Is Evolution a Software Engineer? 
A Case-based Comparative Analysis of Biological and Software Systems

Fahad Khalid * Diana LaScala-Gruenewald † Ana María Gomez Lopez ‡ Renske Vroomans § Stojan Davidovic ¶ and Zhi Qiao ¶

* Hasso Plattner Institute for Software Systems Engineering, † Hopkins Marine Station of Stanford University, ‡ Yale University, § Utrecht University, ¶ Max Planck Institute for Human Development, and ¶ National University of Singapore

Biological evolution and software design are commonly considered separate processes, so similarities between the two tend to be overlooked. While software design changes tend to be more intentional than those produced by the random walk of evolution, the structural and functional responses of complex evolved and designed systems to a dynamic competitive environment exhibit important similarities. Here, we use a novel framework to examine a biological complex system (Hox genes) and a complex system in software engineering (Online Charging System). We discuss the similarities and differences in the processes of innovation, conservation and evolvability in each system using our novel ICE framework, discuss the theoretical underpinnings of these similarities, and consider the potential of these observations to influence work in both fields.

Software  |  Hox genes  |  Innovation  |  Evolution  |  Design

Introduction

Biological systems have inspired algorithm design since the inception of computer science. Since that time, biologically-inspired computing has made steady progress, suggesting the significant impact biological ideas have had on the creativity and productivity of the field. Genetic algorithms [1], genetic programming [2, 3, 4], ant colony optimization [5], and particle swarm optimization [6] are some examples of algorithms whose designs are directly based on observations in biology. The development of Artificial Neural Networks [7] provides another example; this algorithm was the result of research carried out to develop mathematical models of neuronal function in the brain [8].

An even more interesting case of the interplay between biology and computer science was the development of self-reproducing automata [9]: a theoretical model of self-reproducing machines developed to understand biological evolution and self-reproduction. Even though this theory was developed before the mechanisms of DNA replication were discovered, interpretation and copying of information by the self-reproducing machine is similar to the use of DNA in cellular reproduction. This example provides crucial evidence that an artificial system that is functionally similar to a biological system can help clarify the mechanisms employed by nature 1.

Recent advancements in the field of computational biology have made computer science an invaluable tool in studying biological systems. One example is the recent application of “automated theorem proving” to enhance understanding of pluripotency in mouse embryonic stem cells [10]. In numerous other cases, traditional computational tools, such as programming languages and compilers, are used by biologists to develop simulations.

The above discussion establishes the existence of the fertile interplay between biology and computer science. Nevertheless, it is surprisingly difficult to find evidence of a similar exchange of information between the fields of software engineering and biology. Software engineering deals with the processes and design principles involved in the development of software systems. Large-scale software systems are inherently complex, and their successful life cycle depends on the design and process guidelines provided by principles of software engineering. In this case, the emphasis is not on algorithms, but design strategies that render the software system maintainable and evolvable over time.

In our previous work, we have identified novel similarities 2 between design principles in software systems and evolved biological systems. These similarities exist not because the design principles in software engineering are biologically inspired, but because they happen to be similar to certain strategies employed in biological systems. The Appendix contains a detailed account of these similarities. Here we summarize important points:

• Encapsulation and Decoupling: Encapsulation is one of the most important design principles in object-oriented software engineering, and means that the inner state of any object should be protected from the outside environment; any outside access should be possible only through a carefully designed interface. Decoupling complements encapsulation by requiring that the inner function of any object or module should be independent from the interface used to access it. In living systems, cells incorporate both these features. Transmembrane receptors, like G protein-coupled receptors (GPCRs), serve as an interface that allows the extracellular environment to communicate with the cytoplasmic environment, which is encapsulated by the cell membrane.

• Memory management vs. Apoptosis: In low level programming languages like C++, the programmer must explicitly specify the allocation and deallocation of memory for certain types of variables. A memory allocation for which a corresponding deallocation instruction is not implemented can cause a memory leak. This can eventually result in program failure. Analogously, in biological systems, cell death should be triggered by the apoptosis pathway. A cell that dies prematurely due to infection or some other trauma (necrosis) can cause local or even systemic damage.

• Garbage Collection and Proteolysis: As opposed to C++, certain high level languages such as Java provide automated memory management: The language runtime automatically deallocates memory that is no longer in use. The mechanism of tagging memory locations for deallocation and processing them is analogous to proteolysis in living cells. In proteolysis, proteins inside the cell that need

---

1 To the best of our knowledge, the discovery of DNA replication mechanisms was not inspired by the theory of Self-reproducing automata.

2 These similarities were identified by Fahad Khalid in his unpublished research work on the subject, prior to the Complex Systems Summer School.
to be recycled are first tagged, and then brought to the proteasome where they are broken down into constituent amino acids.

The above mentioned biological and software processes have some striking similarities. This raises the question of why design-level similarities exist in these disparate systems. In order to answer this question, we believe it is imperative that we develop a thorough understanding of processes that guide design in both biological and software systems.

Fundamental differences exist between the processes of evolution and design [11, 12]. A system that evolves does so organically in nature. A designed system, on the other hand, is developed with intention, usually by an engineer who aims to artificially create new systems. Because of this key distinction, the similarities between evolved and designed systems are often overlooked [13]. However, upon examination of the structure and function of biological and software systems, important parallels can be found (as presented above). In order to understand why similarities and differences exist between biological and software systems, we need to understand how a (natural or artificial) system develops over time from inception to extinction.

