Chapter 10
Studying Evolving Software Ecosystems based on Ecological Models

Tom Mens, Maëlick Claes, Philippe Grosjean and Alexander Serebrenik

Summary. Research on software evolution is very active, but evolutionary principles, models and theories that properly explain why and how software systems evolve over time are still lacking. Similarly, more empirical research is needed to understand how different software projects co-exist and co-evolve, and how contributors collaborate within their encompassing software ecosystem.

In this chapter, we explore the differences and analogies between natural ecosystems and biological evolution on the one hand, and software ecosystems and software evolution on the other hand. The aim is to learn from research in ecology to advance the understanding of evolving software ecosystems. Ultimately, we wish to use such knowledge to derive diagnostic tools aiming to predict survival of software projects within their ecosystem, to analyse and optimise the fitness of software projects in their environment, and to help software project communities in managing their projects better.

This work has been partially supported by F.R.S-F.N.R.S. research grant BSS-2012/V 6/5/015 and ARC research project AUWB-12/17-UMONS-3, "Ecological Studies of Open Source Software Ecosystems" financed by the Ministère de la Communauté française - Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique, Belgium.

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_10, © Springer-Verlag Berlin Heidelberg 2014

10.1 Introduction

Mathematics and computer science have been very helpful to advance research in biology, even so much that it has spawned a research field of its own: bioinformatics [512]. In the other direction, inspiration from biology has lead to numerous new achievements and improvements in computer science, such as neural networks [377], genetic algorithms [338, 611], optimization and artificial intelligence algorithms inspired by ant colonies and swarms of bees [126]. As explained in Chapter 4, some of these techniques have found their use in the context of search-based software engineering.

More specifically, *ecology* has been a fruitful source of inspiration for software engineering research. Huberman and Hogg considered a distributed computing system of concurrent agents as a *computational ecosystem*, analogous to biological ecosystems [411]. They studied the dynamics and chaotic behavior of such computational systems and showed how reward mechanisms may stabilize the system, thereby optimizing its performance. Calzolari et al. adapted the ecological *predator-prey* model to empirically study and predict the relation between software defects (prey) and programmers (predators) [155]. Lawrance et al. leveraged predator-prey relationships to apply information foraging theory to software maintenance, by considering developers as predators and the information they seek as prey [500]. Posnett et al. studied the risk of using aggregation techniques in empirical software engineering through its relation to the notions of ecological inference and ecological fallacy from sociology and epidemiology [699]. More recently, they also compared the developer-artifact contribution network to a predator-prey relationship, leading to a conceptually unified view of measuring focus and ownership [698].

In this chapter, we explore similar analogies with *software ecosystems* and *software evolution*. Several researchers have advocated biological evolution and ecological principles as a source of inspiration for software evolution [79, 638, 811, 940] but, until now, this has remained mostly at the level of the intention. Although research on software ecosystems is emerging, the application of ideas transposed from ecosystems in nature seems to be underexploited. The transfer of knowledge has essentially limited itself to a reuse of terms.

Despite the fact that natural ecosystems have been studied for many decades, and that many evolutionary theories and ecological models have been proposed and experimentally validated, little research exists that tries to adopt or adapt such theories to the domain of evolving software ecosystems. Although it is true that biological species, systems and ecosystems are quite different from what we can find in software, we share the belief of [240] that ecological models and biological evolutionary theories can be adapted to study how software ecosystems and their constituent projects evolve. Even if biological and software ecosystems do not evolve at the same pace, life on earth got a much longer history and thus, had more opportunities to explore and find optimized pathways through natural selection.

The evolutionary processes that can be observed in nature may therefore be very inspiring for software engineers and researchers. It allows them to gain an increased understanding in how software projects compete or collaborate in their surrounding

environment, and how this differs from biological environments. This insight will hopefully lead to guidelines and tool support to help the software project communities in predicting and improving survival of their projects. This will allow them to stay ahead of the competition, produce higher quality products and increase their fitness, resilience and stability over time in a rapidly changing environment.

The remainder of this chapter is structured as follows. Section 10.2 starts by exploring and comparing the notions of ecosystem and ecological principles that exist in biology and software engineering. Section 10.3 compares the notions of biological evolution and software evolution. Section 10.4 presents our emerging research to study the evolution of open source software ecosystem based on insights from the dynamics of natural ecosystems. Finally, Section 10.5 concludes.

10.2 Ecosystem terminology

The term *ecosystem* exists both in ecology and software. We present the characteristics and examples for both types of ecosystems in Section 10.2.1 and Section 10.2.2, respectively. In Section 10.2.3 we go beyond a simple reuse of terminology by drawing analogies between both types of ecosystem, despite the fact that the domain and discipline in which they are used and studied is completely different. In particular, we explain how ecological principles can be adapted and applied in the context of software ecosystems.

10.2.1 Natural ecosystems and ecology

According to [481], *ecology* is *the scientific study of the interactions that determine the distribution and abundance of organisms.* Typically, the dynamics of these interactions are studied in the context of an ecosystem. The term *ecosystem* was originally coined in 1930 by Roy Clapham, to denote the physical and biological components of an environment considered in relation to each other as a unit [927]. In other words, an ecosystem combines all living organisms (plants, animals, microorganisms) and physical components (light, water, soil, rocks, minerals) that interact with one another.

More generally, the ecosystem *dynamics* are traditionally represented in a *trophic web* (more commonly known as the so-called *food web* or *food chain*). This trophic web forms an interaction network that relates predator to prey or organism to resource [47, 682, 926]. Such a network usefully captures the relationship between *consumers* and the ecosystem's *resources* (such as food, nutrients and space), and the effect of this relationship on the population of different species in an ecosystem. A trophic web is organized in trophic levels corresponding to families of functionally consistent species. Consumer-resource relationships typically take place between different levels of the trophic web.

An ecosystem is the result of a delicate and dynamic balance between its interacting components. Trophic webs can be constrained from the bottom up, limited by the resources available to primary producers, or from the top down, driven by predation by top consumers. Ecosystems with "wasp-waist" control combine both mechanisms with partial effects in both directions acting simultaneously. Several marine ecosystems exhibit such a wasp-waist structure, where a single species, or at most several species, entirely dominate the population [64, 206, 412]. A typical example of the top-down control dynamics in an ecosystem is the so-called *predator-prey model*, representing a biological interaction in which some organisms (the *predators*) hunt for, and feed on, other organisms (their *prey*). The dynamics of such interaction can be described using linear or nonlinear models consisting of parametric differential equations [720].

Since an ecosystem's resources are finite, they need to be recycled or reused whenever possible. To achieve this, *energy* needs to be put into ecosystems constantly, typically in the form of light to drive the necessary biochemical processes that enable recycling of resources. An ecosystem has a *static equilibrium* if there are no exchanges between the components constituting it. Natural ecosystems typically have a *dynamic equilibrium* since there are always major exchanges between its components. For example, there may be important exchanges between the various levels in the trophic web, and an equilibrium is reached by fluxes in opposite directions whose total sum is zero.

The capacity of a biological ecosystem to maintain an equilibrium over longer periods of time is called its *stability*. Systems that can attain the most stable equilibrium survive the longest [826]. Often, this stability is put into peril by human interference, e. g. through the use of some of the resources required by the ecosystem. Examples of such disturbances for the ecosystem of coral reefs are, for example, climate change, water pollution and overfishing. *Sustainability* refers to the ability to maintain the ecosystem despite of humans deriving their needs from its natural resources.

The *resistance* of an ecosystem characterizes its ability to withstand environmental changes without (too much) disturbances of its biological communities. If the disturbances become too important, ecosystems may get out of balance (e. g. a meteorite impact that made all dinosaurs extinct). The ability of an ecosystem to reorganize itself and return to an equilibrium close to the initial one is called its *resilience* [405]. Because of the disturbance, the new equilibrium that is reached may be different from the original one (some types of organisms may have disappeared, and others may have taken up their place), so the ecosystem will have evolved.

Ecologists emphasize the importance of *biodiversity* [570, 576, 671], and generally acknowledge that the stability and resilience of an ecosystem is favored by a higher diversity. If the ecosystem has a large species diversity of producers and consumers that respond in different ways to disturbances, it is more likely that the ecosystem will be able to heal itself after a disturbance, since some species can compensate for others that disappear. Relating diversity to the aforementioned predator-prey relationship, Williams and Martinez considered two symmetric perspectives, from a prey's perspective and from a predator's perspective [570]. Other types of

diversity have been studied by ecologists such as genetic diversity, functional diversity, spatio-temporal diversity, etc.

An *ecological niche* of a species determines the environmental conditions necessary for the species to maintain its population in response to the distribution of physical conditions, resources and predators in the ecosystem. Among others, it characterizes the subregions of the ecosystem's habitat that are usable or accessible to the species (e.g., land animals will not live under water).

