Published in final edited form as:

Comput Sci Eng. 2009 November ; 11(6): 20–29. doi:10.1109/MCSE.2009.198.

Managing Chaos: Lessons Learned Developing Software in the Life Sciences

Sarah Killcoyne and John Boyle

Institute for Systems Biology

Life sciences research is, by nature, borderline chaotic. Scientists tend to work in small, isolated, and focused groups, collaborating only loosely with others. The process of testing and refining (or discarding) hypotheses leads to a multitude of elaborate experiments—each of which differs, using a unique mix of techniques, technologies, and analyses. Research mechanisms constantly change; researchers are continually introducing new technologies and refining older technologies. Experimental results can lead to myriad conclusions, some of which are contradictory and others of which are ignored. This constantly shifting landscape means that scientific discovery can sometimes be perceived as a manic foraging exercise rather than a rational, hypothesis-driven process. One of the most confusing elements of science is that this jumble of experiments leads to the development of ideas that directly advance our understanding of living systems. That is, the system works, and works well.

Broadly speaking, software development in the life sciences typically comes in one of two models:

- isolated developers working within large research teams or
- large multi-year, -site, and -million dollar infrastructure projects.

The isolated software engineer model is prevalent, with engineers expected to act both as developers (in creating tools) and as scientists (in publishing papers and grants). As a result, the software they develop is usually highly specialized to their particular research, but rarely extensible or interoperable. Alternatively, funding agencies might require developers to create software architectures using a set of domain objects that conform to standards that aren't fully accepted within the life sciences. These projects typically take too long to develop and suffer from poor adoption.

As an alternative to these two approaches, the Institute for Systems Biology (ISB) formed a research informatics team to identify and address the unique challenges of developing software in the life sciences. The team's solution includes a set of software processes and design principles tailored to the life sciences research environment.

© 2009 IEEE

In the life sciences, the need to balance the costs and benefits of introducing software processes into a research environment presents a distinct set of challenges due to the cultural disconnect between life sciences research and software engineering. The Institute for Systems Biology's research informatics team has studied these challenges and developed a software process to address them.

Selected articles and columns from IEEE Computer Society publications are also available for free at <http://ComputingNow.computer.org>.

The Challenges

Although various software development issues arise across scientific disciplines,^{1–3} the life sciences have a distinct set of challenges. The life sciences—and systems biology in particular—change rapidly, with new discoveries and alterations in fundamental thinking occurring with alarming regularity. Except for the most trivial examples, developing any software to meet continually changing needs is difficult. Any rapidly evolving domain, no matter the industry, faces numerous challenges:

- skill gaps among team members, with a growing number of developers who have no formal training;¹
- poor specifications resulting from conflicting use cases and a lack of requirements;
- no overall project vision, leading to feature creep or outright failure;
- software developers being expected to play too many roles, including hardware experts and IT support; and
- managing the complexity of required technologies and standards.

These issues, while interesting, are well understood and largely self-explanatory. However, the life sciences present additional, unique challenges to software development. These challenges stem from the fact that, in essence, biology is a descriptive science that currently lacks a grand underlying mathematical theory. In comparison, physics, which arguably hasn't been descriptive since the late 1700s, has a far more formalized basis. Where there's a high level of formalism, there's also a more natural fit with software development, as the focus is on *how* to implement (tools, language, methods) rather than on *what* to implement. In fact, while physicists might not follow best development practices,⁴ they generally understand how code works more intuitively than biologists (most scientists in the life sciences who code are actually physicists or engineers).

Based on our experience at several life sciences academic institutions (including ISB) and commercial organizations, we've identified three major challenges that developers within the life sciences face that might not be obvious to those working outside the field. First, developers are isolated within laboratories; they rarely work in teams, and collaboration isn't actively encouraged. Second, they perceive success differently. Within biology, for example, the focus is on novelty and discovery. Finally, the ground truths in life sciences are transient, which presents problems with any attempts to formalize or model aspects of the domain.

Challenge 1: Developers Are Isolated

A rift exists between the domain-independent software engineering and the scientific computing communities. This “chasm”⁵ is the result of basic training in the respective cultures of software engineering and scientific research.

