Adding Ownership Domain Annotations to and Extracting Ownership Object Graphs from Apache FtpServer

Radu Vanciu **Marwan Abi-Antoun**

December 2011

Department of Computer Science
Wayne State University
Detroit, MI 48202

Keywords: architectural extraction, runtime architecture, architectural conformance, horizontal conformance, vertical conformance, reference architecture, FTP server

Abstract

A runtime architecture could greatly help software developers and maintainers by describing objects and their interactions at runtime. As software products evolve, the architectural details may not be properly maintained in the existing documentation; therefore, inconsistencies may appear between the architectural view and the code.

Extracting a runtime architecture from the code requires a hierarchical organization of objects. Since object hierarchy is not directly observable in the code, we use ownership domain annotations. A sound static analysis analyzes the annotated code and extracts an ownership object graph (OOG). By refining the annotations, we can push objects that express implementation details underneath more architecturally relevant ones. The refined OOG conveys architectural abstraction and we can check if the OOG conforms a documented architectural view drawn by developers.

In this report, we describe the use of ownership domain type system on the legacy code of Apache FtpServer, which is a mature software system developed by experienced programmers. We use an existing analysis to extract the OOG and iteratively refine the OOG until it conforms to a standard architectural pattern or to a reference runtime architecture available in the documentation.

The results show that using ownership domain annotations an analysis can extract an OOG that conforms to the developer's design intent.

Contents

List of Figures

List of Tables

1 Introduction

When a runtime architecture is outdated or missing, an architectural view extracted from the code could greatly help developers by describing objects and their interactions at runtime. An ownership domain is a conceptual group of objects with an explicit name and explicit policies that govern how objects in one domain can reference objects in other domains [4]. A sound static analysis relies on ownership domain annotations to extract an Ownership Object Graph (OOG), which is a hierarchical object graph that shows a runtime architectural view of the system. An OOG shows more architecturally relevant objects at the top of the hierarchy while objects representing implementation details are pushed to lower levels [2].

In this project, we added ownership domain annotations to the source code of Apache FtpServer (AFS), a 13-KLOC open source system. To reduce the annotation burden, we used a tool to generate the initial annotations. Then, using a type checker, we checked the annotations to ensure that the annotations are consistent with the code and with each other. The type checker generated a list of warnings. After solving most of the annotation warnings, we extracted an OOG, then we refined it until it conformed to the reference architecture of an FTP Server [10].

To make the refined OOG conveys architectural abstraction we used two abstraction mechanisms: abstraction by ownership hierarchy and abstraction by types. Abstraction by ownership hierarchy allows developers to collapse the object's substructure for an object at the top of the ownership hierarchy. Abstraction by types acts within a domain to collapse objects that share a common base type into a single representative object. To avoid excessive merging, the base types are specified by developers.

1.0.1 Why AFS?

OOGs were extracted and refined for several systems. These systems were either developed by advanced students or by object-oriented experts [2, Section 7.6]. Conversely, AFS was written by open-source developers and released under Apache license. The differences can be observed, for instance, in the type structure. AFS has a rich type structure with 39 interfaces defined in the code, which allows us to use better abstraction by types in the OOG.

We summarize the reasons for selecting AFS:

- it is small enough to allow us to annotate the code semi-automatically, but large enough to show interesting results;
- it is actively maintained so that we can contact the developers;
- it has a reference architecture available that we can use to analyze conformance;
- it has a rich type structure;
- it is developed by professional developers as a stable release under Apache license;

1.0.2 Research hypothesis.

We used AFS as an extended example to evaluate the following research hypothesis:

Hypothesis: Given a legacy application to which developers add, after the fact, ownership domain annotations that are consistent with each other and with the code, developers can refine the annotations so that a static analysis can extract an OOG that achieves conformance with an architectural style and with a reference runtime architecture.

Following the guidelines in [7], we distinguished between *horizontal conformance* between similar abstractions and *vertical conformance* between different abstraction levels. We analyzed the horizontal conformance between the refined OOG conformed and a given architectural pattern, which is a collection of architectural decisions applicable to a recurring design problem [11]. In this study, we used the State-Logic-Display (a.k.a. three-tier) architectural pattern. For vertical conformance, we needed to analyze the conformance between the implementation and a reference architecture. In this study, we analyzed the conformance between the extracted OOG and the reference architecture available in the AFS documentation. The reference architecture is part of the official specification of the File Transfer Protocol [10], which is supported by the AFS implementation.

1.0.3 Research questions.

To evaluate the hypothesis, we formulated the following research questions:

RQ1: Can developers refine the AFS OOG to conform to the State-Logic-Display architectural pattern?

RQ2: Can developers refine the AFS OOG to conform to a documented reference architecture?

1.0.4 Contributions.

Our contribution is to report on our experience in:

- using ownership annotations to annotate the AFS legacy code;
- extracting an initial AFS OOG;
- refining the AFS OOG and analyzing its conformance to the State-Logic-Display architectural pattern [11].
- refining the AFS OOG and analyzing its conformance to a reference architecture [10].

2 Adding and typechecking the annotations

2.0.5 Terminology.

When we discuss the code structure, we distinguish between application classes and library classes. An application class is defined in the application source code. It is different from a library class, which is defined in an external library or in the Java standard library. The library classes may be available only in binary format. We use the same terminology for interfaces: application interfaces and library interfaces. We refer to an application type (class and interface) that is a root in its inheritance trees as a *top-level application type* (TLAT). C is a TLAT, if there is no other application class or application interface C' such that C extends C'. That is, C may extend `Object` or a library class or a library interface.

2.0.6 AFS code structure.

AFS is a Java server that implements the File Transfer Protocol (FTP) [10]. AFS is an open-source system that is designed to be a complete and portable FTP server engine solution based on

currently available open protocols. AFS can be run as a stand alone application, as a Windows service or Unix/Linux daemon, or embedded into a Java application. It also provides support for integration within Spring applications [1].

Table 1 shows information about the code structure, obtained using the Eclipse Metrics Plugin [8].

Table 1: AFS code structure information.