In this paper, we consider these similarities in the context of our novel Innovation, Conservation, and Evolvability (or ICE) framework. We apply the framework to a case study which we believe illuminates the usefulness both of our comparisons and of this framework. We then consider the mathematical underpinnings of observed similarities, by comparing evolution to the problem of optimization over a dynamic landscape. Finally, we discuss the more general questions raised by our research and how we intend to tackle these in future work.

Our Contribution

- Even though much research has been done identifying similarities between innovation in biological and technological systems, to the best of our knowledge, we present the first detailed comparison between biological and software systems.
- We identify and investigate similarities in design principles between disparate systems.
- We introduce the ICE framework, which incorporates a three-pronged approach to identifying key relationships between disparate systems, and can be used as the basis for subsequent quantitative comparison.
- We present a novel case-based approach to studying the processes of evolution and design.
- We propose foundations for a mathematical framework that could potentially serve as a novel computational approach for studying the process of evolution.

ICE Framework

Both biological and software systems must innovate to survive in a competitive environment without compromising the essential characteristics that allow them to be viable and evolvable [14]. Organisms and species face competition from other biological entities and pressures from the natural environment, while the market economy and technological trends generate similar pressures on developing software systems. Both must innovate to gain fitness in an ever-shifting competitive landscape. However, some changes in both can be detrimental. For example, an organism that has a harmful mutation in its genes may fail to produce viable offspring. Similarly, a software system may add an inferior feature, or a feature that is not useable or well-liked by consumers.

The Innovation, Conservation, and Evolvability (ICE) framework provides a useful lens for examining the similarities between complex systems that adapt to competitive environments. In this framework, the fundamental unit of interest is a particular software program or biological species. It is the program or species that can change through time in response to changes in a competitive environment.

The ICE framework is based on three fundamental pillars that characterize both biological and software systems.

**Innovation.** Innovation is the process by which a novel feature is added to a biological or software system. In the former, innovation can encompass a new morphological trait – like an additional limb, new cell type or novel protein – or a change in regulatory mechanisms. For example, a transcription factor (a protein that binds to a segment of DNA and controls its expression) might bind to a new DNA location, changing the regulation of certain genes.

In software systems innovation encompasses new capabilities of the software. An addition to the original software might allow it to interface with a particular application, or perform a new task.

**Conservation.** Conservation encompasses the constraints that govern both biological and software systems. In an organism, a new feature must not hinder development or reproduction. As a result of this pressure, features that work well tend to be passed on from generation to generation. Similarly, software must maintain certain features as new ones are developed. For example, customers expect certain interfaces and functions that must continue to operate as new features are added.

**Evolvability.** Innovation and conservation combine to create evolvable systems. Evolvable systems have the ability to acquire new features without fatally hindering the functionality or structure of the system. How evolvable a system is influences its ability to survive and thrive in a competitive environment, despite constant changes.

In the following Sections, we use the ICE framework to examine a case study of a specific biological and software system. We use Hox genes and online charging systems as examples to illustrate the usefulness of the ICE framework, the similarities between software and biological systems, and the potential for this process to illuminate opportunities for growth in both fields.

**Case Study: Hox Genes**

All animals share a genetic basis for an anterior-posterior body axis, or a head-to-tail “body plan”, that stems from a common ancestor [15, 16]. These instructions, which produce segmented body morphology, symmetry and axis formation in embryonic development, are encoded in Homeobox (or Hox) genes. All Hox genes are highly conserved throughout the animal kingdom: In fact, it was the transposability of Homeobox eye genes between fruit flies and mice that proved both the existence and conserved nature of these genetic sequences [17]. However the numbers of Hox genes and groups of related genes referred to as Hox gene “clusters” vary among different animal phyla [15, 18]. Mice (Chordata), for example, have four Hox clusters, but insects (Arthropoda) have only two.

Hox genes are considered crucial in the development of new body plans [16]. Mutations in these genes and in the regulatory networks that control their expression are thought to
have played a role in the development of the complex brain, mineralized bones and teeth, differentiated vertebral column, hinged jaws, and paired appendages of vertebrates [19]. While Hox genes are subject to intense selectional constraints, they also are likely to facilitate new interactions within the genome and therefore have a high potential for evolutionary innovation [15, 18]. For these reasons, Hox genes provide an excellent opportunity to examine the ICE framework in a biological context.

**Innovation and Conservation in Hox genes.**

**Conserved genetic content and gene order**

With some exceptions, basic animal body plans have remained remarkably similar during the last 550 million years [20]. To explain this, Davidson and Erwin have proposed the concept of gene “kernels”: highly conserved gene regulatory networks (GRNs) which have been essentially unaltered since the Cambrian Explosion, and which specify where in an organism each body part will form. When gene kernels fail, catastrophic development often takes place: An organism fails to develop a limb or other crucial body part. According to Davidson and Erwin, many Hox genes are part of genetic networks called “I/O switches”, which regulate how long a genetic circuit is turned on [20, 21]. For example, they may control the relative size of homologous body parts. While I/O switches are not as conserved as gene kernels, they are essential to proper axis formation within an organism, and can cause significant changes – both positive and negative – in body structure and fitness.

