

Towards an EDOS API: Modelling the F/OSS Process

Michel Pawlak, Ciarán Bryce, Michel Deriaz
University of Geneva

Draft: April 28, 2005

Abstract

The goal of EDOS is to improve F/OSS production and code distribution processes. A prerequisite to this is to formally distinguish the different concepts in the processes, the roles undertaken by users, the data types used, and their inter-relationships. This allows one to determine the data that must always be distributed together from those that need not be; from a security and management viewpoint, it formalises the privileges and information that a particular user requires at any given time. This note proposes elements of a model for F/OSS that makes roles and data types explicit. We hope that it can serve as a basis for future discussion.

1 Introduction

The goal of EDOS is to develop mechanisms that improve the performance of current F/OSS project processes. Several process weaknesses have been cited by EDOS partners: the error-prone nature of package dependency management, the excessive project server traffic that follows the announcement of a new release, the *ad hoc* and error-prone transfer of project data between mirror servers, and the vulnerability of project servers to security attacks.

A pre-requisite to addressing these concerns, and thus improving F/OSS processes, is to attribute greater precision and clarity to the concepts of F/OSS, the roles of users, the different data types manipulated and their interdependencies. Doing so allows us to understand the exact information required and produced by each F/OSS participant at any given time, his responsibilities and the privileges he requires. Understanding these *information flows* allows us to determine the data that needs to be distributed together from that which can be distributed separately. This clarity is invaluable for the design of the distribution architecture. More specifically:

- Analysis shows that there is no user role – outside of the project organisation – that requires simultaneous access to all project data types (e.g., documents, code units, test suites, etc.). There is therefore no reason to prioritise a distribution architecture where all project data is stored on a central server, or where mirrors keep a copy of all project data. A central server is in fact counter-productive since participants requiring independent data needlessly interfere with each other by competing for server network bandwidth.

- One can distinguish several data types in F/OSS. For instance, the *package* concept is related to three different data types: a unit of content (e.g., manual, program, etc.), a group of content units that form an application, and a group of content units – perhaps from different projects – that form a release. The notion of test can be broken down into types that are associated with each of the package kinds. By defining who creates and who needs access to instances of each data type, we can distinguish the essential information flows in the system from the superfluous flows. For instance, a developer who bundles a group of content units together needs to see the test results for that bundle, but does not need to see test results for others. By automating the essential information flows, some of the latency associated with current F/OSS processes can be eliminated, and the quality of information improved.
- A key notion in the field of information management systems and security is *separation of privilege*. This means that a user who is engaged in a task be assigned only the minimum privileges required for that task [1]. In the context of EDOS, there is a clear distinction between the tasks of testing a package and package committer. These roles need have no privileges in common, meaning that the EDOS process should not allow a tester to modify code, or allow a committer to construct test results. Separation of privilege has several advantages: it clarifies further the information flows and reduces the risk of an insider attack succeeding on the project’s code base since a committer cannot fabricate test results (and a tester cannot modify code).

This short paper proposes aspects of a model and its API for F/OSS. The model tries to be precise about the data types and roles and on their inter-relationships. It is designed to be applicable to a large number of F/OSS projects. We motivate the model’s design in further detail, and discuss both the organisational and technical implications of the model. Section 2 analyses some of the weaknesses in F/OSS processes, and their implications for the EDOS API. The model is presented in Section 3. Related work is presented in Section 4 and conclusions in Section 5.

2 Analysis of F/OSS Processes

Background

An overview of the F/OSS production and code distribution processes is given in [2]. The distinguishing feature of F/OSS compared to proprietary software projects is that it is a community effort with perhaps a large number of participants. Currently, we tend to identify a project management structure, or editor, whose role is to make decisions on the evolution of the project. A project’s community involves *developers* who write code which may be written for a specific project or for general use. In the latter case, the code is placed in the public domain, on *freshmeat* for example, from where it is taken by one or more F/OSS projects. Each project has a set of *committers* who are responsible for making modifications to the project’s code base. Committers may integrate developers’ packages in their submitted form, or they may make their own changes after testing the packages. There are many F/OSS projects today; some are dependent since they include other project releases in their own release, e.g., MandrakeLinux includes OpenOffice.

Information Flows

One noticeable feature of F/OSS projects is the multiplicity of *roles* undertaken by participants, particularly committers. One task of the latter is to build packages; another is to build releases of the project's software, and yet another is to test packages before these are added to the project's code base. A developer is responsible for at least three tasks: he can produce code, signal bugs or run tests. However, these different roles are not clearly distinguished in current F/OSS projects. That is, a participant uses the same account – with the same privileges and information access – for each task.