Example 10.1 (coral reefs). Coral reefs are among the most biologically diverse ecosystems on earth [925]. Competition for resources such as food, space and sunlight are the primary factors determining the biodiversity and population of organisms on a reef. The single most important species of the ecosystem are the scleractinian coral polyps. They secrete hard skeletons that form the coral reef structure required for the other species to thrive: sea anemones (soft coral polyps), sponges, crustaceans, mollusks, sea urchins, fish, sea turtles, algae, sea grasses, and many more. These species have established a dynamic equilibrium with a delicate balance between predators and prey. Fluctuations in the population of one species can drastically alter the population of other species. External forces that may disturb the equilibrium of the coral reef ecosystem are for example hurricanes, but other human-inflicted changes may play an even more important role. Overfishing, for example, may lead to an increased growth of algae and sea grasses, resulting in an increase of the population of sea urchins that may destroy the corals.

10.2.2 Software ecosystems

Software systems are among the most complex artefacts ever created by humans. Collaborative software development has become increasingly popular over the last two decades. It represents a successful model of software development where communities of developers collaborate on a voluntary basis, while users and developers of the software can submit bug reports and requests for changes.

To reflect this increase in complexity and scale, the term *software ecosystem* has been coined by Messerschmitt and Szyperski [603] to refer to such systems. It has now become a very active area of research, as can be seen in a recent systematic literature review [556]. Unfortunately, in contrast to natural ecosystems, there is no common definition of software ecosystem. It can be defined and interpreted in different ways, depending on the point of view.

10.2.2.1 Business-centric viewpoint

One of the first occurrences of the term software ecosystem can be found in [131] where it is used to refer to the way in which software suppliers, vendors, competitors, users, and third-party developers interact in software product lines. This view emphasizes the *business* perspective of a software system. A similar view, including

the socio-economic environment and regulatory framework is adopted by Jansen et al. [433, 434], who define a software ecosystem as a *"a set of actors functioning as a unit and interacting with a shared market for software and services, together with the relationships among them."* This view is schematically presented in Figure 10.1. An entire book is devoted to this perspective of software ecosystems [435]. A typical, but not exclusive, characteristic of these types of software ecosystems is the *competitive* aspect. The different projects in the ecosystem are in competition, either because they target the same end-users or offer the same type of service.

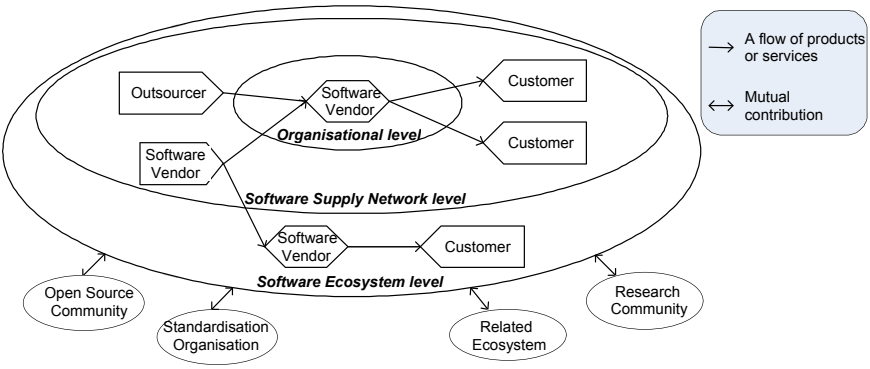

Fig. 10.1: Actors in a software ecosystem. Figure reproduced from [893] with permission from Edward Elgar publishers

Since, as illustrated above, business-centric software ecosystems often constitute a core strategic asset for its contributors and supporting companies, it is crucial to gain more insight in how ecosystems evolve and can be maintained successfully over time.

10.2.2.2 Development-centric viewpoint

An alternative, more fine-grained definition of software ecosystem is provided in the seminal work of Messerschmitt [603] to refer to *"a collection of software products that have some given degree of symbiotic relationships."* A similar definition is given by Lungu [546, 547], who defines a software ecosystem as *"a collection of software projects which are developed and evolve together in the same environment."* This environment refers to the development environment, *i.e.* the software and hardware tools used during the development process.

We extend these definitions to take into account the *collaborative* and *social* aspects as well, by explicitly considering the communities involved (e. g. user and developer communities) as being part of the software ecosystem. Like software

projects, the communities involved evolve over time (users and developers come and go). In addition, there is a high degree of interaction, even some kind of symbiosis, between the software projects and the communities of the ecosystems. This viewpoint is adopted by [321, 333, 335, 594, 684, 723, 886] that focus both on the technical aspects of the software produced and the social aspects of the communities producing and using this software.

It is especially in ecosystems where the community works towards a common goal that the collaborative nature wins over the competitive nature. Typically, software ecosystems consist of a relatively closed core software system that provides the basic functionality and that is developed by a more or less stable core team of developers, surrounded by a large collection of contributions provided by peripheral developers or even end-users [631, 689, 723].

We can provide numerous examples of software ecosystems, and many of them can be interpreted from both the business-centric and the development-centric viewpoint.

Mobile app stores, commercial or free application repositories for mobile operating systems (such as *iOS*, *Android* and *Windows 8*), form a business-centric ecosystem. While these operating systems are provided by Apple, Google and Microsoft, respectively, the SDKs and APIs allow third-party developers to build mobile applications on top of these operating systems. The mobile app ecosystems consist of the users, developers, managers of the mobile OS and the third-party mobile applications built on top of them. The official mobile app stores allow for applications to be sold to end-users, with a shared profit. For Android, there is also a free and open source software repository of applications, called *F-Droid*.

The empirical study of the evolution of mobile applications is an emerging area of research. For example, Battacharya et al. [78] carried out an empirical study on the evolution of bug-related issues in 24 widely-used open source Android apps, while Basole et al. [77] studied the emergence and growth of mobile app stores in the mobile service ecosystem. McDonnell et al. [578] studied the rapid evolution of APIs and their adoption by client apps in the Android ecosystem.

IDEs for programming languages such as *Java* (e. g. *Eclipse* and *NetBeans*) or *Smalltalk* (e. g. *Squeak* and *Pharo* [721]) can be seen from a business-centric viewpoint. For example, the non-profit Eclipse Foundation is involved in the strategic direction, marketing and promotion of Eclipse and contains representatives of different companies such as IBM (the founder of Eclipse), Google, OBEO, Oracle, SAP, Talend. Eclipse is supported by numerous software vendors, and each of these vendors may provide different plugins with similar functionality, that are in direct competition with one another.

From a development-centric viewpoint, the Eclipse ecosystem is the universe of Eclipse *plugins* [191] together with the developers of these plugins. Studying the evolution of plugins is an active area of research [151–153, 915, 916]. All different Eclipse plugins rely on a common underlying architecture, platform and set of libraries without which they are unable to function correctly. The community of plugin developers therefore shares the common goal of improving a complete inte-

grated software development environment. NetBeans, the main open source competitor for Eclipse, has a similar modular architecture with a common core.

Linux distributions form an ecosystem comprising several hundreds of actively competing Linux distributions, that are all based on a common core (the kernel of the Linux operating system [428] and a set of GNU libraries and utilities). The distributions vary in the system they target (e. g. desktop computers, laptops, tablets, smartphones, embedded systems) and the applications that are bundled with the distribution. Some distributions are commercially driven (e. g. Fedora Red Hat, SUSE, Ubuntu, and Mandriva), while others are entirely community-driven (e. g. Debian and Gentoo). An excerpt of the evolution of Linux distributions is shown in Figure 10.2. While the family of all Linux distributions is an ecosystem, each of the distributions that belong to this family can also be considered as an ecosystem of their own, composed of the packages (together with the necessary building and configuration files) contained in the distribution. Gonzalez et al. [341] have taken a closer look at the evolution of the Red Hat and Debian distributions.

Forges. Open Source Software (OSS) repositories, commonly known as *forges*, can be considered as business-centric, since there is no control on the governance of the projects hosted in the forge. Examples of such forges are SourceForge, GitHub, Bitbucket, Launchpad and Savannah. There are also many forges that are dedicated to particular programming languages, such as the CCAN archive network for the C programming language, the CPAN archive network for PERL, RubyGems for the Ruby language, the Python Packaging Index for Python programs, and so on. Because of the lack of control, within and across these forges there are often different projects with similar functionality between which the users can freely choose.
Capiluppi and Beecher [161] performed an interesting empirical study in which they studied the type of software forge (they refer to them as FLOSS repositories) and their mode of governance on the projects they host. They compared SourceForge (which they consider to be an *open* repository) with Debian (which they consider to be a *controlled* repository). They concluded that Debian hosted larger, more active and more complex structures. As a side-effect, more effort is needed to maintain these projects. Chapter 6 of this book explains how socio-technical information recorded in OSS forges (but also in microblogs and software forums) can be leveraged for different types of development and evolution activities, using a variety of information discovery and retrieval techniques.

Social networks, such as *Facebook*, *LinkedIn*, *MySpace* and *Google+* can also be regarded as business-centric software ecosystems. They allow application developers to develop and integrate third-party applications, through a well-defined API. This provides significant added value to both the social network and the application providers.