Aside from these regulating mechanisms, Hox genes also exhibit conserved physical order within a genome. Their spatial placement within a cluster matches the anterior-posterior axis that emerges during embryological development [15]. This phenomenon, called “collinearity”, appears in species higher than insects [15].

**Innovation by duplication and exaptation**

Although contemporary Hox gene clusters are thought to have descended from a single founding gene and are highly conserved, there have been at least two full-genome duplications over the course of vertebrate evolution [22]. The first occurred in a common ancestor shared by all jawed and jawless vertebrates after the divergence of Cephalochordates (Amphioxus), and the second occurred in the common ancestor of all jawed vertebrates, after divergence from jawless vertebrates [23]. Both of these duplications allowed for changes within the gene copies, as well as within their regulatory mechanisms.

Several experts have proposed a genetic duplication-differentiation mechanism to explain these innovations within such highly conserved genes [22, 24, 25, 18, 26]. Duplicated genes are able to diverge from the original copy unchecked by direct selection [22, 24]. These genes either become pseudogenes through the accumulation of deleterious mutations, enhance the original through increased gene dosage, or acquire new functions through changes within the gene sequence or regulatory network [18].

Additionally, duplication is thought to be the only type of mutation which increases robustness, or the ability of an organism to withstand deleterious mutations [19]. Removing a gene temporarily from selection pressure and allowing it to explore genotypic and phenotypic space in a robust way results in greater opportunities to develop new morphological features without facing catastrophic development or poor fitness. Thus, in an evolutionary context, the duplication-differentiation model allows major body plan innovations without violating the constraints that keep organisms and species viable in a competitive environment.

Occasionally, duplication and differentiation results in novel genes which do not retain the homeotic function of their progenitors. One well-studied example is *Fushi tarazu* (*ftz*) in fruit flies (Drosophila) [18]. *Ftz* is thought to establish pattern segments in the early embryo and aid in specifying neuronal types in the central nervous system [27, 28]. It is located between two canonical Hox genes, and its sequence retains similarities to the Hox genes which define the trunk segment of flies [29]. However, its novel function in the central nervous system is divergent, and requires different regulatory elements [18].

This example illustrates a second innovative mechanism called “exaptation”. Exaptation is a term used to differentiate a feature that was produced by natural selection for a particular function (an adaption) from one that has acquired some other function for which it was not initially selected [30]. This concept of a genetic element being co-opted for a novel function is key in biological innovation.

**Evolvability in Hox genes**

A biological system is evolvable if it can produce non-lethal, heritable phenotypic variation through mutations [19]. In other words, genetic changes in an evolvable organism must allow that organism to live, reproduce, and pass on changes to its offspring. More broadly, the concept of evolvability incorporates the idea that an organism can be fit enough to survive in a competitive environment without being what a software engineer might consider “efficient”. It is the ability of an organism to innovate and compete over evolutionary time.

Factors that contribute to evolvability remain an ongoing subject of biological research. Within the ICE framework, evolvability is a balance which results from innovation constrained by conservation. **But what are the biological mechanisms that allow this balance to be achieved?** Below, we discuss two current hypotheses of structural characteristics in biological systems which may contribute to their potential for evolvability.

**Neutral Networks**

According to the neutral theory of evolution, most mutations have no effect on phenotype, and only a small fraction of non-neutral mutations are beneficial [31]. Neutral mutations are mostly a product of genotypic redundancy: one phenotype often corresponds to many different genotypes. Thus, an organism or species can undergo many mutations without changing its phenotype.

This genotypic redundancy expands the number of phenotypes available to a group of related organisms. As the species traverses a neutral network – the network of neutral mutations that connect genotypes and phenotypes – through evolutionary time, it encounters more beneficial phenotypes than it would if no neutral mutations were possible [32]. Thus, genotypic redundancy and the presence of neutral networks enhances the evolvability of an organism or species.

For more information on neutral networks and their role in evolvability, please see the subsection: “Innovation as an Adaptive Optimization Process.”

**Modularity**

Philosopher of science Kim Sterelyny has described evolvability as “the space of evolutionary possibility where lineages
have access" [33]. The evolutionary exploration of genotypic and phenotypic possibility in Hox genes resulted in an ordered arrangement of distinct features within organisms such that no one change disrupted the entire system. This idea was identified by Kirschner and Gerhart [34] as “compartmentation”, which they considered to be an important and widely recognized strategy for evolvability. Compartmentation is analogous to “modularity” [35], which constitutes a group of biological elements “whose intrinsic behaviors and functional interactions yield a mechanistic explanation of an identifiable developmental process or transformation” [36]. In other words, modularity allows the developmental processes of one part of the body to dissociate from the other parts [37]. In this way, it may promote greater size, efficiency, complexity and adaptability in organisms.

Modularity of morphological characteristics has long been discussed in biology. While there are obvious modules such as the cell and the organ within an organism, it is more difficult to clearly identify modular components at larger scales of organization, such as species or populations [38]. One unifying principle that defines modularity is that of connectivity [39]. Indeed, one expansive definition of modularity defines it as tightly connected subsets loosely connected to other subsets [19]. In examining these connections between subsets or modules, many scientists have drawn an analogy between inter-modular connections and electrical wiring (for a recent example, see [40]). In this scenario, a connection between two modules has some intrinsic benefit to the larger system, but it also has a physical, metabolic, or developmental cost, called a “wiring cost”.

