$

where $\sigma$ is a conductivity tensor. Equations (2.3) and (2.6) are collectively known as the monodomain equations. A suitable boundary condition for equation (2.6) is

$$(\sigma \nabla V_m) \cdot n = I_{\text{s}},$$

where $I_{\text{s}}$ is the applied current per unit area on the boundary.

As the heart has an irregular geometry, the equations with spatial derivatives in the monodomain and bidomain equations are usually solved using the finite-element method. One advantage of this method is the ease with which irregular geometries, and the consequent unstructured computational grids, are handled. Furthermore, the derivative boundary conditions—equations (2.4), (2.5) and (2.7)—may be incorporated in a systematic, natural manner. To the best of our knowledge, all of the software packages named in §1 use a fixed mesh when solving either the monodomain or bidomain equations, and approximate the time derivatives appearing in the governing equations using finite-difference methods.

The finite-element method may be used to reduce the PDEs at each time step (equations (2.1), (2.2) or (2.6)) to a large, sparse linear system (Reddy 1993). This system $Ax = b$ has to be solved at each time step, where $x$ is the vector of the unknown potentials at each node. The matrix $A$ is constant in time and needs to be constructed only once. A finite-element simulation of cardiac
electrophysiology therefore loosely breaks down into three tasks to be performed at each time step: integrate the cellular ODEs; assemble the vector $b$; and solve the resulting linear system. The choice of matrix preconditioner and linear solver used to solve the linear system is crucial, as is the choice of ODE solver.

3. Computational and software development issues in cardiac modelling

There are several issues that arise when high-performance computational models that use the monodomain or bidomain equations are developed in academia. These are associated with both the problem domain—cardiac modelling—and the organizational context—academia.

(a) Computational issues

The electrical activity of the heart, as described in §2, is a multiscale process in both space and time. The action potential that is described by the monodomain and bidomain equations usually propagates across tissue with a steep wavefront. In order to accurately resolve this wavefront, a fine computational mesh must be used (a typical spatial resolution is approx. 100–200 $\mu$m). Furthermore, the ODEs given by equation (2.3) model an increasingly large number of physiological processes that vary on a wide range of time scales: to accurately resolve the effect of the fast physiological processes, a short time step must be used—a typical choice of time step being of the order of $10^{-2}$ ms. If an explicit numerical method should be used for solving these ODEs, then there may be issues with the numerical stability of the numerical scheme, particularly as the multiscale nature of the physiological processes implies that the ODEs are stiff (Iserles 1996). These stability issues may be prevented by using a low-order implicit ODE solver, but at the expense of both coding and computing a Jacobian of the resulting nonlinear system.

Computing the electrical activity of one or more beats of the entire human heart is therefore an enormously expensive computational procedure. Meshes with an extremely large number of nodes (of the order of $10^6$–$10^8$) are required to account for the full structure of the heart while having the spatial resolution to capture large gradients and small-scale processes. Also, solutions for a very large number of time steps need to be computed. Therefore, of the order of $10^5$ iterations in time, and correspondingly $10^5$ finite-element solution steps, must be made to obtain the electrical activity over the course of a heartbeat. Potse et al. (2006) stated that it takes 2 days to simulate a single human heartbeat using the bidomain model on a 32 processor machine (albeit at a fairly coarse spatial scale).

The significant computational challenge associated with cardiac modelling means that cardiac software must be highly efficient in its implementation, in terms of both software design and numerical algorithms. Improved numerical methods can alleviate some (or much) of the computational strain, and recent research has focused on developing numerical schemes that improve the efficiency of the solution of both the bidomain and monodomain equations (Quan et al. 1998; Qu & Garfinkel 1999; Skouibine et al. 2000; Sundnes et al. 2001, 2005; Vigmond et al. 2002, 2008; Cherry et al. 2003; Pennacchio 2004; Weber dos Santos et al. 2004; Colli Franzone et al. 2005; Whiteley 2006, 2007).
efficient implementation and algorithms, it is a vital feature of cardiac software that it should run in parallel with good parallel speedup. Previous generation specialist shared-memory supercomputers allowed parallel efficiency to be obtained with shared-memory paradigms such as OPENMP. Nowadays, distributed-memory clusters made from commodity components are ubiquitous. Since most users now have access only to high-performance computing via distributed-memory clusters, it is essential that developers target this architecture. However, programming for efficient load balancing and scaling on such clusters adds an extra complication to the task of the programmer and may require that developers rethink their software designs.

(b) Software development issues

Section 3a illustrated how efficiency is a crucial feature in cardiac software. In order for future cardiac modelling packages to meet the needs of the user community, they further need to be not only robust but also extensible and reliable. The extensible nature of software is particularly important since physiologists may require multiphysics (and multi-organ) simulations and mathematicians may want to adopt innovative numerical techniques. That is, code not only needs to be efficient but also needs to be: (i) robust, (ii) extensible, and (iii) perhaps most importantly reliable, through having been fully tested.