GNU (which is a recursive acronym for GNU's not UNIX) aims to provide a full free operating system based on the GNU General Public License (GPL) and the principles of UNIX. It is composed of GNU projects which are often ecosystems themselves. Examples of such sub-ecosystems are *R* and GNOME. Unlike most other software ecosystems, R is targeted towards end-user programming [321] since, the

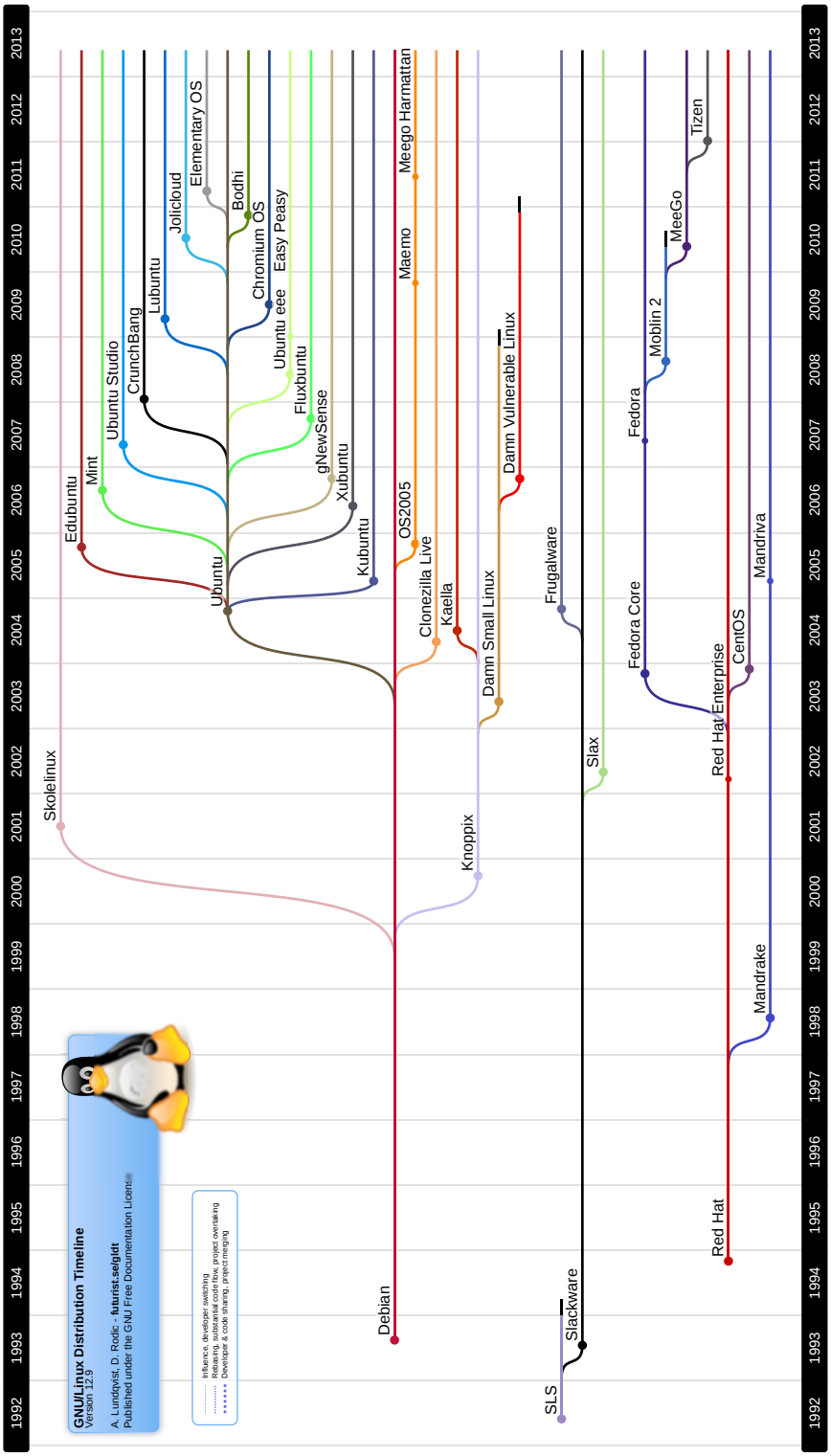

Fig. 10.2: Linux distribution timeline (simplified version based on http://futurist.se/gldt/)

majority of its contributors are statisticians and scientists rather than professional software engineers.

Archive networks. The GNU *R* community shares the goal of creating a statistical computing environment. It achieves this through the Comprenhensive R Archive Network (CRAN), a developer-centric ecosystem in which each community member can contribute packages implementing specific statistical analysis functions and visualization tools. Similarly, the TEX community has its CTAN archive network containing all kinds of material around TEX. There exists similar archives for other languages such as CEAN for Erlang, RubyGems for Ruby and PyPI for Python.

Graphical dekstop environments. GNOME and KDE are two full desktop environments for Linux and BSD operating systems. Both are based on a specific graphic toolkit (respectively GTK+ and Qt4). The developer communities share the common goal of delivering a complete user-friendly desktop environment. GNOME has been the topic of study for many researchers [320, 524, 640, 886].

Apache is an ecosystem with a community of developers based around the Apache Software Foundation and the Apache License. One of its most famous projects is the Apache HTTP server. Apache is a decentralized community that uses a consensus-like development process. The aim is to provide stable, open and quality software developed by technical experts. Mockus et al. compared the Apache development process with the one of Mozilla [613]. Bavota et al. [81] studied the evolution of the dependencies between projects constituting the Apache ecosystem. Weiss *et al.* [911] studied the emails exchanged by the contributors of the Apache projects for discovering simple migration patterns between projects and from the outside to a project. Gala-Pérez *et al.* observed that the ratio of email messages in public mailing lists to versioning system commits has remained relatively constant along the history of Apache, and therefore advocate this ratio as a way to measure healthiness of an ecosystem's evolution [306].

10.2.2.3 Collaborative and socio-technical aspects of software ecosystems

From the two aforementioned definitions of software ecosystems we have seen that projects belonging to a software ecosystem can vary in a continuum ranging from highly *competitive* (if the business-centric viewpoint prevails) to highly *collaborative* (if the sense of community is very strong and there is strong incentive to work together towards a common goal). Many ecosystems fall somewhere in between, with some degree of collaboration and some degree of competition. It is clear that the competitiveness will have an important influence on the way the ecosystem will evolve over time.

Example 10.2 (The R ecosystem). Let us have a look at the collaboration and competition in the previously mentioned R ecosystem. It only minimally complies to the business-centric view because of its open nature: all packages in the CRAN archive network are required to comply to an open source license. Because of this there is much less competition in the sense of having many different packages with similar

functions. When packages do contain similar functions (this tends to be more common for "basic" functionalities), it is mainly because some contributor needed more advanced features for that function in its own package than what was available in existing packages. In many cases, that contributor will write his own function inside his own package instead of proposing to contribute changes to the existing one. Thus, there is little collaboration, but a more fragmented implementation of features across packages that are developed rather separately from each other. Formally verifying the above claims is outside the scope of the current chapter, as it requires an extensive empirical study of R packages.

Technical aspects are essential for software ecosystems. They need to rely on a sophisticated software and hardware infrastructure and tools needed for their proper functioning, distribution, development, maintenance and evolution. Typical support that is provided are SDKs, APIs, download repositories, package management, dependency management and installation tools, version control systems, tools for change tracking, bug tracking and defect management, mailing lists, websites and other communication fora.

Social aspects and communication between the members of the software development team are at least as important as the technical aspects for the success of any software project [90, 236, 265, 868]. This is especially true for OSS projects where it is, in most cases, easier to become involved in the development team. This implies that the team structure needs to be more flexible in order to accommodate the easy integration of newcomers and to deal with the frequent departure of developers. Chapter 6 of this book proposes a number of techniques to recommend "compatible" developers to a project.

Fitzgerald [295] coined the term OSS 2.0 to reflect the new generation of OSS ecosystems that significantly "evolved" over the last decade or so from its single-project antecedents. Empirical results and insights obtained for individual OSS projects do not necessarily apply to projects that are part of a bigger, highly collaborative ecosystem of interacting parts. Nakakoji et al. [631] distinguished between different types of OSS community members: developers, bug fixers, bug reporters, readers and passive users. They further subdivided developers into peripheral developers, active developers, core members and project leaders. They proposed a so-called *onion model* for the OSS community structure, suggesting that there are very few project leaders, a bit more core members, even more active developers, and so on, and that promotion and migration of contributions tends to follow the layers of this model. Jergensen contested this onion model in an OSS 2.0 setting [439], by showing that contributor migrations do not tend to follow this model in many cases. Many other empirical studies have studied the activity patterns of, and differences between, core developers and peripheral developers [162, 250, 689, 723, 828, 941]. A detailed discussion of these is, however, beyond the scope of this chapter. We refer the interested reader to [336].

Still related to developer communication, Abreu and Premraj [2] studied the correlation with software quality. They observed a statistically significant correlation between communication frequency and number of injected bugs in the software.