Recent research suggests that the evolution of modularity in living organisms may be a means to save on wiring costs [41]. Jeff Clune and his collaborators simulated 25,000 generations of in silico evolution, and concluded that maximizing network performance and minimizing connection costs yields networks that are significantly more modular than control experiments [41]. This supports the hypothesis that modularity is caused by cost-reducing selection, and suggests that evolution favors modular designs because they require fewer and shorter network connections.

Case Study: Online Charging System

In this section, we describe the evolution of a software product using the Online Charging System (OCS) software as a case study. Moreover, we refer to the ICE framework in order to compare major characteristics of software product evolution to evolution in biological systems.

Background concepts in software engineering. The discussion of the case study requires the use of certain concepts from the field of software engineering. In this section, we build the necessary background by briefly describing the concept of Software Requirements Engineering [42]. Evolution of any commercial software product is primarily driven by market requirements. The following are the three major channels through which requirements enter the product engineering process.

Market-driven

In this case, the entire potential market for the software system is studied as a whole to decide how to take the product evolution forward. No single customer dictates the requirements. The software company’s Research and Development (R&D) department generates new feature ideas based on the overall market environment and technology trends. A typical example is Commercial Off The Shelf (COTS) products such as office automation tools and desktop operating systems.

Bespoke

A particular customer requests a specific feature, which leads to its subsequent development. Many software systems are custom-tailored for one specific customer, and are never sold to any other customer.

Hybrid

Product evolution is driven by considering the market situation, as well as specific customer requests. The product is typically divided into core and customizations. Each customer buys the core and can either purchase one or more existing customizations, or ask for new customized features to be developed. Certain customer-specific features become core product features over time.

Introduction to Online Charging Systems. The Online Charging System (OCS) is a software system responsible for real-time charging [43] in cellular telecommunications networks. It is one of the many systems in control of processing calls. The OCS is primarily responsible for checking the subscriber’s account balance before and during the calls, as well as deducting from the balance the cost of the call. Cellular service providers’ design tariffs, which are often complex algorithms that define how subscribers are charged for calls. Also, special tariffs are defined for special offerings: cheaper calls during Christmas holidays, for example. These tariff algorithms are implemented in the OCS. Consequently, the OCS plays a critical role in revenue management for a cellular service provider.

The motivation behind using the OCS in concert with the Hox gene case study, instead of any commonly used desktop software, is as follows:

- The OCS product discussed here is based on the hybrid requirements channel, which gives us a broader view of the impact of selective pressure in software product evolution.
- OCS is a critical part of the cellular network, and therefore puts increasing demands on the product development process. This necessitates adherence to best practices in software engineering, which makes the case study representative of the state-of-the-art in this field.
- Typical code base for the OCS can include millions of lines of code, with complex interactions between different functions. Therefore, a careful design and verification process is required for the evolution and maintenance of the product. As with most top-of-the-line products, the OCS development process must employ the best available software development and management processes.

For the purposes of this paper, we assume that the OCS product has already been developed, and is constantly being updated with new features. In the following subsections, we present the evolution of the OCS product through the addition of new features. The process is presented in the context of the ICE framework, with direct comparisons of the characteristics of evolution in Hox genes.

Innovation and Conservation in Online Charging Systems.
Feature success and prevalence

For market requirements that originate with a particular customer (a cellular service provider), the decision of which features should be incorporated into a software and why is driven by the anticipated demands or niches in the end-user/subscriber market. Once the feature is delivered to a service provider, they must determine whether the offerings based on the features meet subscriber demands or market niches. The success of the offerings might not intuitively appear to have an impact on product evolution, but sometimes it does.

A feature that leads to successful offerings has a greater chance of being improved and expanded over time. After its initial success, a customer is likely to use the same feature to develop more offerings. This process can then lead the customer to expand the feature, which results in new requirements that fuel the product development process. Therefore, over time, a new feature can become an integral part of product evolution.

Highly successful features can also attract other service providers. In most cases, these features are utilized by all customers for relatively similar applications. However, in some cases, the same feature can be used in diverse scenarios by different service providers. Novel applications of a feature can further strengthen its place in product evolution. Certain highly successful features tend to become core product features over time.

Conservation of basic function

Regardless of how many new features are progressively added to a product, there is a basic function of the OCS that is conserved over the entire product life cycle: real-time call charging. For example, a product could have started with support for voice calls only, and proceeded to include Short Message Service (SMS) and Internet support afterwards. Fundamentally, however, the OCS continues to perform call charging. This basic function defines the identity of the product, and its placement in the market.

Moreover, certain product features can become building blocks for the development of others in the future. Such a foundational feature is also conserved, since it constitutes a fundamental building block in product evolution. An example of such a feature in the OCS is the ability to define customized tariff plans using a Graphical User Interface (GUI). Before this feature was introduced, a customer had to contact a vendor to implement a given tariff plan – a very time-consuming and expensive process. When service providers can independently define as many tariff plans as needed, the time to market is significantly reduced. This has led to the addition of hundreds of new features to the GUI, and the corresponding back-end.