The software development paradigm chosen is an important factor in the production of high-performance cardiac software satisfying these multiple and sometimes conflicting requirements. However, the fluid nature of the academic workforce does not easily facilitate good software engineering practices. A typical academic scenario is that of a single person writing code for his or her own research, which is understood only by that person. After the lifetime of the researcher’s position in that department, it becomes unused or, at best, a ‘black-box’ that is used without modification or maintenance, since no one understands its workings. Larger, and longer running, software development programmes can involve teams of postgraduate and postdoctoral researchers over periods of several years, but require a continuous stream of funding. The major problem here is that of maintenance. If the developers use their own esoteric coding styles or do not document their code, or if only a small number of developers work on one particular aspect of the code and then all eventually leave the team, their contributions become opaque to future generations of developers. Architectural decisions made in the past become difficult to change as the coding paradigm becomes embedded into all aspects of later contributions. This makes it harder for a developer to modify the code and thus makes it harder to add functionality. Ideally, cardiac codes, in common with all software, should be written in such a way that they can be relatively easily understood by a new member of the workforce, as otherwise the software will have a limited shelf-life and continual redevelopment will be impossible. This is a particular problem in academia, where the short-term contracts offered to research staff lead to a high turnover of the workforce. In the academic environment, it is therefore imperative that code is well documented, and that the understanding of the precise workings of any one component of the code is not restricted to a single individual.

It is clear that the standard academic software development approach described above is not the most appropriate paradigm, since it produces code that is not easily understood by other researchers and not easily extended.
An alternative is to use modern ‘AGILE’ software development techniques. We describe these methods in §4, and show in §5 how we are using agile programming ideas within the Chaste project to produce efficient, tested, extensible and maintainable code.

4. Agile programming and the Chaste project

It was explained in §3 that there are computational and software issues associated with developing software to model cardiac electrophysiology. We have explained the limitations of the traditional academic approach to software development, but a move to a bureaucratic process would probably clash with the organizational culture in academia. Furthermore, requirements for a project are prone to change as new technologies emerge and the underlying science moves forward. We therefore introduce agile methods (a set of lightweight software development techniques such as test-driven development and pair programming) and show that these techniques offer a valuable route to tackling some of these issues. Agile methods are suitable as they provide flexibility through a close relationship between the ‘customer’ and developer through frequent iterations.

When we embarked upon Chaste, we realized that there was a good fit between our own needs and agile methods. Therefore, as an experiment, we decided to apply agile techniques to a four-week training course. Following the success of that course, we expanded the experiment: we adapted and augmented the code written during the course into a set of maintainable libraries using agile methods over the following 3 years.

(a) Introduction to agile programming

In the previous section (§3), we looked at the traditional academic approach to software development where software is written without much of an underlying plan. Here we describe agile methods and briefly contrast these to plan-driven methodologies. The specific standard agile methodology we describe is called ‘eXtreme programming’ (XP; Beck & Andres 2004).

The original response to the issue of scaling software production was the introduction of the notion of methodology. Methodologies, inspired by traditional engineering disciplines, were designed to impose a rigorous process on software development, with the aim of making it predictable and more efficient. These methodologies separate the writing of software into design and coding, where the result of design is a plan that is supposed to make coding a predictable activity. In practice, however, plan-driven methodologies are not noted for being successful, require a lot of documentation and are frequently criticized as being too bureaucratic. There is so much to do to follow the methodology that the pace of development slows down (Fowler 2000a).

While such a separation between design and construction is valid for traditional engineering—where a design can be made, which can be checked against engineering rules and turn construction into a predictable activity—this is not so in software development. In traditional engineering, the cost of construction is much higher than the cost of design and so this is worthwhile. The same is not true in software engineering where the design is neither cheap nor static.
Agile methods can be seen as a reaction to plan-driven methodologies. They try to steer a middle ground between no methodology and bureaucratic engineering-inspired methodologies, by providing just enough methodology to get the task done. Rather than attempting to make software development a predictable activity, they take a different strategy and try to adapt to changing circumstances.

The key to this adaptive approach is feedback. This is achieved through incremental and iterative software development. Fully tested and working versions of software are produced at frequent intervals. So, for example, a plan-driven approach would call for considerable resources to be spent on researching all requirements, culminating in a detailed requirements document, before design and coding can start. The reason for this is that, if inaccurate or incomplete requirements are discovered after the design and coding stages are complete, these stages would need to be redone at great expense. By contrast, the agile approach calls for implementing the top priority requirements as quickly as possible and leaving consideration of the lower priority requirements until later. The logic of agile methods is that, by implementing high-priority requirements into a working and usable system quickly, any problems with requirements will soon be exposed, before too much cost is incurred. The path from requirements to working code is made very short with agile methods, and the amount of work in progress is minimized.

In order for agile methods to work, the software coming out of one iteration must not only meet the requirements of the user, but must also have internal qualities so that it may form a solid basis for the next iteration. Software needs to be built so that it can be adapted and extended, perhaps in ways that were not originally thought about at the start of the project. The XP approach (Beck & Andres 2004) adopted by the Chaste project prescribes a number of engineering practices to ensure that this is the case. Some of the most relevant practices such as releases, iterations, test-driven development and collective code ownership are described below.

(i) **The customer and user stories**

Agile methods introduce the concept of a *customer* who is interested in the finished product and able to steer the project through close interactions with the developers. User stories are used instead of a large requirements document. These are short descriptions (a paragraph or two) of features that the customer of the system desires. User stories contain enough detail for the customer to estimate the relative value of a story (in terms of extra functionality added to the program), and the development team to estimate the effort required to implement the story. The granularity of user stories is such that the developers’ estimate is between one and three programming weeks.

(ii) **Release planning**

The aim of release planning is to agree the scope of the project: which user stories will be implemented in which software release. Releases should be made early and often to maximize feedback. The amount of effort available to each release can be calculated from its duration and the size of the development team, and an appropriate set of user stories (based on the effort estimate for each story) is assigned to the release. In contrast to the plan-driven development, the release
plan can be altered throughout development as more information about the project comes to light. For example, the development team may wish to revise its effort estimates based on its experience of developing previous similar user stories or the customer may decide that some stories deliver greater value than they first thought due to changes in the marketplace (or new scientific discoveries).