Through mining the source code repository and mailing lists of the well-known Apache and Mozilla OSS projects, Mockus et al. [613] investigated the roles and responsibilities of developers, and observed a set of implicit conventions among developers that implies an intensive communication. Madey et al. [305, 868] analysed the social networks involved in OSS development and observed power laws at many scales. Bird et al. [108] analysed social networks emerging from mailing lists discussions and observed a Pareto distribution. Mailers tend to form a small-world network at several points of view; for instance, few mailers received messages from an important number of persons while most of mailers received messages from few senders. A strong correlation between mailing and coding activities was found and evidence was provided that the role of developers in mailing lists is more important than the other mailers.

10.2.3 Comparing natural and software ecosystems

The premise of this chapter is to learn from ecology and natural ecosystems, that have evolved over millions of years, and use this knowledge to improve our understanding of software ecosystems. Existing research on natural ecosystems has already provided many useful insights on the underlying mechanisms and how we could better manage and preserve these ecosystems. Our hope is to learn from this research, and to apply some of its insights to obtain better strategies for managing, developing and maintaining software ecosystems, and to come up with processes that increase the fitness of projects and contributors belonging to the software ecosystem.

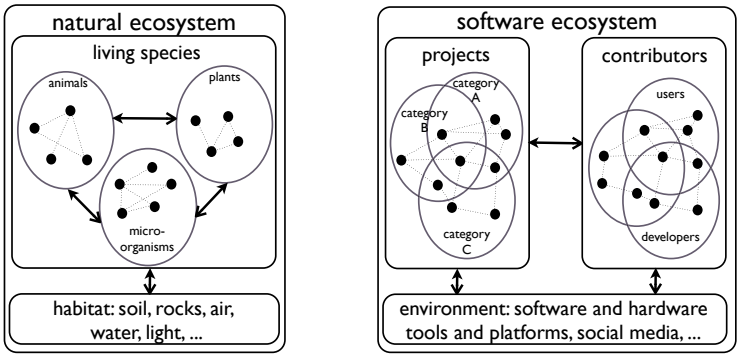

Fig. 10.3: Natural versus software ecosystems

When comparing biological evolution with software evolution, despite their obvious differences, we can also draw many analogies. This analogy is illustrated in Figure 10.3. If we take the development-centric viewpoint of a software ecosystem,

we can consider the software projects as being the equivalent of the "living species" of a natural ecosystem, and the physical habitat is replaced by the socio-technical environment in which these projects co-exist and evolve. The projects require software and hardware resources for developing, installing and executing the software products belonging to the ecosystem. All software projects interact with each other and with the user and developer communities and available resources. The software ecosystem can also be interpreted in an alternative way, by considering the contributors to the software projects as the equivalent of the "living species" and the software products then become part of the software and hardware environment of these species. This view may be particularly suited if we wish to study the social aspects of a software ecosystem. In practice, both of the above views are complementary and need to be combined in order to fully understand how software ecosystems evolve.

Example 10.3 (Coral reefs). In a coral reef ecosystem, the scleractinian coral polyps are responsible for creating the coral reef structure required for the other species to thrive. We find a similar idea in most business-centric software ecosystems, where there is typically a core set of projects (or core architecture), developed by a core group of developers, based on which the other projects are created.

Like natural ecosystems, a desirable property of software ecosystems is to be *sustainable*, in that their user and developer communities can use, maintain and improve the ecosystem's projects over longer periods of time. Just like the habitat of a natural ecosystem, the environment of a software ecosystem may undergo important changes, whether they be planned or unexpected. The *resilience* of a software ecosystem then refers to its ability to return to a stable *equilibrium* after minor or major disturbances. Examples of such disturbances are the appearance of a new competitor products, a loss of interest by the user or developer community, a change of technology (e. g. switch from the use of a centralized version repository to a distributed version repository), the introduction of new communication channels (mailing lists, StackOverflow—cf. Chapter 5 and Chapter 6, respectively) and other ways of collaboration.

Biological species evolve through mutation and crossover of genes between individuals of the same or different species. An analogy of such gene transfer in software projects could be the reuse of code from one project to another, or the migration of software developers from one software project to another.

Natural ecosystems require *energy* (e. g. air, water and sunlight) to thrive. The same is true for software ecosystems, but the type of energy required is quite different. If we consider the software projects as the species of a software ecosystem, the energy required to maintain and evolve them is the time and effort invested by the users and developers contributing to the software ecosystem, through commits in the version repository, bug and change requests, mails in the mailing lists, communication in forums and websites, and so on.

The notion of *biodiversity* also exists in software ecosystems, at different levels (as illustrated, in part, in Figure 10.3). First of all, there is a diversity of contributors involved in software development. The role of contributors may range from more

passive (e. g. users) to more active (e. g. developers, translators, UI specialists, etc.). Zooming in on the developers, we can distinguish between core developers, active developers and peripheral developers at a more fine-grained level [631, 723, 828]. For the software projects that are part of the ecosystem we observe a similar diversity. Some projects will be more user-oriented (i. e. they can be installed and used by end-users) while others will offer the core functionality that is needed by others in order to function properly. Sometimes there may be different projects with a similar functionality. This may be beneficial for the biodiversity since the disappearance of such projects will not be detrimental to the ecosystem since the other project could take its place. Another example of diversity is *conditional compilation*, which allows for a software product to create different variants adapted to specific platforms or user needs. *Software product lines* encourage controlled diversity across different software products with some shared common features (see Chapter 9).

It is likely that the mechanisms controlling the ecosystem *dynamics* (top-down, bottom-up or wasp-waist) can be adapted to software ecosystems as well. If a software ecosystem is mainly driven by its core developers or by limited hardware resources it might follow a bottom-up control process. If it is mainly driven by change and bug requests from the end users, it might rather have a top-down control. In many cases, the type of control is probably a mix between both, in the sense that some projects of the ecosystem (typically the core projects) will be driven or initiated by the developers, while others will be driven by the end-users' change requests and desire for new or modified functionality. A better understanding of the type of dynamics that control a software ecosystem may ultimately lead to better management strategies for maintaining the ecosystem over time.

The notion of *ecological niche* of a species also has a counterpart in software ecosystems: if we consider contributors (e. g. developers) to be the equivalent of a species, their ecological niche is determined by environmental factors such as the operating system they are using, their preferred programming language, the APIs they are using, their domain of interest, and so on. These characteristics will constrain the ecological niche of a developer to a subset of the total set of projects she could potentially contribute to.

10.3 Evolution

10.3.1 Biological evolution

A biological species corresponds to a group of organisms capable of interbreeding and producing fertile offspring. Biological evolution is characterized by the fact that a species is composed of many individuals whose genetic code differs. Those individuals can reproduce, leading to mutations and crossing in the genetic code. The evolutionary driving force is *variation* and *natural selection*. A central idea in the evolutionary theory of natural selection is the notion of *fitness*. It describes

the ability of a species to both survive and reproduce, and is equal to the average contribution to the gene pool of the next generation that is made by an average individual of the specified genotype or phenotype [673].

Different theories have been proposed by biologists to explain the evolution of biological species, and the field still evolves today. The *Darwinian evolution* model is generally considered as the major mechanism driving *biological speciation* (i.e. one species differentiating into two) in life on earth [218]. The field of *phylogenetics* studies, among others, the biological evolution history of a set of species [764]. In the Darwinian model, the evolution history can be represented by *phylogenetic trees* [294]. Such a tree describes the evolutionary relationships among species assuming that they share a common ancestor and that evolution takes place in a tree like manner.

There are other, less well-known evolutionary models, such as *reticulate evolution* [523, 779]. These models cannot be represented using a tree structure, but require some graph-like or network-like structure instead [414]. Reticulate evolution refers to the dependence between two evolutionary lineages. This is radically different from pure Darwinism where there cannot exist such transfer of information between two different species. When reticulation occurs, two or more evolutionary lineages are combined at some level of biological organization. Because life is organized hierarchically, reticulation can occur at different levels: chromosomes, genomes and species. At the species level, events such as *hybrid speciation* (by which two lineages recombine to create a new one) and *horizontal gene transfer* (by which genes are transferred across species) are the main causes of reticulate evolution. A group of animals where reticulate evolution is suspected to be of major importance is the scleractinian corals [891].

Apart from Darwinism and reticulation, other evolutionary theories have been proposed, such as *Lamarckism* [488]. Lamarck considered that the evolution is based on uses and needs rather than on natural selection. While this theory has been superseded by Darwinism in biology, this does not necessarily mean that we should exclude it as a possibly useful theory for modeling the evolution of software ecosystems. Indeed, software is developed with the aim to fill a need and its survival fitness is partially constrained by its likelihood to be used.

A fairly recent evolutionary theory is the so-called *hologenome theory of evolution*, originating from studies on coral reefs [732]. In this theory, the object of natural selection is not the individual organism, but the organism together with its associated microbial communities. This theory may perhaps be more closely related to what one observes in software ecosystems, where one should not consider the object of evolution (the software project) in isolation, but rather together with its associated community of contributors (e.g., users and developers).