The discovery-oriented, project-focused, publish-led culture of research encourages scientists to work in small isolated silos. Collaborations that arise among these silos are usually only at the highest—read principal investigator (PI)—levels. In contrast, software developers are generally trained to work cooperatively in small teams. Each team member offers a particular set of skills and domain knowledge. Not only does this expand a team's knowledge base by including various technical specialists, but it also provides an environment that fosters the necessary creativity required to design useful software.

Within research, developers usually work as individuals. The lone software developer in the research lab is expected to be an expert in the entire software engineering domain, know the technologies and tools available within the scientific computing domain (including domain-

specific data analysis tools), and perform like a scientist (that is, write papers and grants, give presentations, and so on) while providing development (and often IT and basic statistical) support.

Although a group of software developers might collaborate across the laboratory silos, there's often little support from their PIs. It's common for developers to work on collaborative projects in relative secret, until the result is useful to their lab. This means that there's rarely any support for any formal processes that are required for successful development. In one case where several different institutions contributed to an application team, the developers themselves rejected any level of project management or instituting development processes. Instead of being a team, they wished to operate more “like a band,” where each developer would work on whatever feature interested them most.

We can trace much of the resistance to developer collaboration to a lack of regard for software development in general. The common attitude toward software within research is that it's not “real work,”⁶ even when professional engineers are involved. We can see this clearly when reading job advertisements for academic software developer positions, which frequently ask for developers with a background in nearly any discipline (such as engineering, physics, biology, mathematics, and computer science). Although this might be appropriate when the position is primarily research-oriented and the development can be “end user”⁷ driven, it's inappropriate when the work is principally nonspecialized software engineering. This isolation and failure to understand the work of software development leads to two key issues for developers.

First, developers are required to continually educate scientists. Regardless of their actual experience, as resident software “experts,” isolated developers must often educate scientists who lack a basic understanding of computing. Typically, this involves correcting fundamental misunderstandings of what software development is and what it can do, such as believing that the functionality in one application can be easily copied over to a new application or the idea that a single developer can develop commercial grade tools within grant-driven time periods.

Second, the developers are resistant to new ways of thinking. Developers who work in teams question each other frequently during the development process, and scientists commonly question each other's research. In contrast, the isolated developer is rarely questioned. In part, this is due to the lack of regard for software in science; technical decisions are considered unimportant. The outcome of being the sole developer and resident expert is an understandable resistance to new processes and technologies. Developers who've only ever needed to develop a script, single tool, or tool sets greet processes that might provide greater functionality or integration with skepticism. For example, during discussions about a standard Web service architecture, one developer dismissed the need for an underlying layered design, stating that “the idea of these (software) layers is folklore.” Another similarly dismissed it because he didn't “believe in Web services,” while a third simply thought that the use of these “new technologies” was both “banal” and “glitzy.”

Given these issues, the isolated developer is left to struggle with all the development process steps (researching appropriate technologies, investigating previous solutions, and analyzing requirements, along with design, implementation, and testing) as well as be the in-house expert on scientific software and educate their scientific colleagues—often with little support from the scientists themselves. Scientists who are busy keeping up with their own fast-moving field have little interest in understanding what's technically feasible, what software is already available, or how long new development might take.

Because science is driven by grant-based funding, the altruism of the long-term view is difficult to justify and even harder to realize.

Challenge 2: Success Is Perceived Differently

Because science is driven by grant-based funding, the altruism of the long-term view is difficult to justify and even harder to realize. Software developers working within the life sciences have struggled for years with a fundamental disconnect between the ideas and methods that drive software development and the scientific processes that dominate in the life sciences. Because scientists require analysis tools for specific projects immediately, the system favors the developer who provides a quick and dirty solution. Therefore, if integration and reusability are even considered, it's done in retrospect.

Developers are trained to work within a set of specific software methodologies that presumably lead to a working system and user acceptance.⁸ These methods also assume that user acceptance and long-term system viability are the development goals. In science, however, this is rarely the case: scientists are looking only as far as their specific project's lifespan. This narrow scope is needed in life sciences research, where the primary focus is on discovering new functionality and novel mechanisms (such as new gene functions, signaling cascades, regulatory mechanisms, and molecular complexes).