Each F/OSS task can be classified by the information it requires access to, by the information it generates and by the participants who need to gain access to the generated information. Thus, each task has a precise *information flow*. It is important to identify essential information flows in the process for the distribution architecture design. For instance:

- Producing a package requires access to the content being packaged, and the information generated includes a formal package, package description information and a license. This information is used by committers who are responsible for building an application and for package testers. On the other hand, the information has no importance for end-users who are only interested in the release.
- Building an application requires access to a set of packages, each containing code or documentation, and the information produced includes a formalised description of dependencies, the formal bundle of related packages and an envelope license. This information is used by application testers and release builders.
- Testing requires access to benchmark suites furnished by the project management. This task produces verifiable test results that only need to be fed back to the developers of the packages being tested.

Content Types

The core data type F/OSS is the *unit of content* produced by a developer. This can be source code, binary code or documentation. There is no real difference from the process viewpoint between documentation and code. For instance, some of the OpenOffice developers produce natural language translations of user manuals rather than code; their content gets packaged, tested and linked to others in the same way that code from developers gets treated.

Another data type is the group or *bundle* of unit content packages that form an application. The unit content packages of an application contain code, libraries and documentation. The links are important to the application developer, but not to the developer of the component unit content packages. Yet another is a project release – or set of project software installed on the end-user machine – which is also a set of packages, but may use packages from outside of the project, i.e., from another F/OSS project. Thus the important information includes license information and project links. These links are important since changes within one project impact on others.

An end-user is concerned with a suite of applications that he wishes to see running correctly on his machine. He does not want, or need, to know about packages. Each user has a *profile* that

describes his preferences in terms of applications to install, the degree of software stability, and the level of customisation.

Perhaps the most fundamental data type in F/OSS is the *license*. This is **the** distinguishing characteristic of F/OSS compared to proprietary software. It is essential that the F/OSS process be able to manage licenses. License management means ensuring that each unit package has a license, that each application package bundle has a license, and that this license does not conflict with the licenses of the individual unit packages.

Evolution

Another crucial feature of F/OSS is the strong presence of evolution. New projects get created by Internet community members all the time, and existing projects can take new directions. A project may decide to develop new applications, and these applications evolve also to accommodate new functionality or bug fixes. The community involved in a particular F/OSS project also evolves: developers arrive, some eventually get promoted to committers; committers can take time out to test packages, to cater for a new release, or may even leave the project in order to form a new one. The API must be able to express and control such changes to the project's structure. In a project following a meritocracy structure for instance, the API offers the functionality for securely executing and auditing a vote.

3 EDOS Model Elements

This section proposes a model core for the F/OSS process. We start with an overview, before giving some detail that illustrates with out thinking has been on this matter over the last few weeks.

3.1 Overview

The EDOS model formalises data types and roles which we consider in turn. All key terms – which are represented as classes in the API – are written in **sans serif** font.

3.1.1 Data

A F/OSS Project distributes software to the software public. The software is generally programmed by a group of developers who submit their software directly to project committers, or indirectly via public domain software sites like *freshmeat*. Each developer may work on a specific component. A Project may depend on other projects, by including releases in its own distributed software.

A **ContentUnit** is the smallest unit of production in the project, and is produced by a single developer. It can contain code or documentation. The nature of the content is not that important, since all content units have certain features in common: a developer or committer, and a series of tests to apply (e.g., unit tests for code, integrity tests for documentation). The contents of a **ContentUnit** can be directly installable or it may require pre-processing for installation, in which case the unit is respectively classified as *binary* or *source*.

A `ContentUnit` is not distributed by a project. Rather, it belongs to and is incorporated into an application. Thus, it is distributed along with the other `ContentUnits` of the same application. An application may employ several `ContentUnits`, e.g., to contain libraries, binary or source code and documentation. A `Bundle`, once composed, is associated with a `License` that covers all units of the bundle.

A `Bundle` is a recursive structure. This captures the fact that different components of the application may be covered by different licenses, e.g., a project specific license and an LGPL license. In this case, there is a bundle for each different license, where each bundle contains the distribution units covered by that license, and the related bundles are packed into a envelope bundle with a project specific bundle license. An example of recursion in an application bundle is a sub-bundle that contains documentation. Documentation today could be in multimedia form, e.g., a video presentation, in which can the documentation bundle can itself have a sub-bundle for a video player application.