The biological phenomenon of *co-evolution* [862] arises if the genetic composition of one species changes in response to a genetic change in another one. This can occur, for example, when two or more species interact and influence each other, or live in *symbiosis* (e.g., host-parasite, plant-pollinator).

The notion of ecological *refuge* is also very relevant in the context of ecosystem evolution [101]. The conditions in a refuge are such that the species are protected

from certain threats such as predation. A key characteristic of refuges is that they are a reservoir of diversity since they provide a means to sustain species that are not the fittest at some point in time. Refuges are important in an evolution context, since species in refuges may become dominant species in the future in response to environmental changes.

10.3.2 Comparing biological evolution with software evolution

To be able to apply the aforementioned and other biological evolutionary models to study the evolution of software ecosystems, these models will need to be adapted because there are notable differences between software projects and living species.

While biological species evolve due to changes and variations in the genetic code of its individuals, it is difficult to consider a software project as a collection of individuals. Of course, we could view the different instances of a software system that are deployed on particular machines as individuals of the biological species. The major difference is that there is strictly no variation in the code of the various software project instances installed (to draw the parallel with differences in the genetic code of living organisms), while even small genetic differences between biological individuals is a major driving force of biological evolution. It is worth noting, however, that the equivalent of phenotypic changes in living organisms is represented at a varying degree in software: configuration files, installable plugins or packages can modulate how a particular instance operates in a given context.

Another type of software where one can observe a sufficient level of variation necessary for being able to apply biological evolutionary theories are so-called *software product families*. These are addressed in Chapter 9 of this book. Each member of a product family is a variant that has similarities and differences with the other product family members, and the family as a whole evolves over time.

The main driver for evolution of biological species is the creation of offspring through biological reproduction. This is not true for the elements that constitute a software ecosystem: software projects cannot "reproduce" themselves to produce new generations (read: versions or releases) of offspring.[1] Note that one could also consider project forking or branching as some kind of reproduction. A similar argument as above holds for the members of the ecosystem's communities: new generations of developers and users are not produced through interbreeding of existing members, but rather through the intake of new members from outside the ecosystem.

The rate at which software projects evolve is several orders of magnitude higher than the evolution of biological species. Hence, one has to determine the relative temporal scale at which comparison is possible between biological mutations and changes in software projects.

[1] This argument does not necessarily hold for self-adaptive systems, which are capable of dynamically changing their runtime behavior. For more information on this specific type of software system we refer to Chapter 7.6 of this book.

We can only collect very partial records of the evolution of natural ecosystems, restricted to limited sampling in time and space. Models in ecology are thus always applied with a large degree of uncertainty. On the contrary, more exhaustive historical records exist for many open source software ecosystems, from their onset, thanks to version control systems[2] where every change is recorded and documented.

Scientific research on biology is primarily observational and passive. One can observe how natural ecosystems have evolved in a self-organised way over long period of times, and develop theories that explain this evolution. Given the long time scales involved it is hard to carry out "in vivo" experimental research to study how actual ecosystems and the species populating it evolve by modifying certain parameters in the ecosystem. For software ecosystems, it is really possible to carry out applied, in vivo research, since the software environment involves human beings (developers and users). This makes it possible, in principle, to interact with them in order to find out how and why a software project has evolved over time, and making it easier to alter the way in which the ecosystem will evolve in the future.

10.3.3 Transposing biological models to the software realm

Given these many differences, the question arises whether ideas from biological evolution can be easily adapted to gain a better understanding of software evolution. Nehaniv [638] discussed the differences between software systems and biological species from an evolutionary point of view. Svetinovic [811] suggested that a comparison between software evolution and biological evolution is a fertile field of study. Yu and Ramaswamy [940] suggested that software systems share similar evolvability properties with biological systems, implying that studying the evolution of these biological systems can help us understand and improve development of software systems. None of the aforementioned papers, however, have empirically studied this potential.

Some researchers have gone a step further in adapting biological models or mechanisms in the context of software evolution. For example, Hutchins [416] used genetic algorithms to understand evolutionary software development processes. Each branch of a software project is compared to an individual of a biological species and merging of branches is similar to the crossover operation (reproduction of two individuals). Software evolution is then described as a form of human-guided search for a program meeting requirements. Jaafar *et al.* [429] used phylogenetic trees to show the evolutionary history of object-oriented programs. They suggest to use such trees to facilitate the detection of code decay and fault-proneness.

Baudry [79] studied the relevance of the notion of *ecological refuge* in the context of evolving OSS projects. More in particular, they analyzed the potential of largely inactive projects as alternatives for biodiversity and evolution: some of these "unsuccessful" projects may survive and increase diversity by seeding future, successful

[2] While some data can still be incomplete in software repositories, it remains far more complete than for biological species where historical data like fossils are very sparse and incomplete.

projects. They empirically analyzed this by studying project forks for 48 projects in the GitHub forge, and found 3 occurrences of the refuge effect. Similar to national parks, that serve to protect endangered species, software forges may therefore serve to protect unsuccessful projects and reuse or revive them in the future.

Calzolari et al. [155] explored the use of the biological predator-prey model in the context of software evolution. This model has been used in biology to describe the dynamics of an ecological systems using linear or nonlinear models consisting of two parametric differential equations [720]. The basic idea is that software defects (requiring corrective actions) can be seen as the equivalent of biological prey, whereas the programmers act as predators (removing the defects by correcting them). Empirical evidence of the usefulness of this model was given by analyzing the evolution of two industrial software systems and accurately predicting their dynamics using the proposed model. Some adaptations of the original biological model were needed since, unlike species, software defects cannot reproduce themselves, implying the elimination of the reproduction term in the dynamic model.

Posnett et al. [698] explored a similar idea, by considering software modules as predators that feed upon the limited cognitive resources of developers (their prey). They combined this with the notion of biodiversity [570] to measure how focused the activities on a module are, as well as how focused the activities of a developer are. They found empirical evidence that more focused developers introduce fewer defects. Conversely, increased module activity focus leads to a larger number of defects.

To transpose other theories of evolution and speciation of living species to software ecosystems we might require a mix of different evolutionary mechanisms, with probably a domination of reticulate-like mechanisms over pure Darwinian differentiation. For example, we could transpose the notion of *fitness* to reflect the ability of projects to survive and maintain themselves within the ecosystem of which they are part. We could also transpose the notion of *biological speciation* to software ecosystems to represent the mechanism of *software project forking*.

Example 10.4 (Evolution of Linux distributions). One illustration of this phenomenon is the different GNU/Linux distributions that have forked from a few main distributions (Fig. 10.2). The distribution timeline of Linux distributions does not represent a tree structure but forms a directed acyclic graph with some connections between different branches of the tree, indicating the exchange or sharing of ideas, code, and developers (corresponding to horizontal gene transfer across species). This may ultimately lead to *project merging*, if the level of sharing becomes sufficiently high. Such project merging fits the phenomenon of *reticulation* that occurs when two or more evolutionary lineages are combined at some level of biological organization. The following examples illustrate these phenomena: (i) Maemo (Nokia's mobile OS based on Debian) and Moblin (an Intel Atom optimized GNU/Linux distribution) merged to form Meego; (ii) Crunchbang was first based on Ubuntu, but since 2010 it has been based on Debian rather than Ubuntu.

The biological phenomenon of *co-evolution* can also be useful to explain and model certain aspects of software ecosystem evolution. The term co-evolution has

been borrowed by software engineering researchers on numerous occasions and for various purposes, but only at a very shallow level. A typical usage is to reflect the need for different types of software artefacts (e.g., design models and code) to be kept synchronised while they are changing from one version to the next [188, 239, 285]. Chapter 2 of this book discusses the need to co-evolve software models and their metamodels. In the context of open source, Ye et al. [937] explored the co-evolution between software systems and their developer communities. Yu [939] has studied the co-evolution between 12 kernel modules of Linux in 597 different releases and found that co-evolution arises when one module changes in response to a change in another component. Jaafar et al. [429] studied the fault-proneness of co-evolved classes in object-oriented programs. Fluri et al. [297] analyzed the co-evolution between source code and comments. Zaidman et al. [943] explored the co-evolution between production code and test code.

In the context of software ecosystems, we propose to study the co-evolution between different projects belonging to the same ecosystem. Two software projects fulfilling a similar purpose inside the same ecosystem (e. g. two games in a mobile app store, or two drawing tools or text editors in an OSS forge) can be seen as being in a state of competition. This can lead to co-evolution in the sense that a new feature in one of the projects may disavantage the other one and may force its developers to adapt the project if they want it to maintain its fitness for purpose. Similarly, if two software projects are complementary and useless if used separately, developers of both projects will need to collaborate when evolving their software. The latter scenario can be viewed as a kind of symbiosis.