When software engineers new to the life sciences present work within an academic environment, they're understandably confused about the lack of audience interest in the technologies used to solve specific problems. When explaining why decisions were made, new software engineers are often surprised that their ideas are considered "too cerebral" and that they receive requests to remove "technical jargon" (such as "data management" and "robust software") from future talks. Developers are expected to communicate like scientists, which means continually reporting the novelty in software (that is, the new features or new tools), rather than their underlying methods and technologies. Although this equality between experimental scientists and software developers is admirable, it tends to lead to an adoption of a publish or perish approach. There are numerous cases where software is written solely to get a publication. This creates two problems.

First, the software isn't designed for use past publication. That is, publishing software is the goal of creating a tool, rather than designing it to ensure robustness or reusability. Maintenance and deployment issues are rarely addressed, as they're of no interest to scientific reviewers, who focus on the scientific functionality. When discussing these problems at a series of workshops for developers working on a specific scientific tool, the developers simply stated that there's "no point" to maintaining their code past publication. Clearly, this makes it increasingly difficult to develop generally useful infrastructure software, which has the same publication potential as a small utility (compare, for example, multi-year, -site, and -developer projects such as Addama9 versus garage-developed tools such as SeqExpress10).

Second, the software frequently mirrors existing functionality and standards. Because novelty is valued for its publication potential, software developers within the life sciences often recreate existing tools and standards. Examples include the multiple gene ontology enrichment applications (<http://geneontology.org/GO.tools.shtml>), genome browsers (such as the one at the University of California, Santa Cruz, as well as X:Map, Argo, and deCOD Eme), and network visualization tools¹¹ that offer nearly identical functionality. Software developers within the life sciences will also define their own standards, which are never intended for use outside the defining application. This phenomenon occurs in domain

standards—such as Cancer Bioinformatics Infrastructure Objects (caBIO), the University of Manchester's Mimas, and Bio-jETI—and extends to the redefinition of technical standards. Often, these new technical standards are advocated as competitors to existing, widely accepted industry standards (such as portlets being replaced by pipelets and Microsoft WebParts being replaced by Labkey WebParts).

Each of these issues is exaggerated by the demands on the PI to justify projects to granting agencies. Therefore, instead of asking whether the software is actually used, the question at project's end is often, *Where's the publication?* The expectation that software should be as novel as a scientific discovery has meant that even software that a developer would consider a failure (due to lack of use) can be considered a success by scientists.

Challenge 3: Ground Truths Are Transient

The life sciences, perhaps more than other domains, lack a set of basic standards that developers can use to reliably describe the domain. Within the life sciences, new experimental technologies are being constantly introduced, such as ChIP-Seq (Chromatin immunoprecipitation sequencing)¹² and Pulsed Silac (Stable isotope labeling with amino acids in cell culture),¹³ and basic principles frequently change, such as discovering that nearly the whole genome is actively transcribed or discovering new regulatory roles of noncoding RNAs or finding new mechanisms for non-DNA inheritance. While establishing requirements for software is difficult, the fact that the life sciences' established ground truths are transient makes this nearly impossible.

Numerous efforts have been launched to produce standardized models for different aspects of molecular biology, ranging from

- large community-based standardization processes, such as the Object Management Group (OMG) Life Sciences Research (LSR) effort;
- proprietary object models, developed by individual companies who make them public to encourage integration with specific products (such as Acero's Genomics Knowledge Platform);
- consortium standards, produced by cartels of companies, designed to promote market acceptance (such as the Interoperable Informatics Infrastructure Consortium or I3C); and
- research body standards, developed by different funding agencies to coordinate work among the research groups they sponsor (such as the object model from the National Cancer Institute's caBIO).