This process of conservation of basic OCS function (real-time call charging) is analogous to the conservation of Hox gene kernels. In both cases, the core function is conserved, and serves as a foundation for further innovation.

Innovation with duplication

An OCS is expected to be in service twenty four hours a day, seven days a week, without disruption. This is important so that calls can be made anytime and are never interrupted by system failure.

Prior to being introduced into a live system, a new feature must be tested to ensure it does not adversely affect the system function. This is done by installing the new feature on the staging environment, a replica of the live system which processes all traffic alongside the original. Results of the duplicated processing are not fed into the live environment; call processing on the staging environment cannot affect live calls. The purpose of using the staging system is to observe how real traffic is being processed by the upgraded duplicate of the system that includes the new feature. Operations staff closely watch the system behavior for a certain period of time, and only approve migration to the live system if no problems are observed. This procedure guarantees the proper functioning of the system, considerably decreasing the risk of a negative secondary effect.

Hox gene clusters can be considered analogous to the staging environment: both provide replicated functions, where one copy can be used as test bed for innovation. The innovation is integrated into the main system if it provides a beneficial outcome. Therefore, innovation is encouraged and the vital system function is not compromised.

Exaptation

In certain cases, a service provider requests an OCS feature for a specific purpose. This service provider deploys the feature in a specific market, where it succeeds as expected, but no extraordinary results are obtained. At a later point in time, the feature is sold to a different service provider, who decides to use it in a completely different context, and in a different market. This results in extraordinary success of the feature, and it raises the significance of the feature and as a result its contribution to the future evolution of the OCS product. (Note: Due to non-disclosure agreements with our industry sources, we use the Erlang programming language to illustrate the concept of exaptation, rather than providing an example of an OCS feature.)

The Erlang programming language was specifically designed for programming highly concurrent telecommunication software products at Ericsson. Initially, it was only used within Ericsson for their product development. Over time, however, the language became popular with other telecommunications software product vendors, and ultimately found its way to very different business domains, like Facebook and Amazon. These transitions occurred because although the product was developed for a specific domain, it had certain core characteristics that were useful for several other domains as well: the ability to program highly concurrent systems with minimal effort.

Evolvability in Online Charging Systems.

Modularity

A large-scale software system like the OCS typically comprises several independent components, which interact with each other via well-designed interfaces. A change in one component can propagate to others only if the interface has changed as well. Internal changes can be completely localized, and need not produce any harmful side-effects.

The same principle of modularity holds for functions and subroutines implemented inside a component. Function A calling function B only needs to know the signature of function B. The algorithm implemented inside function B may change to some extent, for example it may become more efficient, but this will have no effect on function A, provided that the signature does not change. Modularity allows localized changes, thus improving manageability and efficiency of the development and maintenance process. In addition, the advantage of writing modular software is increased reusability. Once a commonly used function or object matures in terms of im-
Environmental stochasticity and frozen accidents

Although market success can result in the conservation of a feature, this condition does not guarantee long-term prevalence of the feature within the product. It could be the case that a feature that is useful in a very specific market has no general application. If the customer is not considered important enough for the OCS product vendor, there is no incentive to make the feature part of the product core. Moreover, certain features prevail despite their obvious limitations. One example is Short Messaging Service (SMS), which only allows 160 character long messages – a length constraint that would be untenable for e-mails. Yet SMS remains a popular service, with numerous tariff related features in the OCS.

It is important to note here that product innovation and evolvability is also often shaped in part by random influences from either the market or technology. Software products that are well adapted to such environmental stochasticity may be more evolvable than others.

The OCS is a highly concurrent system that generally handles workloads on the order of hundreds of calls per second. To be able to efficiently process these volumes, the OCS requires execution on multi-processor systems. Therefore, concurrent programming is essential for OCS development. Let us consider a hypothetical scenario in which an OCS product starts out as a small project, but very quickly acquires a large market share. Such rapid success was not anticipated by the project's developers.

The code base expands rapidly as the product matures, and eventually reaches a size which makes it clear that Java was not the most suitable programming language for such a highly concurrent software system. The Erlang language would have been a better choice. In this example, the choice of Java as the programming language for the OCS product is considered a frozen accident.

The maturation of a product drastically increases the cost of porting an entire system to a different programming language. First, the sizeable effort incurs a financial cost to justify. A second cost is that a version control system must be coordinated. A developer generally saves his or her daily work to the private branch, which is a copy of the feature branch that is directly accessible to the development team. Since there can be multiple teams working on the same feature, it is safer for each team to have a separate team branch.

Each developer within the team is then assigned a private branch, which is a copy of the corresponding team branch. A developer generally saves his or her daily work to the private branch. Merging down the branch hierarchy are must passes all new tests as well as all regression tests, the new tests are also added to the regression test suite.
larities between innovation in software and biological systems. Here we hypothesize that these similarities are not coincidental, but instead emerge because the underlying mechanisms that drive or hinder innovation are the same. Furthermore, we conjecture that the underlying mathematical problem is “optimization over a dynamic landscape”, and present a theoretical framework to substantiate this idea.

In order for a system to be able to evolve (or change), there should be a finite probability that a change made to the system is beneficial [45]. If this probability is too low, the system should always stay the same, and is therefore not evolvable. If the system is to be evolvable, the underlying instructions (the genetic or computer source code) should be structured such that beneficial changes are within easy reach.