10.4 Exploratory case study

In this section, we report on techniques used to study natural ecosystems and their adaptation and application to software ecosystems. We do this through a case study on the well-known GNOME ecosystem that will be presented in subsection 10.4.1. In subsection 10.4.2, we explore to which extent the characteristics of GNOME, an example of a software ecosystem, differ from the characteristics of a biological vegetation ecosystem. In subsection 10.4.3, we study the immigration of new developers in GNOME and the local migration of developers across GNOME projects, motivated by the fact that the success and sustainability of a software ecosystem depends on its ability to attract and retain developers.

10.4.1 The GNOME *OSS ecosystem*

In order to assess to which extent biological models, techniques and tools for ecosystems and evolution are applicable to software ecosystems, we need to carry out empirical studies. These studies will allow us to determine what are the main common-

alities and differences in the characteristics and dynamics of biological and software ecosystems.

To carry out such empirical studies, we need access to ecosystems that are sufficiently large (in terms of number of projects), active (in terms of number of contributors) and long-lived (in terms of number of years of activity). To avoid confidentiality issues and to facilitate reproducibility and replication of results by other researchers, we also require the analyzed data to be freely accessible. These requirements naturally lead us to OSS ecosystems. OSS is generally established as an important software development practice, and all major software vendors rely, to some extent, on OSS. In some cases, the OSS products they rely on are even critical to the company's success.

Lehman's laws of software evolution [505, 511] have inspired many researchers and have significantly influenced research on OSS project evolution [291, 330]. Many of these studies focus on understanding and predicting the evolution of individual software projects and their developer communities. Much less empirical research exists on the evolutionary study of long-lived OSS *ecosystems* containing hundreds or even thousands of projects and contributors.

As an exploratory case study, we analyse the GNOME OSS ecosystem, since it has been the subject of a lot of research in the past [320, 331, 336, 536, 640, 886]. The historical data of all GNOME projects is accessible through their Git version control repositories. We have shared our extracted data set with the research community [332]. According to `git.gnome.org`, GNOME has been under development since January 1997, and currently contains more than 1400 projects (more than half of which are archived) to which over 5000 contributors have contributed over the entire lifetime of GNOME. Table 10.1 provides some basic historical metrics for the GNOME ecosystem, obtained over a period of 15 years. Figure 10.4 gives an idea of the size distribution of GNOME's projects.

Table 10.1: Basic historical metrics for GNOME from January 1997 to December 2012. A *file touch* corresponds to the addition, removal or modification of a particular file in a particular commit.

Metric	Value
number of projects	1,418
number of projects with coding activity	1,353
number of commits	1,303,649
number of commits containing code files	685,007
number of file touches	12,394,786
number of code file touches	6,183,282
number of contributors having made at least 1 commit	5,885
number of coders (authors having made code file touches)	4,321
considered lifetime	5844 days (16 years) Jan 1997 → Dec 2012
number of considered 6-month activity periods	32

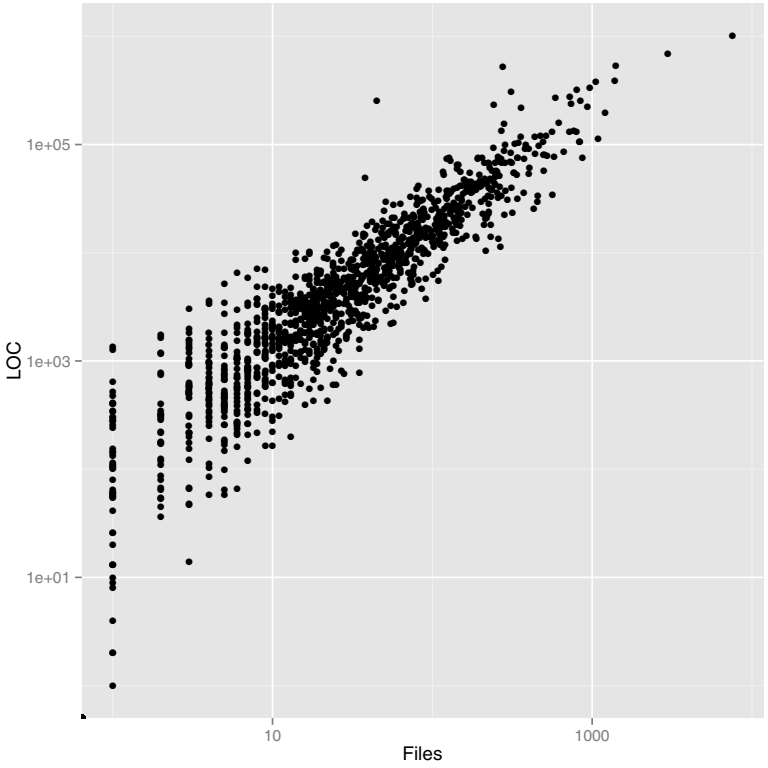

Fig. 10.4: Size (on log-log scale) in number of lines of code (LOC) and number of files of GNOME projects. Extracted using CLOC from the latest version of each git repository of January 8, 2013. Total size: 2,2251,913 LOC and 104,594 files.

In previous work [886], we have observed that the contributors to the GNOME ecosystem can be classified in different, partially overlapping, subcommunities according to their types of activity. The principal activity type of a contributor (approximated by the number of file touches of a particular type in her commits) determines to a large extent her work pattern and part of her ecological niche.

The current case study focuses solely on the coding activity. Our results will therefore be restricted to coders, code files, commits containing code file touches, and projects containing such commits. *Coders* are GNOME authors having an account and code commit activity in at least one of GNOME's Git repositories. *Code files* are files in a commit that are considered to contain source code, based on their file extension (e.g. `.java` for Java files, `.c` and `.h` for C files, `.cpp` for C++ files, `.py` for Python files, `.pl` for Perl files, and so on). Of all thirteen activity types we defined for GNOME in [886], we observed that coding was the most important activity of the frequent GNOME contributors. Figure 10.5 gives information on the usage

of programming languages across all GNOME repositories. It was extracted using the CLOC code lines counting tool (`cloc.sourceforge.net`). We see that C and C++ are by far the most frequently used programming languages in GNOME, followed at a distance by Python and C#, and then followed by Perl.

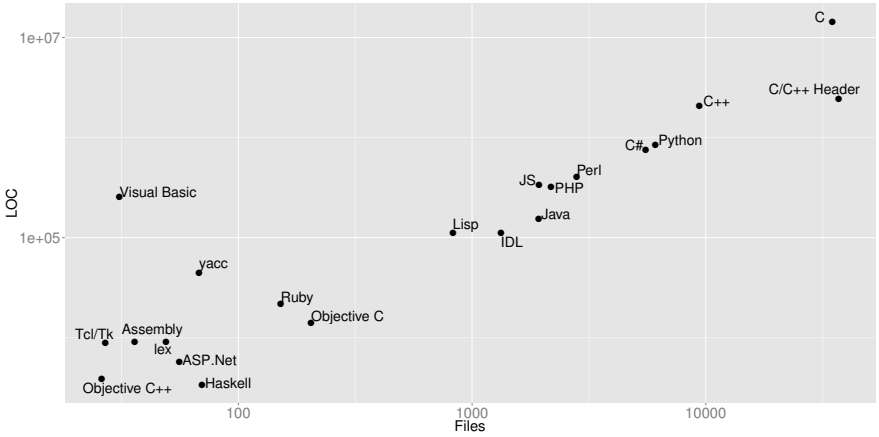

Fig. 10.5: Language usage in GNOME. Extracted using CLOC from the latest version of each repository of January 8 2013.

A challenge during data extraction is that coders may use different accounts. To avoid counting such coders as separate identities, we used identity matching. Multiple techniques have been proposed for this [108, 334, 476, 722]. We merged the different identities belonging to the same person using a semi-automatic approach. First we applied an automatic algorithm detailed in [886] and then we manually post-checked the results to remove false positives.

We chose 6-month activity periods, since GNOME has a 6-month release policy (two releases per year in March and in September). The first considered period starts on 1 January 1997 and the last one starts on 1 July 2012. For each period, we only consider commits containing at least one code file touch. Similarly, we only consider a coder to be active in a GNOME project during a period if she made at least one code commit using one of her accounts during that period. Her number of code commits for that period is the sum of the number of code commits of all her accounts for all GNOME projects during the period. The number of code file touches of a coder during a period is the sum of the number of code file touches in each of her project commits during the period. As we can observe from the boxplots in Figure 10.6, the majority of coders contribute to a single or very few projects (median value of 1, mean value of 4.866) and have a limited number of code commits (median value of 4, mean value of 156.4). The distributions are strongly skewed with a long tail.

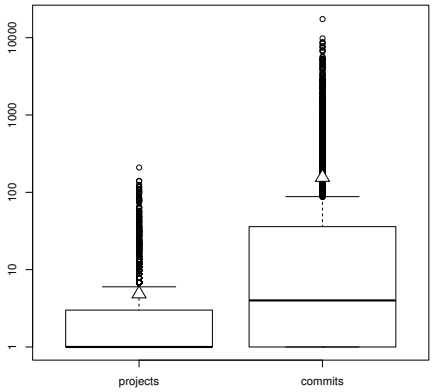

Fig. 10.6: Boxplots showing the distribution of projects and commits per coder. (The white triangle shows the mean value.)