Lines of code	12786
Number of classes	159
Number of interfaces	39
Number of packages	38
Number of TLAT	98
Version used for case study	1.0.5

2.0.7 Preparation of the source code.

The source code available on the official website was organized as a Maven project [6], which accesses the external libraries from a central repository of libraries. We reorganized AFS as a default Java project, and copied the external libraries used by AFS in the local folder of the project. We excluded the unit tests and the harness code since we considered it not to be interesting for the OOG extraction and refinement. The unit test files were placed in a separate folder than the main source folder.

In order to use the ownership domain annotation language, we made minor changes to the code. For example, we used the Eclipse refactoring tool to convert non-generic types to generic types and to extract local variables in the presence of complex expressions currently not supported by the annotation language [2, Appendix A], due to the limitations of the Java 1.5 annotation syntax.

2.1 Adding annotations process

We followed the same annotation process as in previous studies. The process of ownership domain annotation is shown in Figure 1. We used a tool to add default annotations, we ran the typechecker, and we solved the reported problems. Next, we refined the OOG until it conformed to the design

intent we gathered from the reference documentation. While the tools reduced the burden of the process, to avoid possible tool defects, we were proactive in addressing the reported problems with the annotations. For example, when a variable was declared with the wrong number of domain parameters, we fixed the list of actual domain parameters to match the formal ones. Alternatively, we could have waited until the tools reported the problems. Early on during the annotation process, we took several decisions and reflected them in the annotations, for example, by deciding in which domain to put an object. As we built our knowledge of the system incrementally, we found ourselves reverting some of the earlier decisions, for example, by changing the domain of an object.

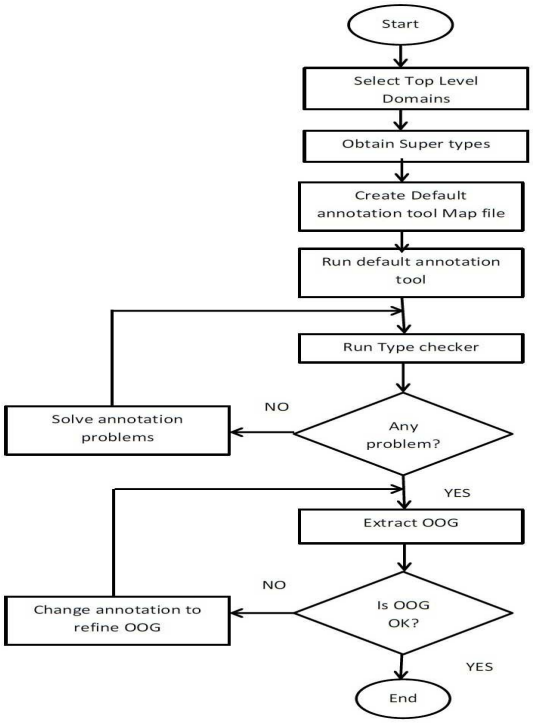


Figure 1: Flowchart of the process for adding annotations.

2.2 Generate default annotations

To reduce the annotation burden, we used the default annotation tool (ArchDefault) to generate initial ownership domain annotations. For example, the domain of local variables in a method is typically `lent`. ArchDefault detects all such local variables and annotates them with `lent`. Due

```
<model id="0">
   <mappings class="java.util.ArrayList" id="1">
      ...
      <mapping id="10" type="Command" domain="L" takeDomainParams="true"/>
      <mapping id="11" type="CommandFactory" domain="L" takeDomainParams="true"/>
      ...
      <mapping id="51" type="StatisticsObserver" domain="L" takeDomainParams="true"/>
      ...
   </mappings>
   <domainParams class="java.util.ArrayList" id="95">
     <domainParam id="96" name="U"/>
     <domainParam id="97" name="L"/>
     <domainParam id="98" name="D"/>
   </domainParams>
</model>
```

Figure 2: A fragment of the mapping between TLATs and domain parameters.

to some defects in the tool, the tool did not add all the annotations and we had to change some of the added annotations. While ArchDefault is not an ownership inference tool, it saves us typing many boilerplate annotations, for example, to declare a private domain on all the classes.

As a first step, we selected the top-level domains which contain the most architecturally relevant objects. Because we intended to analyze the conformance between AFS OOG and a three-tiered architectural pattern, we selected three top-level domains `UI` , `LOGIC`, and `DATA`. For example, all the objects related to the user interface are in `UI`, the objects that are implementing the FTP protocol are in `LOGIC`, while the objects related to the data being transferred such as files are in `DATA`.

In the second step, we selected TLATs, and we mapped each TLAT to a domain parameter and we constructed a TLAT-domain parameter map. The end of the map defined the domain parameters. Each domain parameter corresponded to one of the top-level domains: `U` for `UI`, `L` for `LOGIC`, and `D` for `DATA`. A fragment of the TLAT-domain parameter map is in Figure 2.

Since AFS used expressive names, we relied on the name of the classes to determine the initial domain. For some TLATs, their corresponding domain was straightforward. For example, `CommandLine` objects are in `UI`, while `CommandFactory` was in `LOGIC`. For others, however, it was more difficult to decide their domain. For example, an object of a type implementing the interface

`Command` could be either in `LOGIC` or `UI`, and we decided to map it to `L`. As another example, we mapped `FtpIoSession`, `FtpServerContext`, and `FtpRequest` to `D`, but we could have mapped them to `L`.

While inspecting the class names, we noticed that some of them indicated the use of design patterns such as Observer or Factory. For example, we observed that 14 class names end in `Factory`, such as `CommandFactory`, and two class names end in `Observer`: `FileObserver` and `StatisticsObserver`. As a default, we mapped the majority of these classes to `L`.