(iii) **Iteration planning**

The development of each release is split into iterations that last two to four weeks, as illustrated in figure 1. The user stories to be implemented for the iteration are chosen from the release plan. It is at this point that the developers and customers start to think about the details of the user story, as they unpack the story into finer-grained programming tasks. This provides a more accurate estimate of the effort required for the iteration and it may be necessary to vary the stories in the iteration as a consequence.

(iv) **Acceptance tests**

At the start of an iteration, customers and developers work together to design a set of automated tests, which will show that each user story in the iteration has been implemented. The tests are automated so that they can be run regularly to measure progress. Tests from previous iterations are also run to guard against regression.

(v) **Test-driven development**

Test-driven development requires that an automated unit test, defining the requirements of the code, is written before each aspect of code, so that it will initially fail (in contrast to acceptance tests, which check the behaviour of the overall system, unit tests check the behaviour of a component part). The test-driven development cycle proceeds as sketched in figure 2.
The test driven development allows the developers to think about what the system should do before working out how it will do it. If a test fails, the developer knows where the problem was created and there is a small amount of code to debug. Therefore, a further advantage of test-driven development is that the time spent debugging is reduced compared with other approaches.

(vi) Refactoring

The test-driven development is not only a testing methodology but also a design methodology, when combined with refactoring. Refactoring code means changing its design without changing its external behaviour (Fowler 2000b). Since only just enough code is added to make tests pass, there is less need to plan ahead and the code is not polluted with guesses about what might be needed in the future. However, over time, these small additions can lead to a less than optimal design where, for example, similar code is repeated in several places. Refactoring removes this duplication and keeps the design clean, so that the code stays malleable. Refactoring can proceed in very small steps that gradually make large changes to the design. Figure 2 shows that refactoring is an integral part of the test-driven development cycle.

(vii) Collective code ownership

Collective code ownership encourages everyone to contribute new ideas to all segments of the project. Any developer can change any line of code to add functionality, fix bugs or refactor. No one person becomes a bottleneck for changes. No one person has to maintain the overall vision for the design of the system.

(viii) Pair programming

Pair programming refers to the practice of two people sitting at one keyboard and screen when programming. While one person is typing, the other is reviewing their work and thinking strategically. The pair alternates the one who types. It may be argued that two people working at one computer can add as much functionality as two working separately, but the code will be of higher quality. Pair programming is also combined with pair rotation: from time to time one of the developers in a pair will swap with a developer from a different pair. This
allows people to move around the project and see different parts of the code base, but still maintains continuity since one of the developers in the new pairing remains.

(ix) **Continuous integration**

Continuous integration is a software development practice where members of a team frequently integrate their separate streams of work from their local machines (at least daily). When the code is integrated, the set of automatic tests are run to ensure that no integration problems have occurred. The tests that each programming pair has created are also added to the set of automatic tests during integration, facilitating collective code ownership. Because code is integrated every few hours, incompatibilities between different streams are detected sooner rather than later.

(x) **Stand-up meetings**

A stand-up meeting takes place between the whole team every day. The meeting is held with the team members standing up, in order to keep the meeting short. People report what they have done since the last stand-up meeting and what they plan to do before the next one. It is also a chance to raise any problems that are holding up progress. The stand-up meeting allows each member to have his or her say in steering the project.

(xi) **Whole team**

This practice refers to the use of a cross-functional team with all the skills and perspectives necessary for the project to succeed. Ideally, the customer should be on-site (and therefore part of the team) and available to answer questions as necessary. No documentation is mandated by the agile method, because this is replaced by face-to-face communication. Therefore, where possible, the team is located at the same place.

Predictability is not impossible, but most projects do have changing goals. Agile methods are suited to problem domains where requirements are prone to change. They work best with small teams of 2–12 people (Beck & Andres 2004), although success has been reported with larger teams.

(b) **Adaptations of agile programming to the scientific domain**

The agile approach described in §4a, although desirable for the reasons outlined above, is not ideally suited to the academic environment in its current form. Some principles of agile methodologies, such as the iteration system, are as suited to academia as to industry. However, during the progress of the Chaste project, we found that other aspects needed to be adapted. Other developers of scientific software (Wood & Kleb 2003) have noted similar adaptations. We now discuss our adaptations of the agile method to the Chaste project and present a summary of these adaptations in table 1a. A summary of the positive influences that agile methods have over the traditional methods of software development in the academic culture is shown in table 1b.
The role of the customer is somewhat blurred in academia compared with industry. There are, in essence, a range of different customers, each with different requirements. At one extreme, the developers themselves are customers if they are developing the code partly for their own research. At the other extreme, in the case of cardiac software, the customer is the cardiac modelling community that requires a fully functional piece of software. In between, customers can be researchers in the same department as the developers, with perhaps working knowledge of the code and regular contact with the developers; or if the source code is released, other cardiac developers in external departments. The most important customer (the customer for whom the current work is focused) can change regularly. Traditional agile approaches require the customer to be in constant contact with the developers, but this is not always possible in academia. The different role of the customer in the scientific domain thus needs to be taken into account when using agile methods. Careful setting of priorities is crucial.

(ii) Releases

Agile software development calls for frequent releases of working software (see §4a). There are some difficulties in applying this within an academic setting due to the need to publish. This ties a release (to the wider community) of fresh functionality with an academic paper. In part, the release of Chaste has

Table 1. (a) Adaptations to traditional software development in the academic culture. (b) Adaptations made to fit agile programming into the scientific domain.