In biology, the solution to this problem lies in the way the genetic code (genotype) is translated to an outcome (phenotype) and how different codes lead to different or similar phenotypes. This is called the genotype-phenotype map. The shape of the landscape formed by this map determines the number of beneficial, neutral, and deleterious phenotypes in the neighborhood of any organism. It has been shown that in this landscape, there are vast neutral networks connecting genotypes that yield the same phenotype. These neutral networks allow species to explore the genotype space over time and sample different phenotypes along the way, all while conserving the phenotype on the neutral path. This is part of what makes a biological system evolvable.

“However, the mapping from genotype to phenotype is not static; it is also under the influence of the genotype. In other words, the system can change the probability of a beneficial or neutral mutation, making the property of evolvability itself evolvable. Furthermore, what constitutes a beneficial phenotype may also change over time, because it greatly depends on the current phenotype and the ever-changing environment. Therefore, evolution of evolvability has been studied in vitro and in silico as a reaction to environmental changes.

Yeast, for instance, has the capacity to respond very quickly to environmental changes by efficiently generating offspring with the right mutations to deal with the change. Only a few mutations are necessary for large phenotypic changes, suggesting that yeast have evolved the capacity to gain these mutations easily [46, 47]. This idea is supported by the modeling efforts of Crombach and Hogeweg, who used a computational model to show that over evolutionary time, an in silico genome may become organized such that beneficial mutations become more likely. Also, the gene regulatory network arising from the genome structure tends to become more poised to the environmental changes over evolutionary time, creating a key sensory gene whose presence or absence serves as a switch for network states suited for one or the other environment. Therefore, only one gene deletion is required to make the switch. This shows that the genome and the genotype-phenotype map can become more structured to facilitate evolvability [48, 49].

The metaphor of a dynamically changing landscape of possible phenotypes with a certain fitness can be extended to software systems. In the case of a software system, such as the Charging System (COS), a set of given market requirements is satisfied by introducing new features (innovation). These new features must alter the system in a way that the resulting system version optimally satisfies market requirements. In order to find such features, the system must walk through the “feature space” and select the feature(s) that increases the likelihood of success in the market. However, market requirements (as well as other influencing factors like technology or competition) change over time, and alter the geometry of feature space, much like the changing environment in the biological example.

The change in feature space is influenced by external factors such as changing market requirements, technology upgrades, and the need to keep up with competitors. Therefore, the optimization procedure is akin to a continuous process with a feedback loop. This can be illustrated as follows: Pressure from the market leads to the selection of a certain feature. Let us assume that this leads to a better product fitness. Over time, new market requirements emerge, which result in a lower product fitness based on the current feature set. This updated fitness information alters the feature space, and a new set of features is required to improve the product fitness. This information is fed back into the product development (innovation) process which results in solving the optimization problem over the updated feature space. This loop continues over the entire product lifetime.

The adaptive nature of the above mentioned, feedback-loop-based optimization problem makes it very similar to the concept of learning. Let us illustrate the concept by describing a supervised classification learning algorithm. The objective of the algorithm is to learn a function \( f \) that can correctly classify an input instance. Figure 1 illustrates the concept.

![Learning algorithm with feedback loop.](image)

Please note that introduction of a feature can also alter the market situation. Therefore, it is not only that the market affects the feature space, a feature can affect the market as well.
that maximizes the accuracy of classification over the training set $x$. In other words, the resulting classifier must make as few mistakes as possible while classifying instances of $x$. A good classification performance over the test set is important, but it is not sufficient. The purpose of a classifier is to be able to classify instances that it has not seen before. In order to do that, the function $f$ must be general enough. A function that learns patterns rather than concrete instances is more general, and is therefore more likely to correctly classify unseen instances. Thus, in addition to classification accuracy, generalizability is a vital characteristic of a successful classifier.

We conjecture here that generalizability in a classifier is analogous to the concept of robustness in evolution – the more generalizable a classifier, the more robust it is against changes in input instances. This leads to the speculation that learning algorithms might serve as suitable toy models to study optimization algorithms that adapt to changing energy landscapes, i.e., processes such as evolution. Moreover, the concept of generalizability might provide a mathematical basis for understanding robustness in evolution.

**Constructal law.** According to the constructal law [50], in order to survive, every system – natural or engineered – must maximize the flow of currents through it. A flow system is characterized by 1) properties, such as volume, 2) performance, and 3) structure. The fundamental relation between the properties of the system, its performance, and its flow structure is similar to the formulation of an optimization problem, where the properties form the constraints and the performance defines the objective function. Since it is the flow that needs to be maximized, structural elements such as configuration, layout, geometry, and architecture, also serve as parameters. The constructal law has been put forward as a fundamental law of thermodynamics. In terms of application to evolution, the constructal law has been used to explain survival by increasing efficiency, survival by growing and occupying maximum territory, and survival by maximizing the use of available space.

In the light of the above description, our initial hypothesis is that optimization is the mathematical problem underlying evolution in both software and biological systems might appear similar to the Constructal law. However, there are important differences. Our formulation of innovation as optimization over a dynamic landscape incorporates the feedback loop between the system and its environment. Moreover, we highlight the fact that innovation can itself affect the environment. To the best of our knowledge, these aspects of the innovation process are not explained by the Constructal law. We believe more over that our formulation can be used as a toy model to study the dynamics of the processes of innovation and evolution. A computer model can be applied to any system, natural or artificial, in order to generate the characteristic features of the evolutionary process.