Following the TLAT-domain parameters map, ArchDefault automatically annotated the code. For example, ArchDefault annotated `CommandFactory` and `Command` interfaces, and added `@DomainInherits` for the class `AbstractCommand` which implements `Command`. The formal parameters of the method `execute` of `Command` were all mapped to the domain parameter `D`, according to the initial mapping. On the other hand, the formal parameter of `getCommand` was annotated with `lent` since `String` was not included in the initial mapping. Also, for `AbstractCommand` and for all the classes, ArchDefault declared a private domain `owned` (Fig. 3).

```
1 @DomainParams( { "U", "L", "D" })
2 public interface CommandFactory {
3     @Domain("L<U,L,D>") Command getCommand(@Domain("lent") String commandName);
4 }
```

```
1 @DomainParams( { "U", "L", "D" })
2 public interface Command {
3
4     void execute(@Domain("D<U,L,D>") FtpIoSession session,
5                  @Domain("D<U,L,D>") FtpServerContext context,
6                  @Domain("D<U,L,D>") FtpRequest request)
7                           throws IOException, FtpException;
8 }
```

```
1 @Domains( { "owned" })
2 @DomainParams( { "U", "L", "D" })
3 @DomainInherits( { "Command<U,L,D>" })
4 public abstract class AbstractCommand implements Command {
5
6 }
```

Figure 3: Annotations automatically added.

2.3 Solve annotation warnings

Next, we used the type checker tool (ArchCheckJ [2, Section 4.3.1]), to check if the annotations are consistent with the code and with each other. For inconsistencies, ArchCheckJ reported more than 1000 warning prioritized based on their type. We selected all warnings of a specific type and solved them while iteratively running ArchCheckJ.

The first type of annotation warnings we solved was related to library types. ArchCheckJ supports associating ownership domain annotations to Java library classes using external XML files. For some of the common classes such as `ArrayList` and `File`, we reused the XML files from the previous case studies. We created new XML files for the library classes that are specific to AFS such as `LoggerFactory` from the slf4j external library, or `java.net.InetAddress`, and `java.security.KeyStore` from the Java standard library.

The next type of annotation warning was related to variables of type `String`. We decided to annotate these variable with the `shared` domain. For example, we changed the formal parameter of `getCommand` from `lent` to `shared`.

We solved 240 annotation warnings by extracting a local variable for `new` expressions that we missed during the preparation of the code. For example, the class `ComandFactoryFactory` used static code to create the default commands supported by the server. Each command is instantiated and added to a `HashMap<String,Command>` object. We refactored the code, such that a local variable exists for each command, and we added the corresponding annotation for each local variable. Since these instantiations are in static code, we decided to annotate each variable with `shared`. Figure 4 shows the changes with the original code commented.

2.3.1 Problems with type casting.

Non-generic code tends to suffer from various problems. For example, inserting an objects of the wrong type into a collection may cause a runtime exception. For example, casting from an `Object` instance to another type may led to runtime exception if the instance cannot be casted. While adding annotations, `Object`type does not have any domain parameters. However, if a variable of a type that has domain parameters is assigned to a variable of type `Object`, the type checker

```
1 @Domains( { "owned" })
2 @DomainParams( { "U", "L", "D" })
3 public class CommandFactoryFactory {
4
5     private static final @Domain("unique <shared,shared>")
6     HashMap<String, Command> DEFAULT_COMMAND_MAP = new HashMap<String, Command>();
7
8     static {
9         // first populate the default command list
10
11        //DEFAULT_COMMAND_MAP.put("ABOR", new ABOR());
12        @Domain("shared") ABOR abor = new ABOR();
13                DEFAULT_COMMAND_MAP.put("ABOR", abor);
14
15        //DEFAULT_COMMAND_MAP.put("ACCT", new ACCT());
16                @Domain("shared") ACCT acct = new ACCT();
17                DEFAULT_COMMAND_MAP.put("ACCT", acct);
```

Figure 4: Local variables extracted in `CommandFactoryFactory` static code.

generates a warning that it cannot bind the domain parameters. For example, consider the code in Figure 5 (the original code is available as comments).

```
1 //public boolean equals(@Domain("lent") Object obj) {
2 // if (obj != null && obj instanceof NativeFtpFile) {
3 // theCanonicalFile = ((NativeFtpFile) obj).file
4 // .getCanonicalFile();
5 public boolean equals(@Domain("lent<U,L,D>") NativeFtpFile obj) {
6         if (obj != null) {
7             theCanonicalFile = obj.file
8                         .getCanonicalFile();
9         ...
10 }
```

Figure 5: Code changes due to imprecise casting.

The `obj` parameter is of type Object and in line 3, is casted to `NativeFtpFile`, which is a type that requires domain parameters. To solve the ownership domain warning, we changed the code and defined the type of `obj` as `NativeFtpFile` instead of `Object`.

We found another imprecise type casting problem in `DataConnectionConfigurationFactoryBean`. In this case, we changed the declared type returned by the `getObject()` method from `Object` to `DataConnectionConfiguration`, the actual type returned by calling

`createDataConnectionConfiguration`. We placed the returned object in `D`.

```
1 @Domains( { "owned" })
2 @DomainParams( { "U", "L", "D" })
3 @DomainInherits( { "DataConnectionConfigurationFactory<U,L,D>" })
4 public class DataConnectionConfigurationFactoryBean
5 extends DataConnectionConfigurationFactory implements FactoryBean {
6
7     //public Object getObject()
8     public @Domain("D<U,L,D>") DataConnectionConfiguration getObject()
9     throws Exception {
10        return createDataConnectionConfiguration();
11    }
12
13    public Class<?> getObjectType() {
14        return DataConnectionConfiguration.class;
15    }
16
17    public boolean isSingleton() {
18        return false;
19    }
20 }
```

Figure 6: Code changes due to imprecise casting in `DataConnectionConfigurationFactoryBean`.