Various academic institutions and companies have also tried to standardize data integration and management systems for the life sciences. These have evolved from the database-centric systems that required federated data warehousing to object-centric systems where groups create several data standards that everyone adheres to (see Table 1). These solutions have evolved through a number of generations. Their architectures typically use formal definitions of domain-specific concepts and relationships (using, for example, Document Description Language, Interface Definition Language, or Web Ontology Language) that emerged from standards bodies such as LSR, I3C, caBIO, and the Semantic Web Health Care and Life Sciences (HCLS) group of the World Wide Web Consortium (W3C). Such solutions have generally failed to be widely adopted because they were too slow to evolve and too difficult to incorporate; examples include LSR, myGRID, BioMoby, Cancer Common Ontologic Representation Environment (caCORE), and BioRDF. The only true successes among these efforts have been simpler standards that provide basic annotation of

explicit types of experiment information (such as the Geo Soft format for gene expression and the mzML data format for proteomics spectra).

The main reason for the lack of universally adopted standards for specific life sciences concepts is that opinions always differ. These differences arise due to projects' diverse research aims and continually evolving scientific understanding. Even a concept as well known as the gene is difficult to model; the word is overloaded with meaning because different groups use it synonymously with a cistron, coding sequence (CDS), allele, locus, translated region, exon series, or operon. The concept of what is (or is not) a gene frequently changes. For example, the Encyclopedia of DNA Elements (Encode)¹⁴ project has highlighted how little we really understand about DNA transcription's role. We now understand that nearly the entire chromosome is actively transcribed, not just the bits that encode proteins (which in itself is a complex and poorly understood process).

Even if we could define a domain-specific object model, achieving adoption of that object model would be difficult solely due to granularity: the model would be too specific for some and too general for others. Likewise, many standard software processes are difficult to apply within the life sciences because model-driven approaches rely on a central tenant that might be inappropriate. In this sort of environment, software development can either attempt to impose a model—which will be poorly adopted because it won't be appropriate for all environments—or find - exible methods to provide the required functionality. When designs or models are imposed, past experience shows that it's better to ensure that they can be easily replaced (or ignored), typically by keeping the model small and putting the software in place quickly.

Design Principles and Process

The challenges of working in the life sciences are by no means insurmountable; it's possible to marry science's ad hoc nature with software engineering's structured requirements. Unfortunately, this is difficult to do within the traditional boundaries of project-funded sciences. The ISB has had the uncommon opportunity to directly change this environment by creating an informatics team that isn't tied to a single project but draws from many across the institution. This organization lets the team function like a traditional software development team and provides a number of services and benefits to ISB software development.

Because the informatics team spans multiple projects and laboratory groups, it can specifically address the challenges we described earlier. Having a team permits an exchange of ideas, as well as information about specific projects and dependencies. It also assists software developers in communicating with scientists—through papers, presentations, and grants—about the features and software the team's creating. Finally, using a team supports the rapid development of scientific applications.

Creating a Team

Because funding for our team comes from numerous different research projects, the first problem we had to solve was how to cooperate successfully within a research environment's constraints. Our informatics team spans many labs and projects, so we had to adopt processes that would support good communication across the team, rapid development and delivery, and project management to coordinate development and manage dependencies. These requirements were best met using ideas within the agile methodology.¹⁵ As in any team, we adopted, altered, and sometimes discarded processes. While we continue to refine our processes, we currently use a number of ideas to maintain communication:

- *daily “stand up” meetings* ensure that team members stay in contact with each other, help each other through issues, and discuss new projects;
- *iteration planning meetings* offer formal discussions about features, dependencies, requirements, and time scales for the next (short) development period; and
- *pair programming* ensures that each developer shares in new project designs and implementation decisions as well as providing a method for knowledge transfer.

Our 10-member team is both larger than ISB's typical development team (which has one to three members) and is involved with many projects across the institute, so we need high-level project management to ensure cohesiveness of both the team and the code development (see Figures 1 and 2). In typical scientific software development, project management has proven difficult to justify. Most scientists don't value software development or understand the process to begin with, and thus adding a position that doesn't directly involve writing code makes little sense to them. However, with project management our informatics team has managed to produce more software of a higher standard, because we can quickly identify commonalities across software applications, agree on milestones, and rapidly reassign team members to priority areas.

The biggest difficulty in creating this team has been in finding developers who are suited to working in the chaotic life science research environment. When we added developers to the team who were used to being isolated and independent, they were generally frustrated by the processes and formalized environment. This is understandable; such developers might suddenly feel limited by processes that require greater discipline in tool development. Similarly, developers who were more comfortable within large industry teams—where expectations and projects are better defined—found the fairly loose and constantly changing environment of scientific development too stressful. Among the causes of such stress are few requirements, little understanding of what users want, and a need to deliver usable tools immediately. In the end, we found that -exible developers who can work at all process levels and are comfortable with direct user interactions have adapted most readily to the research informatics environment.