**Punctuated Equilibrium.** Our interpretation of punctuated equilibrium in OCS product evolution is as follows: Even though new features are consistently added to the OCS product, most of these features stay customer-specific for a long time. Also, very few features cause major changes to the system. It is only over a period of three to four years that the system is upgraded to a new major version. This transition is driven not only by customizations, but also by wider market demands, possibilities to exploit new market niches, competition with other products in the same category, and technological evolution. It is only these major version upgrades that significantly alter the product outlook.

In addition, punctuated equilibrium can be measured empirically using the mathematical formulation of the process in terms of power law statistics. This was shown in [51] by collecting commit statistics of three open source software systems. The commit statistics comprised the number of additions and deletions of lines of code between commits. It was shown that the probability distributions of number of lines added and number of lines removed can be accurately approximated by power functions. This power function approximation is interpreted as the manifestation of punctuated equilibrium.

We argue however, that a model of software evolution based solely on addition and deletion of lines of code is overly simplistic, and not representative of the complex process of innovation in commercial software products such as the OCS. Moreover, research shown in the paper does not provide insight into the process of innovation. It is therefore our belief that in order to conclusively prove the existence of punctuated equilibrium in software evolution, and to understand why it is a fundamental characteristic of the process, the dynamics of the process must be understood in greater detail.

**Discussion and Future Work**

In this paper, we have explored the characteristics of evolution that are shared between software systems and biological systems using the novel ICE framework. We have shown that it is indeed the case that several aspects of the evolutionary process are similar in these systems. Moreover, we have presented certain similarities in the design principles employed by both systems.

By viewing system evolution as an optimization process, we see that the space of all solutions available in biology is constrained. A major constraint is imposed by the laws of physics, which determine the degrees of freedom that can be exploited during interactions at the molecular scale. It is this constrained solution space that eventually results in the emergence of complex behavior at the cellular, system, organ, organism, and ecological levels.

The behavior that emerges from constrained molecular interactions has given humans the ability to engineer systems, which help solve problems in everyday life. The most remarkable fact, however, is that these artificially engineered solutions are (in many cases) very similar in principle to the solutions that already exist in biological systems (often at the cellular level).

This raises the following questions:

1. Are our creative processes ultimately bound by the physical constraints that underlie molecular mechanisms? Or, do we tend to interpret the phenomena at the molecular level according to our own understanding, which is limited by our senses and neurological processes?
2. Is life, as we see it today, a mere product of the optimization problem that is evolution and innovation? In other words, does the optimization problem serve as a *generator* for all mechanisms of life?

It is perhaps the case that the constraints that apply on the molecular level are projected onto the behavior that emerges as a result. This would explain why we can see glimpses of the underlying molecular mechanisms in the emergent behavior.

Earlier in this paper, we presented a model that describes the processes of innovation and evolution as optimization
problems. Whether this model completely describes these processes is an open question. We have yet to discover the necessary and sufficient conditions for the process of evolution.

**Future work.** Our treatment of the subject matter so far has been primarily descriptive and to some extent speculative. An important next step is to move toward a more mathematically rigorous and data intensive research methodology. We are currently in negotiations with potential industrial collaborators for access to data that will help us empirically evaluate the process of commercial software evolution. One of the high priority tasks would be to verify the existence of punctuated equilibrium using a model that is reflective of the complex dynamics of software evolution. More generally, we intend to use our collaborators’ data to approach the problem from the point of view of developing mathematical models that fit the data.

Furthermore, we intend to develop a more rigorous model of optimization over a dynamic landscape. A possible avenue would be to use *Artificial Neural Networks* to study the dynamics of such a process, and use small toy problems to develop a deeper understanding of the characteristics of the evolution.

Our long term objective is not only to find similarities and differences, but understand why these exist. We are particularly interested in the design principles that govern the successful evolution of both software systems and biological systems. It is possible that design patterns or motifs can be discovered, that abstract away the system details and yet specify the circumstance in which a certain design can be useful.

**Appendix: Similarities between Design Principles of Software and Biological Systems**

The similarities described below were discovered while studying [52] and [53].

**Decapsulation, Facades, and GPCRs.**

**Encapsulation, Decoupling, and Facades**

The central concept in object-oriented programming is that of a Class. A class is a blueprint for runtime objects. Each class is generally partitioned into public and private sections. One object can call the public methods of another object, but access to private methods is restricted. We refer to the set of public methods of a class as its public interface.

The public interface is a list of method signatures. The general structure of a method signature is:

```
<return type> <name> (<parameter types>)
```

For example, following is the signature of a method written in the Java programming language:

```
Double sqrt ( Double input );
```

The only information available is the list of data types of input values, as well as the data type for the return value. No information is available about how the method performs the computation (in this case the algorithm used to compute the square root). Moreover, the public interface can be split into multiple facades. Each facade may consist of only a subset of method signatures from the public interface. This way, different objects see different facades and may never be aware of the whole picture. If one facade changes, it may not have an impact on the others.