2.3.2 Public methods expose internal representation.

The class `AbstractListener` has two fields, `blockedAddresses` and `blockedSubnets`, both collections of addresses or filters used to block specified connections. Since the collections appears to store sensitive data, we placed them in the private domain `owned`. However, there are two public methods that return a direct alias to each field. We annotated the return type of these methods as `unique` without solving all the annotation warnings. To solve the warnings, a shallow copy of the field should be returned instead of a direct alias. Performing such a change would avoid the representation exposure in `AbstractListener`, but since we did not want to alter the original behavior, we did not go further with the change. Instead, we added a comment to the problematic code (Fig. 7).

```
@Domains( { "owned" })
@DomainParams( { "U", "L", "D" })
@DomainInherits( { "Listener<U,L,D>" })
public abstract class AbstractListener implements Listener{

    private @Domain("owned<shared>") List<InetAddress> blockedAddresses;
    private @Domain("owned<shared>") List<Subnet> blockedSubnets;


    public @Domain("unique<shared>") List<InetAddress> getBlockedAddresses() {
        //TODO: Return a copy of owned
        return blockedAddresses;
    }

    public @Domain("unique<shared>") List<Subnet> getBlockedSubnets() {
        //TODO: Return a copy of owned
        return blockedSubnets;
    }
```

Figure 7: Public methods expose the internal representation of `AbstractListener`.

2.3.3 Missing domain parameters for collection types

ArchDefault did not add the domain parameters for collection data types like `Array`, `List`, `HashTable`, and `Map`. For example, `List<Element>` was a collection of object of type `Element`. We changed the annotation as follows:

```
//@Domain("lent") List<Element> children = SpringUtil.getChildElements(childElm);
@Domain("lent<shared>") List<Element> children = SpringUtil.getChildElements(childElm);
```

To make the annotations consistent with each other, we searched in the code for the invocations of methods that return objects from the public domains of collections, and we changed the annotations of the variable that refers to the returned objects. For example, we searched for the method invocation `iterator()` of `Map<Long, IoSession>` that returns an iterator over the values in the map of type `IoSession`. We changed the annotations of `sessionIterator` which refers to the iterator object by adding the domain parameter:

```
@Domain("lent<shared,L<U,L,D>>")
                Map<Long, IoSession> sessions = session.getService().getManagedSessions();
//@Domain("lent") Iterator<IoSession> sessionIterator = sessions.values().iterator();
@Domain("lent<D<U,D,L>>") Iterator<IoSession> sessionIterator = sessions.values().iterator();
```

2.3.4 Array declarations.

Given an array A of objects of type C we should annotate the array declaration like `@Domain("ARRDOM[OBJDOM]") C[] A`, where `ARRDOM` is the domain of the array object itself and `OBJDOM` is the domain of the objects stored in the array. Here are some other examples:

```
1 //@Domain("lent") File[] matches = ...
2 @Domain("lent[shared]") File[] matches =...
```

3 Extracting and refining the OOG

3.0.5 Selecting the root class.

We used an implementation of an existing analysis (ArchRecJ [2]) to extract the OOG. As an input for ArchRecJ, we specified the root class, i.e., the class where the analysis starts. Usually, in Java applications this is a class that contains the static `main` method. Domains declared in this class are the top-level domains. AFS contains 3 classes with a `main` method: `AddUser`, `CommandLine`, `Daemon`. In addition there are 2 classes in the `examples` folder containing a `main` method that deal with embedding AFS into a Java application. These classes are `EmbeddingFtpServer` and `ManagingUsers`.

As the first attempt, we chose the `EmbeddingFtpServer` class as the root class (Fig. 8). We moved all the code in the `main` method to a new `public` method `init`, and called it from the `main` method. This extract method refactoring was a necessary step for extracting OOG, because the `main` method was a `static` method, that cannot access the domains declared by the class.

In the first extracted OOG, the `UI` domain was empty, because no object in `UI` domain has been instantiated in the `EmbeddingFtpServer` class. Since our goal was to extract a sound OOG, we defined a new class `OOGRootClass` that instantiated the three main classes in AFS identified above, and several factories as indicated in the `examples` folder. The result was an OOG with objects in all three top-level domains.

```
1 @Domains( {"UI","LOGIC","DATA" })
2 public class EmbeddingFtpServer {
3
4     @Domain("LOGIC<UI, LOGIC, DATA>")
5     FtpServerFactory serverFactory = new FtpServerFactory();
6
7     public void init() throws FtpException {
8         ListenerFactory factory = new ListenerFactory();
9         ...
10        @Domain("LOGIC<UI, LOGIC, DATA>")
11        SslConfigurationFactory ssl = new SslConfigurationFactory();
12        ...
13        @Domain("LOGIC<UI, LOGIC, DATA>")
14        PropertiesUserManagerFactory userManagerFactory
15                                     = new PropertiesUserManagerFactory();
16
17        @Domain("shared")
18        File propFile = new File("myusers.properties");
19
20        @Domain("LOGIC<UI, LOGIC, DATA>")
21        FtpServer server = serverFactory.createServer();
22        server.start();
23    }
24    public static void main(String[] args) throws Exception {
25        @Domain("lent") EmbeddingFtpServer server = new EmbeddingFtpServer();
26        server.init();
27    }
28 }
```

Figure 8: EmbeddingFtpServer class as a root class.

3.0.6 Move objects between top-level domains.

We moved several objects between the top-level domains. For example, we moved the object of type `DefaultFtpServerContext` from `DATA` to `LOGIC`. We also moved the object of type `NativeFtpFile` from `LOGIC` to `DATA`. We considered `NativeFtpFile` objects to be in `DATA` since they represent files, the actual data handled by the server. Another object that we moved from `LOGIC` to `DATA` was the object of type `BaseUser`. We decided to do this refinement since `User` is a field of `NativeFtpFile`, i.e., the user appears to be a property of a file.

```
1 @Domains( {"UI","LOGIC","DATA" })
2 public class OOGRootClass {
3   public void init() throws FtpException
4   {
5      @Domain("UI<UI,LOGIC,DATA>")
6       CommandLine cli = new CommandLine();
7
8      @Domain("LOGIC<UI,LOGIC,DATA>")
9      FtpServerFactory serverFactory = new FtpServerFactory();
10
11     @Domain("LOGIC<UI,LOGIC,DATA>")
12     ListenerFactory factory = new ListenerFactory();
13
14     @Domain("LOGIC<UI,LOGIC,DATA>")
15     SslConfigurationFactory ssl = new SslConfigurationFactory();
16
17     @Domain("LOGIC<UI,LOGIC,DATA>")
18     PropertiesUserManagerFactory userManagerFactory
19                                     = new PropertiesUserManagerFactory();
20     @Domain("UI<UI,LOGIC,DATA>")
21     Daemon daemon = new Daemon();
22
23     @Domain("UI<UI,LOGIC,DATA>")
24     AddUser addUser = new AddUser();
25
26        cli.init(args);
27    }
28    public static void main(@Domain("lent[shared]") String[] args) throws Exception {
29      @Domain("shared") OOGRootClass oog = new OOGRootClass();
30      oog.init(args);
31    }
32 }
```

Figure 9: The auxiliary root class for refining the extracted OOG.