The software that a `Project` offers to the software public is termed an `InstallableUnit`. This can include several applications, which corresponds to several `Bundles`. It is more than a bundle since it may contain bundles from different and independent projects. A `InstallableUnit` is what the end-user sees; it corresponds to his environment, e.g., it contains the applications that the end-user in question requires or the object code contained may be compiled for his target architecture. In any case, a project may produce a number of `InstallableUnits`. A `Release` is an `InstallableUnit` that the project management qualifies as *stable*.

An end-user in F/OSS processes installs `InstallableUnits`. He is quite an important individual in F/OSS since, ultimately, his software preferences determine what F/OSS projects succeed, and what software these projects produce. The preferences of an end-user is encapsulated in a `Profile`. It should be possible to map a profile to a set of `InstallableUnits` that belong to a project.

The distinction made between `ContentUnit`, `Bundle` and `InstallableUnit` impacts on the definition of other data types. For instance, `Test` and `TestResults` are specialised to unit, bundle and installable unit since the nature of the tests differ, as do the participants who do the tests and the participants who need to see and act on the results. For instance, the test for a `ContentUnit` might be a standard input/output test if the unit contains a program, or a spell-check if the unit contains an ASCII manual. The test for a bundle might be a compilation test against different operating systems. The test for an `InstallableUnit` might be a test against different platform architectures.

3.2 Working Method Employed

To give more meat to our presentation, we describe here the approach that we have been using to define the types and roles appearing in F/OSS code distribution. The technique is known as a *conceptual map* [3]. This is a graphical technique where we start by placing the key concepts of the process being modelled, and then identifying those which are linked. In a second phase, we add further concepts to qualify the meaning of these links, for example, the concept of `ProjectConvention` (see Figure 1) expresses the nature of the conventions that a project adopts for its content. In a third phase, we draw *information zones* – represented by colours in the graph – that identify the concepts that are closely related based on the information flows and roles appearing in the zone. For example, there is an information zone for testing content units. The combination of the information zones depicts the knowledge of the field being modelled (F/OSS in our case).

The conceptual map can then be used to build UML Use Case Diagrams, Class Diagrams, etc. The key to this use is that each information zone can be refined to class diagrams or other implementation forms independently of other zones. The separation of concerns is thus already present in the global design phase, and remains present through the whole development, implementation and operation phases.

There are three principal reasons for taking this approach.

First, as stated before, each information zone is associated with a precise role, with particular responsibilities for executing a given task and with information flows. Describing the concepts involved in each information zone shows exactly what information is needed for this role, and what the boundaries of the role are.

Second, the map shows clearly that some information zones overlap, and that some concepts of the zone are completely independent from the rest of the map. Overlapping information zones imply that concepts in the intersection need to be analysed as they affect more than one zone. If overlapping zones cannot be avoided in the design, then one of the intersecting zones has to be attributed ownership of the involved concepts – or a new information zone must be defined to cover the intersection. The owning zone is the only one in which refinement and development of concepts takes place. Other information zones are simply clients of these concepts and have to negotiate modifications if needed. In Figure 1 for instance, the concept of project convention appears in several zones: installable unit, content unit and bundle. Ownership is attributed to a new zone whose aim is to globally manage the design, implementation and operation of the conventions of a project independently of the zones that use these conventions. In contrast, non-overlapping parts are internal to each information zone, and so can be managed with no risk to the development of others.

Third, the addition of a new concept – and its interaction with existing concepts – can be easily modelled. An addition may imply modifications of existing information zones or even the creation of a new specialized zone. This feature is probably the most important as it allows at any moment to evaluate the global impact of the addition of a new concept to the whole model. This change cannot easily be modelled by formalisms that insist on use case scenarios as this is too focussed.

Below is a short description of some of the information flows illustrated in Figure 1.

Project Management This information zone defines the boundaries of a **Project**. The set of produced **InstallableUnits**, **ContentUnits** and corresponding **Bundles** and sub **Projects** is indexed within this information zone.

Project Conventions Management All produced content within a **Project** have to respect a set of chosen **Conventions** associated with the **Project**. These **Conventions** can have many types, e.g., coding conventions, documentation conventions, design and format conventions, etc. This information zone deals with issues related to the definition and the management of **Conventions**.