This can be seen as decoupling of *form* from *function*. Decoupling is a very important feature, because it allows a programmer to change the internal details of a method as much as needed; as long as the signature is not changed, none of the calling methods will require any changes. Similarly, altering one facade does not impact the others. Large-scale software systems consist of hundreds of inter-dependent interactions between classes. Updating such a system would not be feasible if changes could not be localized.

**G Protein-Coupled Receptors**

G Protein-Coupled Receptors (GPCRs) are transmembrane receptors. There are two important parts of this molecule: the extracellular ligand-binding site, and the cytoplasmic region. In response to ligand binding, GPCRs can trigger an intracellular signaling cascade. As such, they provide a mechanism for transmission of information from the extracellular space (outside the cell) to the cytoplasm (inside the cell).

The extracellular ligand binding site is only receptive to molecules with a certain set of characteristics including shape and size. This is very similar to a method signature: just like a method signature can accept only certain data types, the ligand binding site reacts only to ligands with particular properties. The set of all transmembrane receptors for a particular cell can be thought of as its public interface. Moreover, the intracellular signaling cascade triggered by the cytoplasmic end of the GPCR can be different in different cells or under different conditions. Consequently, different cascades can be triggered by the same ligand, just like updated methods can be triggered by the same signature.

There are two important architectural concepts at work here. The intracellular mechanisms are encapsulated by the public interface comprising receptors and channels. Moreover, the ligand binding and recognition mechanism is decoupled from the intracellular signaling cascade. These features make the receptors much more generic, and therefore applicable to a variety of different cell types and functions.

**The Link**

Encapsulation and decoupling are not only the cornerstones of object-oriented software design, but also appear to be fundamental to cellular function as well. Object-oriented programming was created in the 1970s. It was not inspired by the molecular mechanisms described above (although certain other aspects were biologically inspired[10]). In fact, not much was known about the molecular mechanisms at the time. Yet we see that the architectural concepts created by the software engineering community had already evolved in nature and been deployed at the molecular level.

**Memory Management, Apoptosis, and Necrosis.**

**Objects and Destructors**

Let us consider dynamic memory allocation and deallocation in the C++ programming language. The memory required by an object must be allocated prior to its creation. This is done using the *new* operator. When the object is no longer needed, it is removed from memory using the *delete* operator. On the surface, this sounds simple enough. For large-scale software

9 The structure of a method signature may vary between different programming languages.

systems however, creation and destruction of objects is more complicated. When we delete an object, it means deallocating memory for that specific object. If the object itself has allocated memory for other objects, that additional memory has to be explicitly deallocated from within the object. This is one use for destructors, a destructor is a method where the programmer can specify which resources to release when the object is destructed. So if an object is designed correctly, and it is properly destructed when required, all resources should be released when they are no longer in use. Let us call this *programmed object destruction*.

In certain error scenarios, programmed object destruction is short-circuited and some of the steps necessary for resource release are not executed. Such error conditions often lead to memory leaks, dangling pointers, and other faults that may result in program failure.

**Apoptosis and Necrosis**

**Programmed Cell Death (Apoptosis)** is very similar to programmed object destruction. In certain cases, as in the case of cell injury, special intracellular signaling pathways are triggered to ensure that before the cell dies, all potentially toxic substances are stored in vesicles. These vesicles can then be taken up by neighboring cells and metabolized [53].

When apoptosis pathways fail, it leads to necrosis. Not only does the necrotic cell die, but toxic substances are released into the extracellular space which are harmful to the surrounding cells.

**The Link**

Life is a constant struggle against disorder; maintaining ordered processes is vital. Similarly, a software system has to constantly shuffle resources like memory in order to ensure smooth function without failure. It appears that the concept of disorder applies to both software and biological systems, and that the mechanisms developed by software engineers to deal with disorder are similar to those used in cells.

**Garbage Collection and Proteolysis.**

**Garbage Collection**

Java is a high level programming language. It relieves the programmer of explicit memory allocation and deallocation. Whenever an object is created in Java, memory is automatically allocated by the language runtime. Similarly, destruction of an object need not be triggered explicitly in the program. A special program in the language runtime (called the garbage collector) looks for objects that are no longer needed and removes them from the memory. This process is known as Garbage Collection.

Garbage collection ensures that unwanted objects do not occupy memory resources. This is important over time, because aggregation of such objects results in out of memory runtime errors.

**Proteolysis**

Just like unwanted objects are collected and deallocated in garbage collection, when a protein is no longer needed for cellular function, it is transported to the proteasome, and broken down into smaller parts (polypeptides and amino acids). These constituents can then be used to manufacture other proteins. This is important because aggregation of unwanted proteins can cause fatal illnesses such as the Huntingtons disease [53].

**The Link**

Unwanted objects in software are like unwanted proteins in a cell. The two need to be removed for the proper functioning of software and cells. There appears to be a remarkable similarity between how the problem is handled in both systems.

**ACKNOWLEDGMENTS.** We would like to thank the lecturers, staff, and participants of the Santa Fe Institute’s Complex Systems Summer School 2014. A special thanks to Andreas Wagner, Miguel Fuentes, Melanie Mitchell, and Zoran Nikoloski for encouraging us to move forward with our ideas. We would also like to thank Yu Liu and José Aguilar-Rodriguez for sharing with us their perspectives on the subject.

---

8. McCulloch W, Pitts W (1943) A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics 5:115–133.