3.0.7 Move objects from `shared` to top-level domains.

In the initial OOG, the commands objects were all in the `shared` domain since they were instantiated in static code. Since these objects appeared to be architecturally relevant, we moved them to the `LOGIC` top-level domain by changing the annotations from `@Domain("shared")` to `@Domain("L<U,L,D>)`. Guided by the type checker, we also changed the annotation for the factory object responsible for creating the `Command` objects from `@Domain("L<shared>")` to `@Domain("L<U,L,D>")` (Fig 10).

```
1 @Domains( { "owned" })
2 @DomainParams( { "U", "L", "D" })
3 public class CommandFactoryFactory {
4 ...
5    //@Domain("shared")
6    @Domain("L<U,L,D>")
7         private ABOR abor = new ABOR();
8
9    //@Domain("shared")
10        @Domain("L<U,L,D>")
11        private APPE appe = new APPE();
12
13   //@Domain("shared")
14        @Domain("L<U,L,D>")
15        private ACCT acct = new ACCT();
16 ...
17 }
18
19 @Domains( { "owned","STATISTICS" })
20 @DomainParams( { "U", "L", "D" })
21 @DomainInherits( { "FtpServerContext<U,L,D>" })
22 public class DefaultFtpServerContext implements FtpServerContext {
23
24 //private final @Domain("L<shared>") CommandFactoryFactory
25 // COMMAND_FACTORY_FACTORY = new CommandFactoryFactory();
26 private final @Domain("L<U,L,D>") CommandFactoryFactory
27     COMMAND_FACTORY_FACTORY = new CommandFactoryFactory();
28 ...
29 }
```

Figure 10: Move object of the `Command` type substructure from `shared` to `LOGIC`.

3.0.8 Place objects in new public domains.

Since the command objects were all in `LOGIC`, and there were 50 such objects, we decided to move them into a new public domain named `COMMANDS` of `CommandFactoryFactory` (Fig. 11). Next, we divided the commands in two public domains according to the existing documentation: default commands and site commands[1]. In response to a site command, the server displays statistics about the server state such as the number of file downloaded, and the users that are logged in. As opposed to the default commands required by the standard FTP specification, the site commands are not part of the FTP specification, and not all FTP servers are required to support them. By placing them into a separate domain, the OOG makes visually obvious the fact that AFS supports these

[1] http://mina.apache.org/ftpserver/site-commands.html

commands. Consequently, we created a new public domain `SITE` (Fig. 12). After these refinements, the object of type `CommandFactoryFactory` has three domains, one private and two public. We show a simplified version of its ownership substructure in Figure 13.

```
1 //@Domains( { "owned" })
2 @Domains( { "owned" ,"COMMANDS"})
3 @DomainParams( { "U", "L", "D" })
4 public class CommandFactoryFactory {
5 ...
6    //@Domain("L<U,L,D>")
7    @Domain("COMMANDS<U,L,D>")
8         private ABOR abor = new ABOR();
9
10        //@Domain("L<U,L,D>")
11        @Domain("COMMANDS<U,L,D>")
12        private APPE appe = new APPE();
13
14        //@Domain("L<U,L,D>")
15        @Domain("COMMANDS<U,L,D>")
16        private ACCT acct = new ACCT();
17 ...
18 }
```

Figure 11: Push object of the `Command` type substructure from `LOGIC` to a public domain `COMMANDS`.

```
1 //@Domains( { "owned" ,"COMMANDS"})
2 @Domains( { "owned", "COMMANDS", "SITE"})
3 @DomainParams( { "U", "L", "D" })
4 public class CommandFactoryFactory {
5 ...
6    @Domain("COMMANDS<U,L,D>")
7         private ABOR abor = new ABOR();
8 ...
9     //@Domain("COMMANDS<U,L,D>")
10        @Domain("SITE<U,L,D>")
11        private SITE_WHO siteWHO = new SITE_WHO();
12
13    //@Domain("COMMANDS<U,L,D>")
14        @Domain("SITE<U,L,D>")
15        private SITE_STAT siteSTAT = new SITE_STAT();
16 ...
17 }
```

Figure 12: Create a new public domain `SITE` with objects from the `COMMANDS` public domain.

Another public domain we added was `STREAMS`, to contain objects of a type that extended from

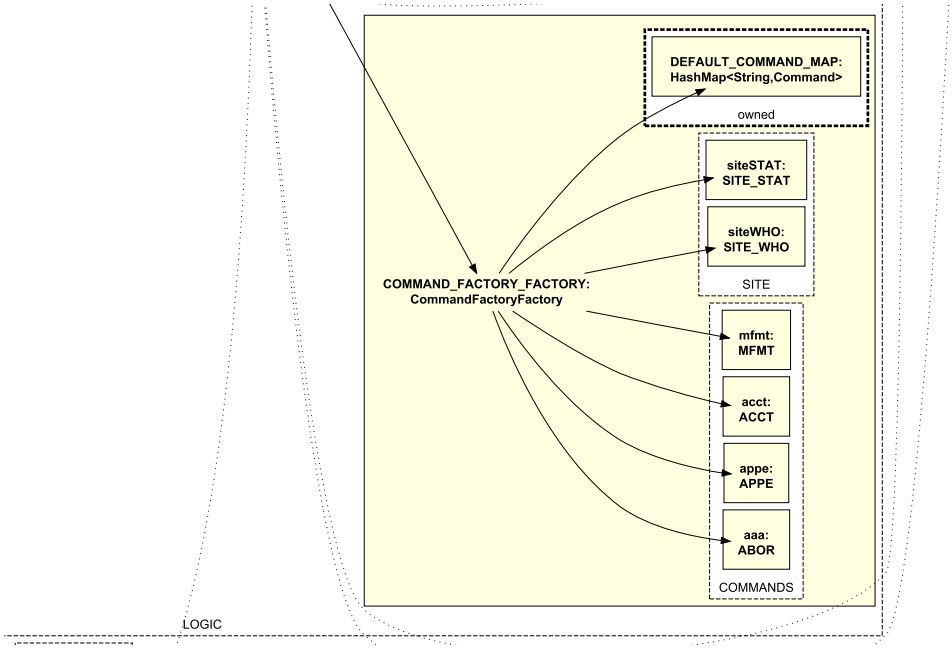


Figure 13: AFS OOG expanded with zoom on the commands.