10.4.2 Comparing GNOME *with a natural ecosystem*

In Section 10.2.3 we presented different ways to compare natural ecosystems to software ecosystems. There is, however, another useful analogy that we can draw. When studying natural ecosystems, such as a vegetation community of different species of plants in a forest [880], one can take samples of individual plants at different arbitrarily chosen locations (so-called sampling stations), and use this to get an idea of the coverage of the location by each species and the variation of this coverage across the ecosystem, for example in order to assess the biodiversity. For software systems, one can adopt a similar approach: randomly select a number of software projects belonging to the ecosystem, and count the coverage (in number of commits, or any other measure of activity) of each contributor to the ecosystem. In this analogy, contributors correspond to the equivalent of a plant species, and their number of commits to the project correspond to the coverage. One can then use the same portfolio of techniques as those used for studying natural ecosystems.

One such technique is hierarchical clustering. For the considered GNOME lifetime, we computed a matrix with projects (i. e. locations) as columns, coders (i. e. species) as rows and the number of code commits per coder as cell values. We have found in the boxplots of Figure 10.6 that more than half of the coders (54.5% to be more precise) were not involved in more than one project. Thus in the remainder of this section, we will ignore these "singleton" coders, since we would like to group together projects based on the similarities of their community and "singleton" coders do not provide useful information on such similarities or dissimilarities. We removed the columns containing only zeroes (i. e. projects without coding activity) and the rows with less than two non-zero cells (i. e. coders that were active in zero or only one project). This gives a matrix containing a total of 1352 projects and 1966 coders.

After applying a hierarchical clustering on this matrix, in contrast to the results for a natural ecosystem, we observe a large number of small clusters, implying that coders are much more restricted to a few projects than plants are on sampling stations, resulting into most items connected much higher in the clustering dendrogram.

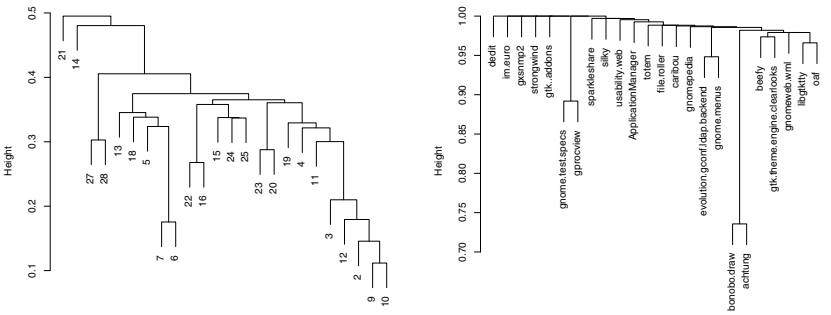

Fig. 10.7: Comparison of hierarchical clustering applied on: [left figure] a vegetation dataset at 24 randomly chosen locations on 44 plant species; and [right figure] a GNOME dataset of 24 projects chosen randomly from the fourth quartile and 44 randomly chosen coders chosen randomly from the fourth quartile.

On the left of Figure 10.7, the aforementioned vegetation community [880] measured at 24 randomly chosen locations is hierarchically clustered.[3] The Bray-Curtis distance was used as a basis for the clustering process [139]. On the right of Figure 10.7 the same hierarchical clustering technique is applied to a sampling of GNOME software ecosystem and its code contributors for 24 projects chosen randomly from the last quartile (i. e. projects with at least 283 commits) and 44 coders chosen from the last quartile. The values of 24 projects and 44 coders were chosen so that the clustering contains the same amount of species and locations as the vegetation ecosystem data.

From this comparison, we observe that a vegetation ecosystem seems to behave quite differently from a software ecosystem. The survival strategy of plants is to be as ubiquitous as possible at all locations of the ecosystem (through direct competition for sunlight and other nutrients with the other plants in its direct surroundings). In contrast, the survival strategy of code contributors appears to be by specialising themselves in very few projects of the software ecosystem. As such, there is much less competition with the other coders, and the dynamics of the ecosystem are based primarily on collaboration, as opposed to competition with other coders.

[3] We applied a hierarchical clustering with single linkage using the R function `hclust`.

After ignoring all coders that are involved in a single project, and carrying out a hierarchical clustering on *all* GNOME projects, we observed an interesting pattern: the majority of GNOME projects related to the programming languages Perl and Python, respectively, were clustered together. The fragments of the cluster dendogram illustrating this phenomenon are shown in Figure 10.8. Hence, the programming language used in projects appears to be both a barrier limiting expansion of developers across projects, and a subdomain inside which developers tend to interact more closely. This allows us to confirm and further refine the notion of ecological niche for GNOME code contributors.

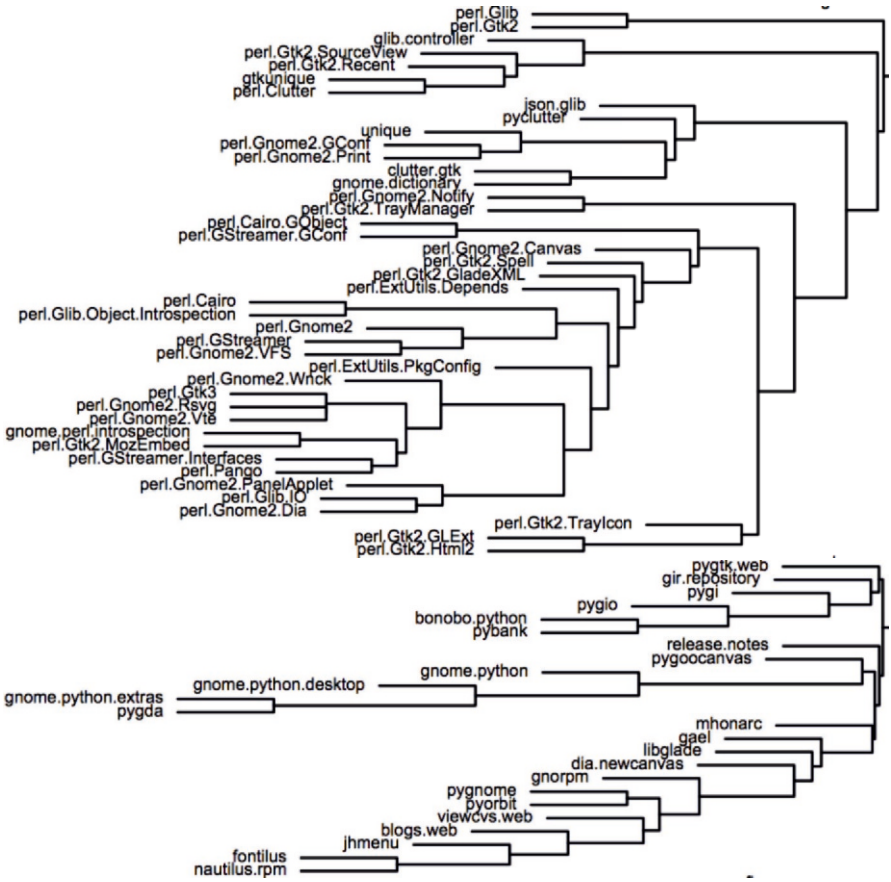

Fig. 10.8: Zoom on two interesting clusters (representing the communities of Perl coders and Python coders, respectively) in the dendograms obtained through hierarchical clustering of GNOME project and coder data. Those clusters contains a majority of Perl and Python projects. This shows that those projects' communities are very similar and tied.

Another technique frequently used for studying natural ecosystems is *principal component analysis* (PCA). Figure 10.9 again compares the vegetation community measured at 24 randomly selected locations to the coder's commits measured at 24 randomly selected Gnome projects. The PCA is carried out on correlation matrices in both cases. Figure 10.9 shows how the total variance decreases among the first 10 principal axes. On the left, we observe that the vegetation data can easily be reduced down to the first three axes while loosing less that 20% of the total variance. This means the dataset is highly structured with essentially three degrees of freedom in the distribution of the vegetation. On the right, we do not observe an important decrease of variance of the 10 principal axes for the Gnome dataset. The variance is therefore more homogeneously distributed, meaning there are rather different groups of coders working on each of the 24 projects. This confirms our previous findings that, in contrast to the vegetation ecosystem, GNOME has a relatively well-balanced community.

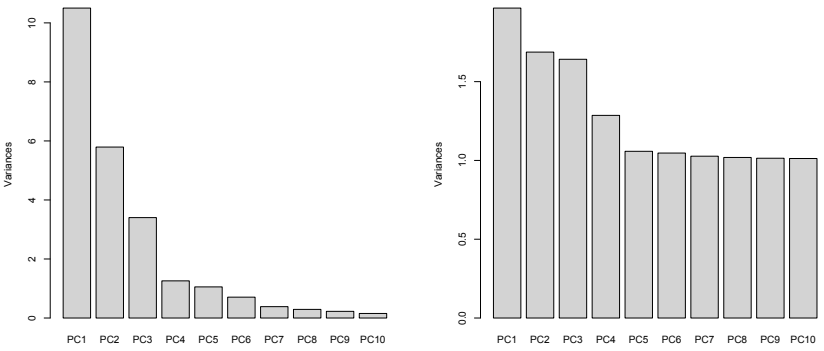

Fig. 10.9: Comparison of the variance of the first 10 principal components of PCA applied on the biological vegetation dataset (left) and the GNOME dataset (right).