(a)

<table>
<thead>
<tr>
<th>agile development (§4a)</th>
<th>agile adapted to the scientific domain (§4b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer on-site</td>
<td>investigators/programmers are the customers</td>
</tr>
<tr>
<td>frequent releases to the users</td>
<td>release tied to publication</td>
</tr>
<tr>
<td>tests are well defined</td>
<td>unknown solutions and numerical tests need</td>
</tr>
<tr>
<td>all code collectively owned</td>
<td>tolerances</td>
</tr>
<tr>
<td>all code pair programmed</td>
<td>some solo project code</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th>traditional academic (see §3b)</th>
<th>adapted agile</th>
</tr>
</thead>
<tbody>
<tr>
<td>code is handed between researchers</td>
<td>pair programming</td>
</tr>
<tr>
<td>single investigator in charge of code</td>
<td>collective code ownership</td>
</tr>
<tr>
<td>tests decided after implementation</td>
<td>acceptance tests written first</td>
</tr>
<tr>
<td>code tested on grand scale</td>
<td>test-driven development with small unit tests</td>
</tr>
<tr>
<td>long-term (grant-level) plan</td>
<td>levels of planning (release and iteration)</td>
</tr>
<tr>
<td>less-malleable code</td>
<td>frequent refactoring</td>
</tr>
<tr>
<td>infrequent sharing of code</td>
<td>frequent integration with team</td>
</tr>
</tbody>
</table>

(i) The customer

The role of the customer is somewhat blurred in academia compared with industry. There are, in essence, a range of different customers, each with different requirements. At one extreme, the developers themselves are customers if they are developing the code partly for their own research. At the other extreme, in the case of cardiac software, the customer is the cardiac modelling community that requires a fully functional piece of software. In between, customers can be researchers in the same department as the developers, with perhaps working knowledge of the code and regular contact with the developers; or if the source code is released, other cardiac developers in external departments. The most important customer (the customer for whom the current work is focused) can change regularly. Traditional agile approaches require the customer to be in constant contact with the developers, but this is not always possible in academia. The different role of the customer in the scientific domain thus needs to be taken into account when using agile methods. Careful setting of priorities is crucial.
been held back by this issue, since a large amount of functionality is required to compete with the existing cardiac simulation packages, giving a large initial threshold to cross.¹

Note that internal releases, within the group responsible for development or close collaborators, do not share the same need to be driven by publications. Thus, the iteration structure of producing working code at the end of each iteration is not affected. We are continuing to consider the optimal strategy for wider releases.

(iii) Test-driven development and unit testing

Unit testing in scientific computing can be more difficult than that in the industrial setting. Components of a typical scientific computing system, and of cardiac codes in particular, can be loosely classified in one of the following three categories: structures; numerical algorithms; or models. Structures (such as a mesh class) have a well-defined and relatively simple functionality, are relatively easy to test and, in this sense, are the most similar to the typical components of software developed in industry. Numerical algorithms for solving generic problems, for which a problem with an analytical solution is available, can also be easily tested, with near 100% faith in the quality of the test. Writing tests for numerical algorithms often has the added complexity of having to choose suitable numerical parameters and tolerances, such that the tests pass even when using different compilers and optimizations. Algorithms for more complex problems are more difficult to test. Models, especially complex models such as cardiac models, can be very difficult to test, with often the only available types of test being qualitative comparison against experimental results or, if available, comparison with other software.

To illustrate this, consider the following requisite components of a cardiac bidomain model: ODE cell models; ODE solvers; a mesh class; a finite-element assembler (a component that assembles, at each time step, the r.h.s. of the linear system $A\mathbf{x} = \mathbf{b}$, as described in §2); and a linear system solver. Each of those components needs to be individually tested, as does the top-level bidomain model, which brings the components together. There are no problems testing the mesh class, nor are there any in testing the ODE solvers or the linear system solvers, other than choosing tolerances, as ODEs or linear systems with known solutions are easy to set up. The finite-element solver can also be tested by solving simplified problems (such as the diffusion equation) and comparing against known solutions. However, cell models are difficult to test, since they tend to be extremely complex, meaning that (i) analytic solutions are not available and (ii) a parameter being entered incorrectly may not cause any large change in the results obtained using that cell model. Similarly, it is extremely difficult to test the full bidomain code, for the same reason of the correct solution being unknown, and also because the complexity of the system (together with the computational issues described in §3) mean that long-running simulations (especially in three dimensions) need to be run to confirm the code passes each test.

These problems are partially overcome by performing heavier testing of the individual components of the system and performing heavy and wide-ranging tests, run overnight if necessary, of the final system. Of course, the difficulty

¹ Licensing has also put back a release; see §6.
in testing the complex final model illustrates that test-driven development is vital to, rather than inapplicable in, code development in academia. All base components of a scientific computing system must be tested in order to have a reliable final model.

(iv) Collective code ownership

The fact that developers have their own research goals means that collective code ownership is not always maintained for the whole code base. Individual projects, based on the core Chaste libraries, may be the work of a single person. This may even be necessary—postgraduate students, for example, need to write their own thesis and must be able to demonstrate their own original work. For these reasons, there are components of the code which are maintained only by one or two members of the team.

For the core components, however, collective code ownership is still required if the project is to enjoy long-term success, especially considering the fluid nature of the academic workforce. The optimal mixture of pair and solo programming, which (i) prevents components of the system becoming opaque to all developers except one while (ii) not being too restrictive for academia, is yet to be determined. A code review mechanism is currently being discussed, whereby changes coded individually are peer reviewed before being committed to the core code base. Code review and communication of such changes are currently facilitated by the ‘changeset’ feature in Trac (see §5), and enhancements to better support peer review are being investigated.