ContentUnit Management **ContentUnits** are bound to a **Project** and have to respect the **Conventions** chosen for this **Project** as well as a **License**. As stated before, **ContentUnits** can contain any type of content like code, documentation, images, etc. A **ContentUnit** can be designed for a **TargetArchitecture**. Note that we do not yet distinguish source and binary content as we consider

this information as being a feature of the `ContentUnit`.

Bundle Management The Bundle Management information zone shows that in order to build a `Bundle`, `ContentUnits` which will be part of it have to be known. The units of the `Bundle` can be retrieved at any time. A `Bundle` can be part of another `Bundle` and is bound to a `Project`. This recursive nature of `Bundle` allows hierarchical building of components. A `License` has also to be chosen for the `Bundle`. As the overlap with the `ContentUnit` information zone shows, we need to keep in mind that the chosen `License` has to be compatible with the `Licenses` of all component `ContentUnits`. Further, the information zone has to verify that the `Licenses` of all parts of the `Bundle` are compatible with each other. Finally the `Bundle` has to respect the `Conventions` associated with the `Project`.

InstallableUnit Management `InstallableUnits` are made of chosen `Bundles` which belong to the `Project`. To include features provided by other projects, `InstallableUnits` can use other `InstallableUnits`, thus creating a dependency link between them. When an `InstallableUnit` is in a stable state, it can be declared as a `Release`. Again, as for `Bundles` and `ContentUnits` a `License` has to be chosen. The overlap with the Bundle Management information zone shows that this `License` has to be compatible with the `Licenses` of all used `Bundles`. Further it is mandatory that all `Bundle` licenses be compatible with each other and the `License` of an `InstallableUnit` be compatible with the `Licenses` of the `InstallableUnits` it depends on. Finally, this information zone has to ensure that the `InstallableUnit` respects chosen `Conventions` for its `Project`.

License Compatibility Management Previous information zones indicate that a central point in defining the content of an `InstallableUnit` which can be distributed to the community of users is the license choice and all issues related to the verification of license compatibility. Thus it may be interesting to centralise this responsibility in order to offer consistency to the whole project and provide security from the legal point of view.

Figure 1 also includes six other information zones for issues related to the testing and debugging of `ContentUnits`, `Bundles` and `InstallableUnits`. As the scope of these units as well as their nature is different, the type of tests will be also extremely different. These points are currently being developed.

3.2.1 Roles

The key to defining the process roles is to separate work on a `InstallableUnit`, `Bundle` and `ContentUnit`. Each abstraction has a task for a leader, developer, committer and tester. A role can have a concrete representation in the F/OSS process. For instance, it may be represented by different tools or even different accounts. Thus, when a committer – in the current process – wishes to work on a specific bundle, he runs the tools for, or logs in as, `Bundler`.

A `User` is someone who uses a `InstallableUnit` but who does not contribute code. He may however contribute a `BugReport` for the installed software or suggest new features via project forums.

A `Developer` writes new features, code, patches or documentation. There need be no formal link between him and the project. He might not even be aware that his contribution is used by

a particular project. A `Committer` creates or modifies a `ContentUnit` in some part of the project's code repository. A `Tester` can be anybody that wants to test content units. He needs read access to the code repository where content units are stored.

A `Bundler` creates a `Bundle` from `ContentUnits`, enters it into a project's code repository and attaches a license to it. A `BundleLeader` analyses the needs of an application and defines which functionalities need to be implemented. He designs how the `Bundle` should evolve and split the task into sub-tasks. A `BundleTester` applies a battery of tests to a bundle and writes the test results.

A `InstallableUnitMaker` creates a `InstallableUnit` from a `Bundles` and other `InstallableUnits`. These can be downloaded and installed by users. A `InstallableUnitCommitter` is responsible for creating an `InstallableUnit` in the code repository. A `InstallableUnitTester` runs tests and creates test results.

A `ProjectLeader` is responsible for the coordination of the project and the hosting of the files. A `LicenseManager` is responsible for managing information of all licenses used anywhere within the project. A `Conventionist` is responsible for defining the conventions used within the project.

The following table indicates the privileges associated with key roles, and also mentions who is responsible for nominating participants to project roles.