Communicating with Scientists

Any software development process created for use in a scientific environment must recognize that success is measured by the metrics of scientific advancement. This means that any process employed requires efforts to formally present the work at scientific seminars and conferences, write papers for science journals, and write grants to ensure the team's continued viability and obtain sponsorship. All such tasks require that the work presented offer new features or a new application of software ideas.

Most developers working in domains other than science are rarely required to communicate their ideas or work outside the team until they take a management or team-lead role. Development in the scientific environment requires that every developer be able to propose, present, and write about his or her software to scientists (and other software developers). Our informatics team created a two-part system that helps developers within the team and across the institute communicate with the scientists:

- *Persistent communication.* With each developer working on different projects (often without formal specification or many requirements), we need some persistent documentation. We therefore use a wiki and project-tracking system to communicate within the team as well as to provide documentation to serve as a reference for papers, grant authoring, and grant reporting.

- *Formal seminars.* Teaching software developers the methods scientists use to report progress to each other requires that they describe their work in seminar-style sessions. We use these seminars to not only present novel work, but also to share ideas for future development and discuss interesting new technologies or standards. The seminars also help address isolation issues associated with individual developers.

Software developers in any domain must learn to interact with their users. In the research environment, this communication must occur through venues that scientists are most comfortable with. So, while developers must first provide software to the scientists, they must also communicate about it through formal presentations and journals. By learning to use these research science methods, developers can help change scientists' perception, of software development as a risky endeavor to support, to seeing it as an integral and necessary part of their research.

Rapid Development of scientific Applications

Technologies that rely on well-defined domain-specific models are inappropriate in the life sciences. Core concepts, such as the gene, are often context specific and difficult to model, so attempts to impose standards have largely failed. Providing easy access to the data is more imperative than providing a developer-focused, model-driven system. To create ease of access, we must provide tools that scientists are comfortable with and that give them the required -exibility to do their work. Scientists (and most end users) are generally more comfortable handling document-based data because

- they're open and readable, making them easy to share;
- access is natural and -exible, so documents can be saved and retrieved from the file system using standard desktop tools; and
- information can be retrieved through various mechanisms (such as email and Web pages).¹⁶

Using documents (instead of objects) makes it relatively simple to provide document transformations to support collaboration and -exible data delivery. Thus, the data modeling that occurs is at the level of the "boundary objects,"¹⁷ where the user is most familiar with the data. Scientists know what they need from the document and how they want to model it, so enabling them to control the model means that they can manage change conveniently (compared to changing an entire system's domain model). Confining domain models to client-level translations also leads to the use of technologies and patterns that we can provide in a distributed and loosely coupled manner. To create the basic services for data persistence, access, and annotation, we use easily accessible protocols such as HTTP, Representational State Transfer (REST) and JavaScript Object Notation (JSON). To quickly create complex services (such as image analysis and network visualization), we layer simple data-access and translation services.

New technologies, data, and analyses are introduced continually in the life sciences. Keeping up with the science requires that the software team's top goal be rapid development. Where possible, our informatics team works to incorporate the standard principles of good software design (such as extensible code, layering concerns, and reusable libraries), but the goal of providing and deploying usable code necessitates several requirements:

- *Support ad-hoc and end-user development.* Requirements are often vague for research software and new functionality might be required on short notice. To ensure we can provide software rapidly, it must be -exible and thus easily modified when requirements change. Content-management systems are a good example here;

a highly -exible schema lets us quickly alter data storage or access to provide new functionality.

- *Ensure ease of use for scientists.* The principles of any system developed in the research environment must be simple so that end users and developers can quickly learn and use the system. Again, content-management systems provide a good example: Users can understand the system's principles by comparing it to a file system, which additionally supports data annotation.
- *Support use for multidisciplinary research.* Supporting various users in a research environment (such as bioinformaticians, software developers, and laboratory scientists) requires that any system developed provide a quick, nonintrusive means of integrating analysis tools and data. Thus, any language or client should have convenient and robust system access. By providing core data management (such as persistence and access) and useful horizontal services (such as identity mapping and annotation) through easily accessible protocols, developers can quickly add new functionality.