the `InputStream` and the `OutputStream` interfaces from the Java standard library. We considered such objects to be low-level implementation details, so we pushed them from the top-level domains into lower-level public domains.

While placing the commands into public domains was straightforward, and required changing the annotations in only one file, moving the `InputStream` and the `OutputStream` objects into the `STREAMS` public domain required changing 5 files. Significantly more effort was required to move the object of type `NativeFtpFile` from `LOGIC` to `DATA`, and required changing annotations in 41 files scattered across 5 packages. The difference occurred because the in-degree for `Command` objects was 1. A comment in the code of each `Command` class states that they should not be directly used by applications extending AFS. Instead, applications should use the corresponding factory. On the other hand, `NativeFtpFile` was instantiated 6 times in 2 different files and referenced in all 41 files in which we had to change the annotations.

3.0.9 Annotate exceptions as `unique`.

Since exception objects are passed linearly between objects, we annotated them as `unique`. For example, the objects of type `FtpException`, `DataConnectionException`, `FtpServerConfigurationException` were annotated as `unique`.

3.0.10 Push objects into private domains.

Another refinement we did was to push objects that were implementation details into private domains. We observed that the `LOGIC` domain has an object of type `DirectoryLister`. From the documentation, we found that it is only in a few cases that a `Command` has a `DirectoryLister`. We traced to the code the `DirectoryLister` object and we found that `DirectoryLister` was instantiated as a private field in four classes: `LIST`, `MLSD`, `NLST`, and `STAT`, and each of these classes implemented a command. Each private field was only used internally by the methods; therefore, we placed the referred object in a private domain. For each field, we changed annotation from `@Domain("L")` to `@Domain("owned")`. Consequently, the OOG showed four objects of type `DirectoryLister`, each one in a different private domain (Fig. 14).

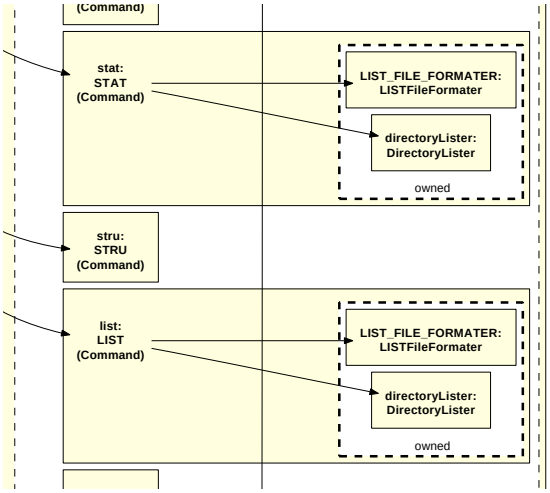


Figure 14: AFS OOG expanded with zoom on the commands with children of type `DirectoryLister`.

3.0.11 Using labeling types.

A developer can specify an optional list of labeling types for decorating to show subtyping information. For example, in Figure 14 the types of the objects `stat`, `stru`, and `list` implement a common interface `Command` shown as a labeling type. This feature is supported by the ArchRecJ tool as an OOG refinement.

3.0.12 Using abstraction by types.

To use abstraction by types, a developer specifies a list of design intent types (DIT). Abstraction by types acts within a domain to collapse objects that share a common base type from DIT into a single representative object. For example, the AFS OOG showed in the domains `COMMANDS` and `SITE` multiple objects that extend the common interface `Command` (Fig. 13). Abstraction by types merged these objects into one object, `command:Command`, in each of the two domains. As a result, the OOG became less cluttered (Fig. 15).

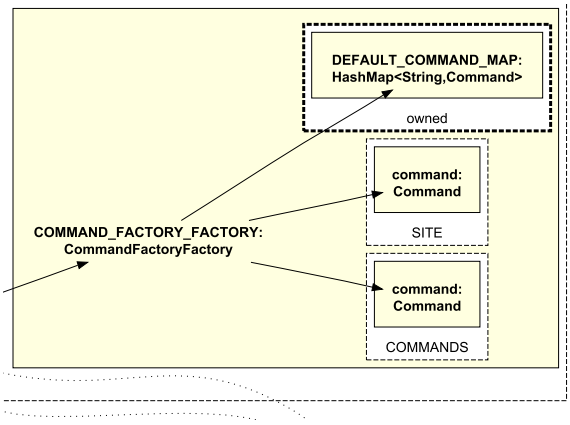


Figure 15: AFS OOG expanded with zoom on the commands after using abstraction by types.

4 Results

In this section, we describe the lessons learned while extracting the AFS OOG. Next, we describe the results of refining the OOG to achieve conformance with the State-Logic-Display Architecture, followed by the results of refining the OOG to achieve conformance with the reference architecture

available in the AFS documentation.

4.1 Extraction of the AFS OOG

The AFS OOG extracted by ArchCheckJ has 48 top-level objects out of a total of 163 objects with 3 top-level domains, 5 public domains and 27 private domains (Fig 16). After extraction, we explored the OOG using the ArchCheckJ features that allowed us to expand and collapse the substructure of objects, and to trace from objects to code. We also used more advanced feature such as abstraction by types, which allowed us to merge objects in the same domain that extend a common type.