	User	Developer	Committer	Tester	Bundler	Bundle Committer	Bundle Tester	Bundle Leader	InstallableUnitMaker	Project Leader
Read access to the bugs repository	y	y	y	y	y	y	y	y	y	y
Write access to the bugs repository	y	y	y	y	y	y	y	y	y	y
Read access to Distribution Units		y	y	y	y					
Write access to Distribution Units			y							
Read access to Bundles						y	y	y		
Write access to Bundles						y				
Read access to InstallableUnits	y								y	y
Write access to InstallableUnits									y	
Read access to Releases	y									
Write access to Releases										y
Nominating people to Developer			y							
Nominating people to Committer					y					
Nominating people to Tester			y							
Nominating people to Bundler						y				
Nominating people to Bundle Committer								y		
Nominating people to Bundle Tester								y		
Nominating people to Bundle Leader										y
Nominating people to InstallableUnitMaker										y
Nominating people to Project Leader										y

4 Related Work

Red Hat Network (RHN) is an architecture whose design is also articulated around an API [8]. Its scope is narrower than ours, since it essentially focuses on code distribution. Binaries in RHN are provided via a customised management architectural solution that customers also use RHN to download distribution ISO's, errata (patches) and software packages. Clients that subscribe to RHN can automatically update their system in a customised way. The network is accessible through an *Access API*. The key abstraction is the *channel*, which corresponds to a set of packages and every client machine that is connected to a specific channel can be updated when the content of the channel changes. Channels can be created and managed by the system administrator. One possibility is for him to associate access rights with a channel and thus control the local systems that read from it.

A channel can be used to implement a *staged environment*. Along with the base channel that corresponds to the core system, other types of channels exist. A development channel is used by developers of the community to distribute their work. A Testing & QA channel is used to report on the packages under development and for bug reports. A production channel is used to develop beta versions. The architecture allows actions to be defined on channels by users. An example action could be to remove packages whenever a new version is available, or to rollback to a previous version of the system when a compilation error occurs. Another use case is a system administrator that downloads new patches and tests them on specific machines. If the test passes, he copies the updates in the production channel, where the users' machines are connected.

A discussion of the motivation of contributors to F/OSS projects is presented in [7]. The author argues that the motivation is not just computer science, but the reasons are equally related to economics, law, psychology and anthropology. Building a new code distribution architecture, or changing the internal structure or policy in an existing project could therefore have a strong impact on the success of a project. The author recommends that all factors that motivate volunteers to contribute to a project be considered when reorganising the architecture.

An overview of how F/OSS projects are organised is presented in [6]. The paper explains key terminology and overviews the main processes. The processes include decision-making within the project management, accountability of bugs to packages, communication among developers, generation of awareness about the project in the software community, managing source code, testing and release management. The paper also compares well-known F/OSS projects like Apache, Linux kernel and Tomcat with respect to their treatment of these processes. A related paper is [5] which describes techniques and issues in modelling the software processes used within three large open source software development communities: Apache, Mozilla and NetBeans.

In [4], the authors talk about collaboration, leadership, control, and conflict negotiation in the Netbeans.org community. The lessons are applicable to any F/OSS project. The authors claim that there are at least three kinds of issues that arise in F/OSS management: collaboration, leadership and control, and conflict. These are aspects that need to be considered for an EDOS API. An interesting point is that people outside the community, like users, former users or potential users, do not use the NetBeans.org community message boards to express themselves. Rather, they prefer to do it with other well known tools. This suggests that it is important to use standard tools, even if they are not necessarily the best.

5 Conclusion

This short paper has outlined a proposal for the EDOS API. The note reflects our thinking on the matter over the last few weeks. The main aim of the model is to insist on the different roles of F/OSS participants and on the data types. This helps to clarify data organisation issues as well as general and security administration of F/OSS.

References

- [1] Ross J. Anderson. *Security Engineering — A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2001.
- [2] Radu Pop et al. Code Distribution Process - A Definition of Evaluation Metrics. EU IST EDOS Project Deliverable, February 2005.
- [3] Gilles Falquet, Michel Leonard, and Jeanne Sinda Yamaze. F2concept: a Database Extension Language for Managing Class Extensions and Intensions. Final report, University of Geneva, May 1993.
- [4] Chris Jensen and Walt Scacchi. Collaboration, leadership, control, and conflict negotiation in the netbeans.org community.
- [5] Chris Jensen and Walt Scacchi. Process Modeling Across the Web Information Infrastructure.
- [6] Richard N. Taylor Justin R. Erenkrantz. Supporting Distributed and Decentralized Projects: Drawing Lessons from the Open Source Community.
- [7] Maria Alessandra Rossi. Decoding the Free/Open Source (F/OSS) Software Puzzle - a survey of theoretical and empirical contributions.
- [8] Sean Witty. Best Practices for Deploying and Managing Linux with Red Hat Network.