(v) Pair programming

Pair programming is clearly not always possible in the academic environment. Unlike in industry, where each member of the workforce is employed by the business and can focus entirely on the project if required, the ‘workforce’ in academia is made up of researchers with a wide range of objectives and goals. Developers of communal codes in academia will generally be made up of postgraduate students and postdoctoral researchers. Postgraduate students have their own large, fixed-deadline project that limits their availability for pair programming. Also, at some point, their use and development of the code will need to focus on their own work, which may not be the same as the current aims of the project, in which case pair programming becomes inappropriate. The same, to a slighter lesser extent, applies to postdoctoral researchers who, again, have their own goals, areas of research and deadlines. Furthermore, the Chaste project has involved developers from multiple institutions, separated geographically, rendering pair programming too restrictive to be used without exception.

The Chaste project has used a mixture of pair and solo programming throughout its lifetime. Initially, development was carried out only part-time, and all development was done in pairs. As the code progressed, changes have had to be (and are still being) made to the working process. A group development session is held a day a week, where all code is pair programmed. Outside these sessions, the software is developed with solo programming, although large-scale changes to core aspects of the system are discouraged. This naturally has implications for the concept of collective code ownership, as we saw above.

*Phil. Trans. R. Soc. A* (2008)
One significant advantage of agile methodologies outlined in §4a is that they allow for adaptations such as those mentioned in §4b. We have thus been able to alter the way in which we work on Chaste to cope with changing circumstances. It should be stressed that the development method itself is constantly refined under an iterative process since the exact needs are not well defined. In §5, we present the current status of the project, showing what our use of agile techniques has enabled us to achieve.

5. Current status of the Chaste project

After approximately 3 years of development effort, the Chaste libraries now number approximately 50 000 lines of C++ code. This main code is supported by a similar amount of testing code (also in C++) and a number of infrastructure scripts (written in Python). In this section, we concentrate on the current capability of the software and on assessing progress to date in terms of performance and software engineering.

(a) Choice of programming language

The aim of the Chaste project is to create programs for use in a number of different biological problem domains (initially cardiac and cancer), but to do so with as much modularity and code reuse as possible. For example, modellers in both problem domains have the notion of a cell model, which is expressed mathematically as a set of ODEs. Cell model functionality (and the underlying ODE solvers) can and should be shared between modellers.

The modular nature of the mathematical models influenced our choice of programming language (C++) and has also affected the software architecture. C++ was chosen because it has object-oriented notions such as inheritance and encapsulation that facilitate the writing of modular code. More modern object-oriented languages (like Java) were rejected since they carry a speed overhead.

(b) Capability

We focus here on Chaste’s capabilities with regard to cardiac modelling. In terms of electrical propagation, Chaste is able to simulate either the monodomain or bidomain models. There is a small suite of cell models (available as C++ classes) including Luo & Rudy (1991) and Fox et al. (2002) together with supporting tools to enable new models to be derived easily from CellML using PyCML (https://chaste.ediamond.ox.ac.uk/cellml/).

A standard Chaste bidomain experiment involves reading a tetrahedral ventricular mesh (typically a rabbit model with approx. 60 000 nodes), applying cardiac electrophysiological cell models and homogeneous material properties to the nodes, stimulating a patch of apex cells with a step-function stimulus current and tracking propagation over the course of one heartbeat. The propagation of one heartbeat (approx. 0.5 s of real time) currently takes over a day to calculate on a standard 2 GHz user desktop machine. This bidomain experiment uses a static geometric model, but we have developed solution methods for electromechanical feedback and are incorporating these into larger scale models.
A typical Chaste cancer experiment involves setting up an approximation to a colon crypt on a cylindrical domain using a triangular mesh. The experiment is run for approximately 1 day of simulation time using a cell–cell spring interaction model (Meineke et al. 2001) and a bespoke research model of other important features in the genetic pathway to mutation.

(c) Assessment

In terms of assessing the performance of Chaste, it has proved difficult in the absence of access to other source code and developers to perform like-for-like computations and thus to gauge where Chaste fits in relation to other packages. Over a series of sample runs, it has been estimated that the speed of Chaste on a bidomain problem is within a factor of five of the fastest available bidomain solver (CARP). This is encouraging since Chaste functionality is generic and has not been specifically crafted for bidomain problems.

Meanwhile, we have had a full evaluation of the source code from an industrial partner. This independent review showed that Chaste scored very well on code quality, readability, software architecture and general software engineering. A senior software engineer (who has no physiology background) commented on the ease with which he was able to understand how the code related to the underlying mathematical model. The industrial partner also performed a number of sample tests of the bidomain solver on a range of computer clusters. These tests highlighted that the techniques then used for writing output were an issue for future scalability, but that (without output) the code scaling was close to linear on up to 16 processors. On some tests, super-linear speedup was achieved.

As we mentioned in §3b, software development in academia usually faces the problem of the high turnover of the workforce. During the lifetime of Chaste, there has been a large changeover of developers. The original group of 20 full-time students was reduced to a group of part-time developers. Some doctoral students joined the team as their research interests converged on Chaste objectives, while some developers abandoned the project as their research interests diverged. As more funding was obtained, new full- or part-time researchers were contracted. Based on approximately 3 years of experience, we are able to assess the suitability of agile programming methods to address this issue. This turnover of the workforce generates two different problems: how the team assimilates a new member and what effects the departure of a member of the team has on the team.