To support rapid development within the confines of research funding, it's essential to use appropriate open source technologies. Figure 3 shows an example of one such system. The Adaptive Data Management (Addama) service architecture⁹ uses standards such as

- the Java Content Repository specification, which generically stores experimental information;
- Web technologies, such as REST and JSON, to provide interoperability across languages;
- service registries, such as Galaxy (<http://mulesource.org>), to allow for runtime discovery of available services (including information indexing or search capabilities); and
- data/metadata identification using the Uniform Resource Identifier standard.

There are numerous other technologies that developers can adapt to solve software needs in the life sciences, from data management to data annotation. Open source technologies also offer valuable access to user and developer communities.

The informatics team at the ISB has worked to find the middle ground between top-down, generic modeling and small-scale, lab-based computing. Our infrastructure supports large-scale data production and integration across a variety of data, while also encouraging and supporting ad-hoc development for specific experiments. In addition, we use a mix of formal and mainly agile processes to help foster rapid development and delivery, as well as to support a level of team communication uncommon in most life sciences development.

Our design principles provide a support infrastructure by using document-based data and just-in-time client-defined models that let developers provide usable software without prespecifying the domain concepts. These principles also emphasize the use of simple, loosely coupled services connected through easily accessible protocols to ensure support for ad-hoc development. The goal of these principles is to support software developers in rapidly creating new tools and services, as well as supporting end-user development by bioinformaticians and computational biologists. Our principles and processes also assist developers in communicating about these tools with each other and their end users (scientists).

The challenges we present here have led to a culture in which scientists and software developers have failed to communicate. The processes we adopted to meet these challenges

were born of experience from several life science academic institutions and commercial organizations. These experiences have limited applicability to other sciences, as each discipline has its own nuances that have arisen from a complex set of factors (such as the discipline's history, training methods, funding limitations and priorities, and experimental methods). However, the focus within the life sciences on rapid innovation is paramount because innovation leads to important discoveries, including new therapeutics. Any candidate software process must clearly demonstrate that it can enhance science by enabling the expedient delivery of necessary tools.

Acknowledgments

This work was supported by a National Institute of General Medical Sciences grant (P50GM076547), the National Institute of Allergy and Infectious Diseases NIH contract HHSN272200700038C, and a National Cancer Institute grant (R01-1CA1374422). The content is solely our responsibility and doesn't necessarily represent official NIH views. We thank David Burdick, Christopher Cavnor, Hector Rovira, and Ilya Shmulevich for providing assistance and invaluable advice in developing these processes.

Biography

Sarah Killcoyne is the project manager of the informatics team tasked with developing data management systems, integration architecture, and general analysis tools under John Boyle at the Institute for Systems Biology. She also develops and supports the Cytoscape software for the ISB. Killcoyne has a BSc in biological sciences from Colorado State University. Contact her at skillcoyne@systemsbiology.org.

John Boyle is the director of the Informatics Core at the Institute for Systems Biology. His research interests are in information management and visualization. Boyle has a PhD in computing science and molecular biology from the University of Aberdeen. Contact him at jboyle@systemsbiology.org.