4.1.1 Observation 1: Abstraction by types reduced the number of top-level objects.

To reduce the number of top level objects, we used abstraction by types which merges objects in the same domain that have a type that extends a given common type. We specified the list of design intent types from the top-level application types or from library types. The complete list is available in Table 2.

We applied abstraction by types incrementally, adding design intent types, as needed. In the process of using abstraction by types, we noticed that objects were misplaced in some domains. For example, objects of a type that implemented the `AuthorizationRequest` interface were in both top-level domains `LOGIC` and `DATA`. As a result, these objects were not merged when we added `AuthorizationRequest` to the list of deign intent types because the objects were in different domains. So we moved all of these objects into the same domain `DATA`, and the objects were merged in that domain.

After applying abstraction by types, the number of top-level objects was reduced from 48 to 35, thus reducing the cluttering of the OOG. For example, all the objects implementing authorization permission were merged into an object of type `Authority` (Fig. 16, Fig. 17).

Table 2: AFS design intent types.

Qualified type name	Observation
org.apache.mina.core.filterchain.IoFilter	library interface
org.apache.ftpserver.ftplet.Authority	application interface
org.apache.ftpserver.ftplet.AuthorizationRequest	application interface
org.apache.ftpserver.command.impl.listing.FileFilter	application interface
org.apache.ftpserver.ftplet.Authentication	application interface
org.apache.ftpserver.FtpServer	application interface
org.apache.ftpserver.ftplet.Ftplet	application interface
org.apache.ftpserver.ftplet.FtpletContext	application interface
org.apache.ftpserver.ftplet.FtpFile	application interface
org.apache.ftpserver.command.Command	application interface
org.apache.ftpserver.ftplet.FtpReply	application interface
java.io.InputStream	library class
java.io.OutputStream	library class
org.apache.ftpserver.ftplet.FtpSession	application interface
org.apache.ftpserver.impl.FtpIoSession	application class
org.apache.ftpserver.ftplet.UserManager	application interface
org.apache.ftpserver.ftplet.FileSystemView	application interface
org.apache.ftpserver.ftplet.User	application interface

4.1.2 Observation 2: We reused the list of design intent types as labeling types.

If two objects share a common base type, an OOG can show this subtyping information as labeling types, i.e., both objects are decorated with the common type. In general, labeling types are different from design intent types, but for AFS, we decided to use the same list. This decision, allowed us to investigate if we can express all the top-level objects as instances of the key interfaces in the system.

4.1.3 Observation 3: After refinement, we managed to express most of the top-level objects as instances of the top-level application interfaces.

At the code level, the top-level application interfaces were scattered into multiple packages. ArchRecJ helped us to select the top-level application interfaces by displaying a class diagram with inheritance relationships. Then, we tried to make all the instances of those interfaces as the most architecturally relevant objects. At the same time, we pushed objects that do not have a type that correspond to those interfaces underneath more architecturally relevant objects. We

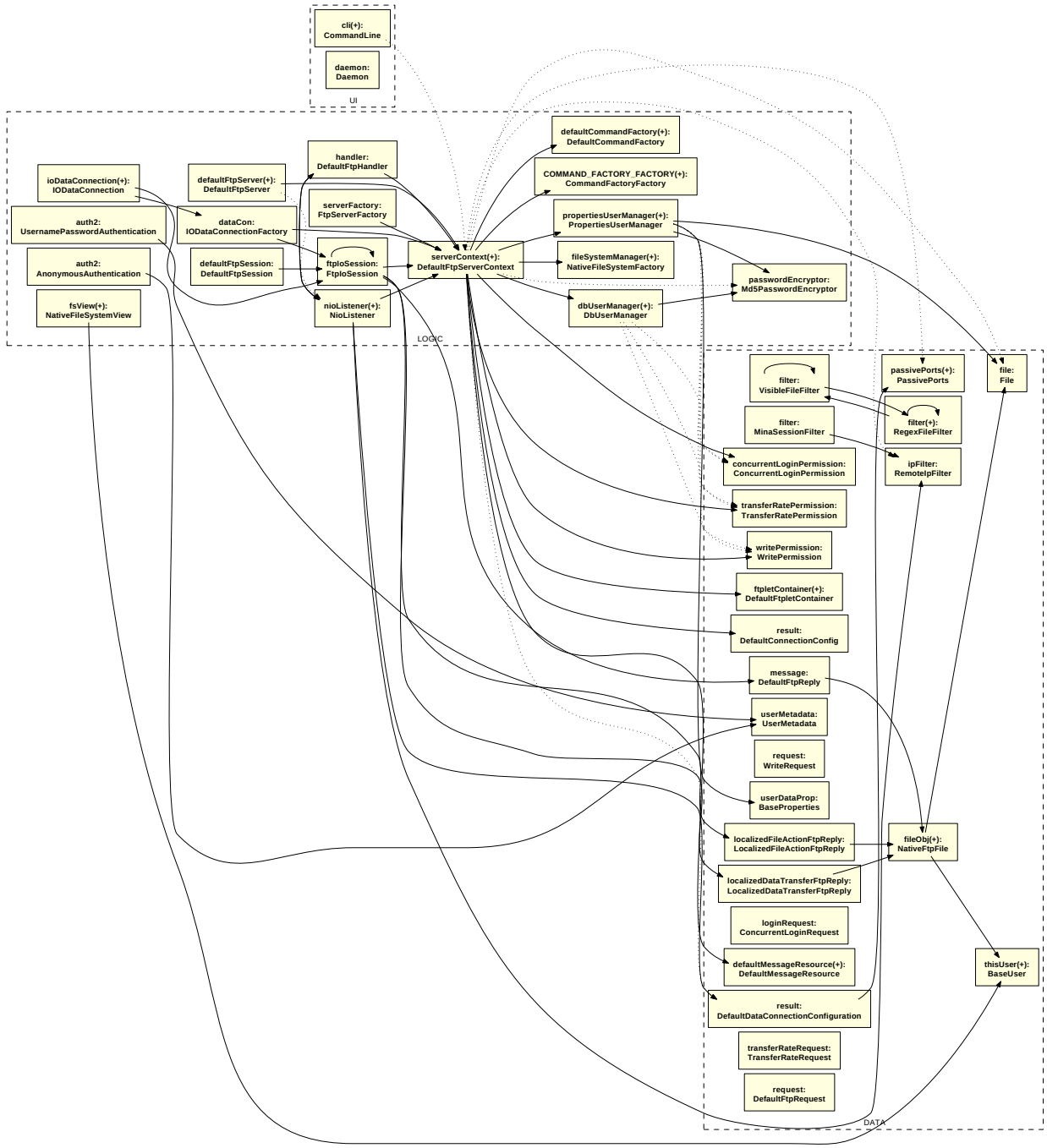


Figure 16: Top-level objects and domains in the AFS OOG, with no edges between `UI` and `DATA`.