First of all, following the traditional approach to software development, a new researcher would spend several weeks reading documentation and studying the code before integrating into the project in a useful and productive way. On the other hand, with agile programming, the novice is paired with an experienced programmer during the pair-programming sessions. The pair will focus on one user story, so the novice will be introduced to a small, specific part of the code by the experienced programmer. Only after a few pair-programming sessions, the novice will become familiar with several independent parts of the code and, by extension, with their interaction. This aids an incremental overall knowledge of the code base. Some might say that this approach reduces the productivity of the experienced programmer. Nevertheless, this tuition allows for the experienced programmer to consolidate his or her knowledge of the code and the techniques.
implemented; therefore, we consider it profitable for both programmers. In addition, during this ‘guided tour’ of the code, the novice programmer carries out an external evaluation of the code based on their previous experience. Deficiencies can be pinpointed and discussed with the experienced programmer. With this approach, the novice programmer will feel useful from the beginning and will not be overwhelmed by the size of the code base.

The second challenge is how much the departure of a member of the team will affect the project. Thanks to the pair-programming sessions, in which the knowledge of the workings of any part of the code is shared among all the developers, the black-box situation cited in §3b is not likely to arise and the longevity of the code is ensured.

At the same time, we have realized that the adoption of agile programming methods has dramatically increased the quality of our code. All the programmers are aware that the code they write is going to be reviewed by another person in a matter of a few days if not hours. This situation encourages programmers to be rigorous in the design and documentation of the source. Frequent pair rotations ensure that the majority of the developers have worked in a user story before it is completed.

(d) Infrastructure

As the Chaste project grew and code was developed, we found it necessary to develop a support infrastructure (a set of web pages, programming tools and scripts) to facilitate the developers’ work. Initially, the development team used textbook agile methods for introducing, breaking down and solving tasks. Tasks were written on postcards and all developer interaction was in stand-up meetings, via emails or on whiteboards. It soon became clear that some of these methodologies were not suitable for the academic environment and would not scale with time (see §4b). The development team therefore adopted the TRAC (http://trac.edgewall.org/) project management tool in order to track programming tasks. TRAC also provides a place to document project decisions (in the form of a Wiki) with links through to test results. Developers may cross-refer between Wiki discussions, TRAC tickets (programming tasks and bug listings) and ‘changesets’ in the Subversion (http://subversion.tigris.org/) code repository. The TRAC installation used on the Chaste project has been augmented to aid browsing of unit tests—scripts on the website are able to retrieve the pass/failure results of current and past automated tests and to display them as a series of green/red bars.

The Chaste project developers use the Eclipse integrated development environment (http://www.eclipse.org/) that interfaces well with the Subversion version control system used. CxxTEST (http://cxctest.sourceforge.net/) provides a unit testing framework and SCons (http://www.scons.org/) is used for build automation.

(e) Testing

The importance of test-driven development and of regular testing was introduced in §4a, while complications specific to scientific programming were discussed in §4b. In order to support testing, Chaste uses the SCons build scripts not only to compile code but also to trigger the execution of tests. Dependencies
are checked so that incremental builds only trigger the execution of tests that depend on changed code. Custom Python scripts provide an HTML document with a summary of test results and hyperlinks to the detailed output of all tests. This allows developers to tell quickly whether the build is working and, if not, find the detailed test output associated with any problems.

When a new version of code is committed to the version control system, a script checks out the new version, performs a fresh build and executes the tests on a separate machine that is not used by any developers. The results are then published on a web server. Developers should check whether the code passes tests prior to committing—the addition of a neutral integration machine is a double-check. A secondary purpose of the machine is to make the test results public, so peers can verify that nobody is checking in faulty code. Some tests have a long execution time so they are run only overnight or once in a week. Also included in the overnight tests are a memory-leak check that uses VALGRIND (http://valgrind.org/) to alert the developers of memory leaks and non-determinism in the code, and a coverage test to ensure that every line of executable code is called at least once during a standard set of tests.

An example of nightly testing is that of the convergence tests that are high-level tests built on tests of more basic functionality. We regularly check that the code converges as predicted by mathematical theory by repeatedly running the same bidomain simulation (a simple stimulation of a block of tissue) with various parameters (time and space steps). Between concurrent runs, we monitor changes to some key physiological parameters: the transmembrane potential and action potential duration at a representative point in space and the wave propagation velocity between two points. The primary convergence parameter is the relative change to the transmembrane potential plotted over time, measured in the $L_2$-norm metric. By simultaneously tracking several vector and scalar quantities, we can not only confirm that our convergence criterion happens at the right point, but also confirm the analytical prediction that convergence is linear in the size of ODE and PDE time steps and quadratic in space step. Furthermore, we are able to inform users of the precise trade-off (between the speed of a simulation and accuracy of a particular quantity) made by setting certain parameters.

(f) Architecture

In describing the current state of the Chaste project, we conclude with an outline of the software architecture. The main components of the cardiac code (and their dependencies) are sketched in figure 3. Throughout the course of the project, we have made use of open-source libraries where there is a good fit to our needs. The large-scale linear-algebra routines and the parallelization of data are handled through calls to the PETSc library (http://acts.nersc.gov/petsc/) that is built on the de facto standard libraries for distributed-memory parallel programming (MPI) and linear algebra (BLAS and LAPACK). The linalg library in figure 3 is largely a set of interfaces into the PETSc library. For efficient small-scale matrix and vector manipulations, Chaste uses routines from the C++ Boost libraries (http://www.boost.org/). Boost is also used for the serialization of objects to enable checkpoint and restart.
In order to experiment with a range of bidomain finite-element solution techniques (such as backward time stepping, operator splitting or space refinement as described by Whiteley 2007), we have developed our own hierarchy of ‘assemblers’ that relate PDEs to the construction of the underlying linear system. An assembler class takes in a mesh, a PDE object and a set of boundary conditions, uses finite-element theory to approximate the PDE and outputs a corresponding linear system.