References

1. Scaffidi CSM, Myers B. Estimating the Numbers of End Users and End User Programmers. Proc. 2005 IEEE Symp. Visual Languages and Human-Centric Computing, IEEE CS Press 2005:207–214.
2. Hannay, EJ., et al. Proc. 2009 ICSE Workshop on Software Eng. Computational Science and Eng. IEEE CS Press; 2009. How Do Scientists Develop and Use Scientific Software?; p. 1-8.
3. Segal J. When Software Engineers Met Research Scientists: A Case Study. Empirical Software Eng 2005;10(4):517–536.
4. Wilson G. Where's the Bottleneck in Scientific Computing? Am. Scientist 2006;94(1):5.
5. Kelly D. A Software Chasm: Software Engineering and Scientific Computing. IEEE Software 2007;24(6):120–119.
6. Segal, J. tech. report no. 2004/25. Computing Dept., Open Univ.; UK: 2004. Professional End User Developers and Software Development Knowledge.
7. Segal, J. Proc. 1st Workshop End-User Software Eng. ACM Press; 2005. Workshop on End-User Software Engineering; p. 698
8. Star, SL.; Ruhleder, K. Proc. ACM Conf. Computer-Supported Cooperative Work. ACM Press; 1994. Steps Towards and Ecology of Infrastructure; p. 253-264.
9. Boyle, J., et al. Adaptable Data Management for Systems Biology Investigations; BMC Bioinformatics. p. 2009www.biomedcentral.com/1471-2105/10/79
10. Boyle J. Gene-Expression Omnibus Integration and Clustering Tools in SeqExpress. Bioinformatics 2005;21(10):2550–2551. [PubMed: 15746290]
11. Pavlopoulos, G.; Wegener, A.; Schneider, R. A Survey of Visualization Tools for Biological Network Analysis; BioData Mining. p. 2008www.biodatamining.org/content/1/1/12

12. Johnson DS, et al. Genome-Wide Mapping of In-Vivo Protein-DNA Interactions. *Science* 2007;316(5830):1497–1502. [PubMed: 17540862]
13. Schwanhäusser B, et al. Global Analysis of Cellular Protein Translation by Pulsed Silac. *Proteomics* 2009;9(1):205–209. [PubMed: 19053139]
14. The Encode Project Consortium. The Encode (ENCyclopedia Of DNA Elements) Project. *Science* 2004;306(5696):636–640. [PubMed: 15499007]
15. Beck, K., et al. Manifesto for Agile Software Development. 2001. www.agilemanifesto.org
16. Boyle, J., et al. Systems Biology Driven Software Design for the Research Enterprise; *BMC Bioinformatics* vol. p. 2008 www.biomedcentral.com/1471-2105/9/295
17. Star SL, Griesemser JR. Institutional Ecology, ‘Translations’ and Boundary Objects: Amateurs and Professionals in Berkeley’s Museum of Vertebrate Zoology, 1907–39. *Social Studies Science* 1989;19(3):387–420.

Figure 1.

The Institute for Systems Biology informatics context. The ISB's high-throughput data-production facilities include genomics, proteomics, imaging, and microfluidics. The informatics team interfaces with each facility, providing data management tools to capture data and associated metadata. Computational biologists use these services to access, analyze, and annotate data to support the ISB's ultimate goal of improving human health.

Figure 2. Informatics team services. The team provides three services to software projects across ISB: a set of enterprise-level projects and services (such as data management, security, and sample information management), collaboration across projects to ensure reusability and reliability of integrated services and projects, and application of good programming practices through project management.

Figure 3.

The Addama service architecture. The architecture offers rapid integration and development of formalized services, ad-hoc services, analysis pipelines, and data repositories.⁹ Using Representational State Transfer (REST) framework, laboratories and users can quickly transform data to suit their specific needs.

Table 1

The evolution of solutions for integrating biological data, knowledge, and tools.

Year	System	Technology	Generation
1996	Sequence Retrieval System (Lion Biosciences)	External indexing of flat files	Database
1997	Discovery Center (Netgenics)	Corba-based components	Integration frameworks
1998	Alliance (Synomics)	Distributed application server	Integration frameworks
1999	MetaLayer (Tripos)	XML message passing	Integration frameworks
2000	DiscoveryLink (IBM)	Federated database solution	Integration frameworks
2001	Genomics Knowledge Platform (Acero)	Enterprise JavaBeans-based object interaction	Integration frameworks
2002	Life Sciences Platform (Oracle)	Embedded Web services	Stateless Web services
2003	myGrid (EPSRC)	Ontology-driven services	Stateless Web services
2004	caBIG (National Cancer Institute)	MDA-based architecture	Stateless Web services
2005	BioMoby	Registry and Semantic Web	Document-based
2006	CancerGrid (Medical Research Council)	Resources using Web services	Document-based
2007	caGrid (National Cancer Institute)	Web service/registry solution	Document-based
2008	Amalga (Microsoft)	XML document warehouse	Document-based