To summarize, the results we obtained for GNOME are quite different from what one typically observes in natural ecosystems, where there is a high degree of competition between the species. This usually leads to well-differentiated subcommunities with identifiable key species that largely structure the whole dataset, leading to well-separated clusters in the dendrogram and to most of the variance caught by the few first principal components in the principal component analysis. It remains to be seen if this major difference with natural ecosystems is found in other software ecosystems as well. If this turns out to be the case, the traditional biological evolutionary theories (such as Darwinian evolution) are probably not applicable to OSS evolution, because of the much lower level of competition observed, while competition is an essential driver of biological evolution. Future studies on other software ecosystems will allow us to shed more light on this issue.

10.4.3 Migration of GNOME *developers*

The process of intake (also known as immigration) and retention of developers to OSS projects has been the subject of study by many researchers. Von Krogh et al. [903] have studied how one can join a project, get write access to a source code repository and then how the newcomer specialisation is related to contribution barriers. Canfora et al [160] have designed an approach to identify which contributor could be assigned as a mentor to a newcomer. Zhou and Mockus [950, 951] have shown that the social environment impacts both rate at which people joins a project and the chance that a new developer becomes a long-term one.

The reason for this interest is that the success and sustainability of a project depends on its ability to attract and retain developers. There is a crucial difference with natural ecosystems, where populations of individuals can create new generations through reproduction. In OSS projects, the only way to increase or renew the population is to attract new contributors from the outside. If a software ecosystem is not interesting enough, it will not attract new developers, or worse it may even loose its developers to other systems.

New developers are interested in joining OSS projects for variety of reasons, such as personal interest in, need for the software, increasing their personal reputation, out of altruism or because they are being paid for it [114, 292, 369, 400, 637].

Little empirical studies exist, however, on the migration of software developers across projects. Weiss et al. [911] studied the emails exchanged by the contributors of the Apache projects for discovering simple migration patterns between projects and from the outside to a project. They observed that many developers joining a project come from another project. These developers tend to migrate together with their workmates. Based on three case studies (Apache web server, Postgres and Python), Bird et al. [109] found three factors that influence immigration, i. e., intake of new developers: their technical commitment, skill level and social status. Among others, they found evidence that demonstration of skill level by submitting patches to known bugs will increase the likelihood of becoming an official developer of the project.

Jergensen et al. [439] studied how GNOME developers start using social mediums and move progressively to socio-technical and technical mediums. They tried to see if migrating from one project to another could result in bigger centrality of the developer in the newly joined project.

Due to the little studies of developer migration at the level of software ecosystems, we started to study the effect of the intake, retention and loss of developers at the level of individual projects of the GNOME ecosystem. For each 6-month activity period we counted the number of *joiners* and *leavers*. We distinguished between *local joiners* to a project (resp. *local leavers*) and *global joiners* (resp. *global leavers*). Local joiners are incoming coders in the considered project that were not active in this project during the preceding 6-month period, but that were involved in some activity in other GNOME projects instead. Global joiners are incoming coders in the considered project that were not active in any of the GNOME projects during the preceding period. A similar definition holds for the local and global leavers.

The formal definition of these metrics is given in Equation 10.1. Let p be a GNOME project, t a 6-month activity period, $t-1$ the previous period, c a coder, *Gnome* the set of GNOME's code projects, and $isDev(c,t,p)$ a predicate which is true if and only if c made a code commit in p during t:

$$
\begin{aligned}
&localLeavers(p,t) = \\
&\quad \{c | isDev(c,t-1,p) \wedge \neg isDev(c,t,p) \wedge \exists p_2 \ (p_2 \in Gnome \wedge isDev(c,t,p_2))\} \\
&globalLeavers(p,t) = \\
&\quad \{c | isDev(c,t-1,p) \wedge \forall p_2 \ (p_2 \in Gnome \Rightarrow \neg isDev(c,t,p_2))\} \\
&localJoiners(p,t) = \\
&\quad \{c | isDev(c,t,p) \wedge \neg isDev(c,t-1,p) \wedge \exists p_2 \ (p_2 \in Gnome \wedge isDev(c,t-1,p_2))\} \\
&globalJoiners(p,t) = \\
&\quad \{c | isDev(c,t,p) \wedge \forall p_2 \ (p_2 \in Gnome \Rightarrow \neg isDev(c,t-1,p_2))\}
\end{aligned}
\tag{10.1}
$$

Fig. 10.10: Historical evolution (timeline on x-axis) of the number of local (solid) and global (dashed) joiners (y-axis) for three GNOME projects.

We did not find any general trend, the patterns of intake and loss of coders are highly project-specific. Figure 10.10 illustrates the evolution of the number of local and global joiners for some of the more important GNOME projects (the figures for leavers are very similar). For some projects (e. g. `evolution`) we do not observe a big difference between the number of local and global joiners, respectively. These projects seem to attract new developers both from within and outside of GNOME. Other projects, like GIMP (a popular image manipulation program that can be used and installed separately from other Gnome applications), attract most of its incoming developers from outside GNOME. A third category of projects attracts most of its incoming developers from other GNOME projects. This is the case for GTK+ which can be considered as belonging to the core of GNOME. This observation seems to suggests that libraries, toolkits and auxiliary projects attract more inside developers, while projects that are well-known to the outside world (such as GIMP) attract outside developers.

However, it is also important to measure if the projects that attract developers from the outside of the ecosystem tend to keep those developers inside the project or also "diffuse" them to other projects of GNOME. In order to give an idea of this

on the three previously mentioned projects we defined a metrics we called the *collaboration factor* of a project. It represents the percentage of coders contributing to the project and who are also contributing to another project of GNOME. The collaboration factors for Evolution, GIMP and GTK+ are respectively 65.1%, 85% and 94.7%. This leads us to think that while GIMP attracts a lot of people from the outside of GNOME it seems that its community is not integrated into the GNOME community as well as other projects like GNOME. At the opposite, the GTK+ community appears to be more integrated in the GNOME community, which is probably not surprising since GTK+ is the core user interface library which is used by all GNOME end-user programs. It is worthwhile to study this phenomenon in more detail to find empirical evidence of this. One might consider, e. g., concentration of project participants' contributions to projects within the ecosystem which can be measured using inequality indices (cf. [768, 881, 888]). Presence of many developers with highly concentrated contributions would suggest low integration within the community.

10.5 Conclusions

This chapter presented an in-depth analysis of the analogy between natural and OSS ecosystems, from the evolutionary point of view. While there are many similarities between both types of ecosystems a lot of differences can be observed.

From a technical viewpoint, many techniques and models that have been proposed and used in ecology may provide new insights for the study of evolving software ecosystems. Some examples of techniques are the use of phylogenetic trees and cluster dendograms. Some ecological models, such as the dynamic predator-prey model have already been adapted with success in a software evolution setting [155, 500].

Some other models, even after adaptation, appear to give different results when applied to OSS ecosystems. For example, for the GNOME ecosystem there appears to be a much higher degree of collaboration than what is found in many natural ecosystems, and a lower degree of competition. For such collaborative ecosystems, the more recent hologenome theory of evolution that has been proposed to explain the evolution of coral reef ecosystems [732] may perhaps be closer to how software ecosystems evolve, since it considers the evolving organism together with its associated communities, just like a software project co-evolves by the grace of its associated user and developer communities.

Because the traditional biological evolutionary theories are essentially driven by competition between species in a shared resource pool, they are not always readily applicable to explain the dynamics of highly collaborative OSS ecosystems. Other, more business-driven proprietary software ecosystems, such as the app stores for mobile devices, are likely to have a higher degree of competition since all apps struggle for a larger market share in order to increase their profits. The developers of commercial software ecosystems are also remunerated, while contributors to OSS

ecosystems often work on a voluntary basis and usually have no direct financial benefits from their involvement.

The main challenge is that historical data of commercial software ecosystems is much harder to obtain, making it difficult to study evolutionary theories on such ecosystems. OSS ecosystems like GitHub and SourceForge do not have this limitation and probably fall somewhere between both extremes, with some amount of competition but also a certain degree of collaboration.

Seen from a complex systems viewpoint, OSS ecosystems seem to be closer to their biological counterpart than business software ecosystems [435]. Commercial ecosystems are typically governed by a decision maker that decides how the ecosystem should evolve, while OSS ecosystems often have a much more flexible decisional structure. Like in biological ecosystems, decisions are taken at the level of individual species (read: projects), with an emergent overall effect on the software ecosystem as a whole.

In the current state of software ecosystems research, it is still too early to make any general conclusions, and much more empirical results are required to understand how one can benefit the most from existing research on natural ecosystems.