As described in §3, some aspects of the cardiac electrical propagation problem are stiff and in these cases we need to make a judicious choice of numerical technique. In the case of the cell ODE model, we use a backward Euler solution method as described by Whiteley (2006), which solves the linear and nonlinear dependencies in separate phases. Since this solution technique breaks the standard model of ODE solution, we have developed our own library of ODE solvers. Improvements (described by Cooper et al. 2006) have been made to the speed of computing the cell models through the use of partial evaluation and lookup tables. Since the process of computing the split between linear/nonlinear dependencies is tedious and error prone, there is an interface to enable automatic generation of ‘backward’ models from CELLML using PyCML (https://chaste.ediamond.ox.ac.uk/cellml/).

(g) Analysis

In this subsection we analyse some of the process data from the Chaste project to help indicate the usefulness of an agile approach.
Figure 4 shows a plot of the sum total of effort of completed stories as a function of time, where ‘effort’ is estimated \textit{a priori}. (Recall that project work is defined in terms of user stories and when these are created an estimate of effort to implement the story is made; see §4a.) The origin of the chart corresponds to a point midway through the project (the start of iteration 8), which is when we started to record the data necessary to produce the chart. Each datum point on the chart corresponds to the end of an iteration. It is remarkable that the plot is almost a straight line. This indicates that the team is consistently able to estimate its ability to complete work. At least this is the case from iteration 8; it may be that such a plot corresponding to the very early part of the project would be more erratic indicating that the team was still getting used to the agile processes and honing its ability to estimate work.

Figure 4a shows a plot of the sum total of effort of completed stories as a function of time, where ‘effort’ is estimated \textit{a priori}. (Recall that project work is defined in terms of user stories and when these are created an estimate of effort to implement the story is made; see §4a.) The origin of the chart corresponds to a point midway through the project (the start of iteration 8), which is when we started to record the data necessary to produce the chart. Each datum point on the chart corresponds to the end of an iteration. It is remarkable that the plot is almost a straight line. This indicates that the team is consistently able to estimate its ability to complete work. At least this is the case from iteration 8; it may be that such a plot corresponding to the very early part of the project would be more erratic indicating that the team was still getting used to the agile processes and honing its ability to estimate work.

Figure 4b shows data that are commonly referred to as a \textit{burn-down} chart. This plots how the estimate of the amount of work left in an iteration varies through the iteration. (The effort estimate associated with a user story is

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4}
\caption{Some project statistics gathered from the Trac project management database. (a) The total amount of work completed (solid line) over the last six iterations. (b) The burn-down chart for iteration 11 (solid line, estimated work remaining).}
\end{figure}
updated as the story is worked on. Whereas the initial estimate was used as a basis for the data in figure 4a, the current estimate is used as a basis for the data in this chart.) It can be seen that the plot is not strictly decreasing: sometimes, when working on a user story, one realizes that its implementation is more difficult than previously thought and the effort estimate needs to be increased. Our records show that the steep initial descent of the plot was due to some overly cautious estimates of effort: having spent a little time on some of the user stories at the start of the iteration, their effort estimate was revised down. The developers did not work on the iteration in the second week (due to other commitments). From then on, slow progress was made with some user stories proving more difficult than expected. Indeed, some user stories were moved from the scope of iteration 11 to the following iteration during the last week as it became clear that they would not be completed.

Figure 5a shows how the number of lines of source code and test code varies over the lifetime of the project. Because the organization of the source code repository was in a state of flux up to week 8 of the project, the data for this part of the chart are incomplete.

The steep ascent of tests during the first few weeks of the project was due to the intensive period of work associated with the training course, which kicked off the project. Then came a period of consolidation where the code was refactored, memory leaks (which had not been tested in the training course) were debugged and the code was parallel enabled. This was interspersed with periods of inactivity when no one was working on the code and lasts from weeks 10 to 60. This explains the flat portion of the graph during this period. From then onwards, the number of lines of code increases, reflecting the greater amount of effort expended on the project, with new features being added. From time to time, declines in the number of source lines of code (and to some extent test lines of code) reflect the refactoring work.

It is remarkable that the number of source and test lines are almost identical throughout the project. To some extent, this is expected from the test-driven methodology, although why the ratio of source to test lines of code is almost exactly one is not known. From approximately week 130 though, this relationship appears to be breaking down. An explanation is that some code added to the Chaste repository around this time was machine generated from CELLML using PyCML. This is more verbose than handwritten code and is tested by adapting and augmenting tests for other cell models.

There is a natural close correlation between the number of lines of test code (figure 5a) and the number of individual unit tests (figure 5b). This shows that a typical test is approximately 50 lines long. The fact that the number of tests has grown linearly with the amount of source code and that there are now approximately 900 tests demonstrates the completeness of unit testing. When functionality is changed erroneously and tests fail, then it is a simple matter to locate the failing test(s). Because testing occurs at all levels of the software hierarchy, the tests that fail make it easy to find and fix problems in the actual source code. This is different from the traditional software testing method where either a simple simulation serves as an acceptance test or few distinct sections of the code (such as cellular models) are protected by a small number of tests.

Phil. Trans. R. Soc. A (2008)
6. Conclusion

The Chaste project started in May 2005 as a four-week teaching exercise and an experiment in the use of agile methods to produce high-quality computational biology software with an interdisciplinary team. In continuing this project, and adapting to changes in the team composition and requirements from users, we have made modifications to the agile development process to better fit our environment.

The fluid workforce that characterizes academia was one of the main reasons we decided in favour of the agile approach. Indeed, a possible consequence of such a workforce is that efforts on a software project may get postponed.

Figure 5. (a) The number of lines of code (in the actual source, excluding test code and in unit tests; solid line, source code; dashed line, test code) over the lifetime of the project. (b) The number of individual unit tests.
or abandoned) when someone involved in (or responsible for) that project leaves. In the worst case, this will mean months, if not years, of work rendered useless.

Such an issue is, however, inherently addressed by the agile approach, since no project can solely rely on the expertise of one particular developer. This approach also allows for the easy and quick integration of new team members who can learn from well-documented and clean code, as well as from other coders through pair programming, a key feature of this methodology. Tests can also be used by any new or existing team member to learn what the code does and how to use it.

The use of pair programming in agile development fits well with interdisciplinary projects such as the present one. Someone with a strong numerical background, but limited coding skills, might for instance be paired with someone with the opposite background, allowing the two to benefit from one another’s expertise. Such a mutual benefit is reinforced by the team sharing the same office during coding sessions. Such an environment allows for what has been termed ‘osmotic communication’ (Cockburn 2001), in which knowledge is transferred through overhearing conversations between other pairs—one may hear a discussion on a problem that one knows how to solve, and hence be able to contribute.

The wide range of coding skills available in a project like Chaste means that it is inevitable that some pairs will produce better code than others. However, we believe that the quality of our code would have been worse had we opted for a more traditional approach to software development. Also, its quality can improve only with time, since nobody is solely responsible for any particular piece of code and, therefore, any pair of programmers may end up reviewing and improving existing code.

Agile approaches are based on the observation that project requirements are rarely set in stone, but change over time. This is particularly the case with academic research projects, since it is in the nature of research to suggest new directions to follow which had not been anticipated. We regard this as a key reason to favour the use of agile methods in developing academic software. Discipline in following the test-first approach yields thoroughly tested code and provides a solid foundation for refactoring to cope with changing goals. This is not to say that project leadership and vision should not be sought, since they can only ease the software development process, aiding in setting priorities. Rather, it recognizes that such vision is not infallible and, indeed, that a solid foundation is also useful for extensions that were predicted.

One of the most important characteristics of a programming methodology is whether or not it allows rapid code development. This is of particular importance in academia where both research students and staff are expected to complete a significant volume of work before the end of their fixed-term contracts. Although we have mainly focused on the application of Chaste to cardiac problems elsewhere in this paper, it is perhaps useful in this regard to compare this section of Chaste to the section that focuses on cancer modelling. The programmers on the cardiac section of Chaste have mathematics and computer science backgrounds. Although these workers have chosen to work within a Computational Biology Research Group, their research interests are in writing efficient, reliable code that is underpinned by efficient, robust
numerical algorithms. As a consequence, effort has been directed towards those goals, and the generation of a code that may be used as a tool by workers with an interest in physiological problems has been slower than originally anticipated. By contrast, the cancer section of Chaste has been coded as a collaborative effort between workers on the cardiac section and research students who are mathematical modellers and have a clear idea of the modelling problem they wish to solve. There are more tangible outputs on this section of the code with three research papers currently in preparation for submission to peer-reviewed journals. Based on these experiences, we tentatively suggest that this methodology is more efficient when the programming team includes workers in the application area of interest, who stand to benefit from the final product.

It should also be pointed out that the cancer section of Chaste benefitted significantly from both the software infrastructure and the functionality of the code that had already been put in place when developing the cardiac section of Chaste. We find this encouraging, as it suggests that future novel application areas may be developed relatively rapidly compared with the cardiac section of Chaste. Furthermore, we anticipate that developing the cardiac section of the code to include features such as tissue deformation and fluid mechanics will also be more rapid for the same reasons.

It is our opinion based on our experience with the Chaste project that the use of agile development is beneficial to producing high-quality scientific software capable of being easily adapted to current research requirements, and gives significant advantages over development with no thought of methodology. The features of our research setting which led us to embark upon an agile development process are largely covered in §3. Briefly, these features can be identified as (i) a high turnover of workers on short-term contracts, (ii) a wide range of scientific skills and coding proficiency, (iii) the need for a maintainable and extensible code base that will outlive particular research grants and (iv) changing requirements that are driven by new scientific discoveries.

Finally, there are two points that should be considered by anyone interested in applying the agile approach to their software project. Having a good infrastructure to support software development is essential, and indeed a significant portion of the time spent on Chaste to date has been in improving our infrastructure (see §5). Also, the type of licence to be used for the project should be addressed as early as possible, as this can be a time-consuming process. This is especially the case if it is not considered until after work has started, since it is then practically impossible to determine which institutions own which parts of the code, leading to a confusing legal situation. In the case of the Chaste project, we have now opted for a GNU LGPL licence, as this allows our industrial partners more flexibility while still allowing for open-source distribution. When the licensing issues are resolved, Chaste will be available from https://chaste. ediamond.ox.ac.uk/.

Development work on Chaste is supported by the EPSRC as part of the eScience Integrative Biology pilot project (GR/S72023/01) and as a software for high-performance computing project (EP/F011628/1). B.R. is supported by an MRC Career Development Award (G0700278). A.G. is supported by a grant from the UK Biotechnology and Biological Sciences Research Council (BB/E024955/1).

Phil. Trans. R. Soc. A (2008)
References


*Phil. Trans. R. Soc. A* (2008)
Rodríguez, B., Li, L., Eason, J., Efimov, I. & Trayanova, N. 2005 Differences between left and right ventricular geometry affect cardiac vulnerability to electric shocks. *Circ. Res.* 97, 168–175. (doi:10.1161/01.RES.0000174429.00987.17)