used labeling types to decorate the objects with the interfaces. After the refinements, most of the top-level objects are labeled with a top-level application interface.

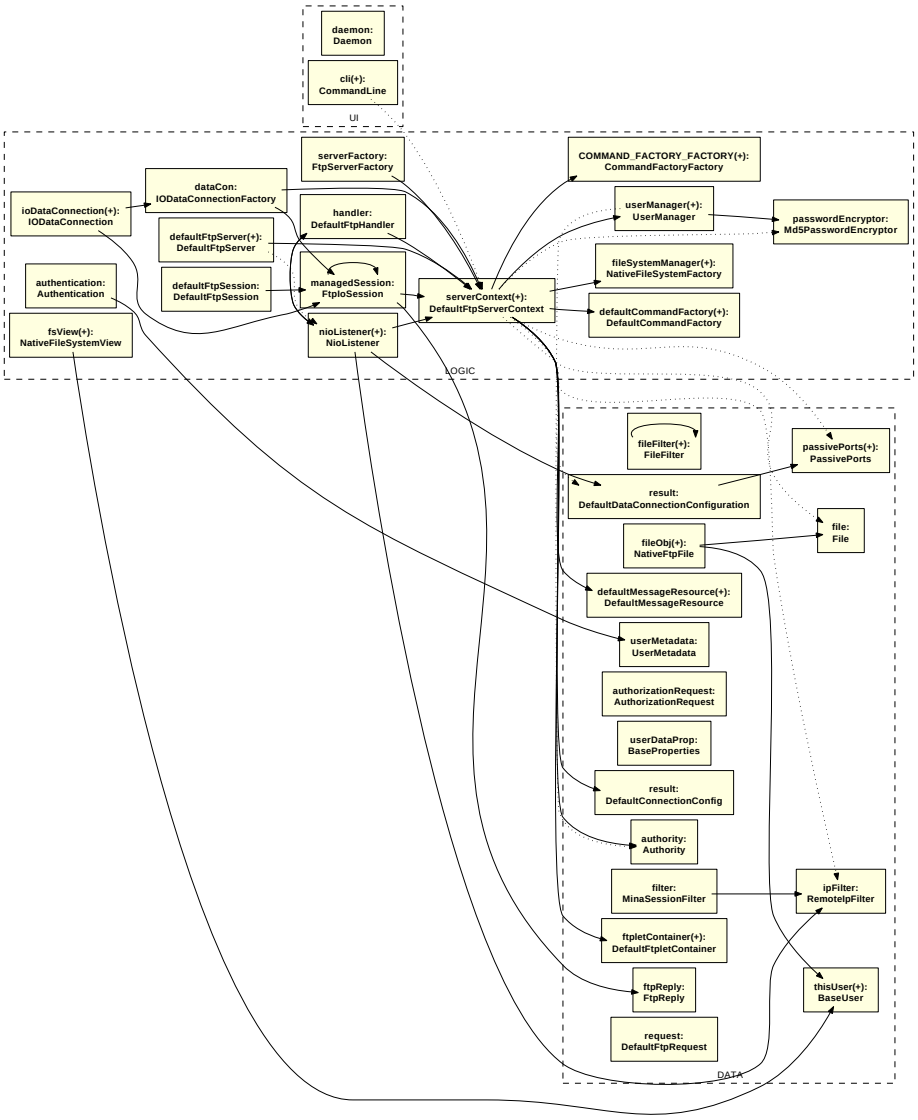


Figure 17: AFS-OOG after using abstraction by types.

4.1.4 Observation 4: AFS developers followed good object-oriented design principles.

The fact that we were able to express the top-level objects in terms of top-level application interfaces, implies that the AFS developers followed the design principle: "design to an interface, not to an implementation" [9] which makes the AFS more flexible. We were not able to obtain a similar result for systems where developers did not follow a similar design principle [3].

It was common that AFS interfaces had a default implementation. For example, each of the interfaces `FtpSession`, `FtpRequest` has only one corresponding concrete class: `DefaultFtpSession`,

and `DefaultFtpRequest`, respectively. Other interfaces such as `FtpReply` are the root of a much richer inheritance hierarchy. The default implementation was still available as `DefaultFtpReply`, but at the same time, the inheritance hierarchy has other interfaces and concrete classes corresponding to specific server replies such as: the result of renaming a file `RenameFtpReply`, or the result of data transferred `DataTransferFtpReply`.

4.1.5 Observation 5: Fewer objects than expected were merged using abstraction by types.

Other architectural extraction techniques use type clustering based on naming convention. Abstraction by types relies only on the subtyping relation and ignores naming convention. One can be mistaken in thinking that developers were using a naming convention to refer to classes in the same inheritance hierarchy. For example, in the case of authorization permission objects, the naming convention indicated that the objects of types `ConcurrentLoginPermission`, `TransferRatePermission`, and `WritePermission` were to be merged using abstraction by types. However, this was not always the case. ArchRecJ helped us to select the design intent types by displaying a class diagram with inheritance relationships. For example, we found that the only common type extended by `AuthorizationRequest` and `DefaultFtpRequest` was `Object`, although based on their names, we assumed that they both implement the `FtpRequest` interface. Other similar examples of misleading names were the ones ending with "`Filter`" such as `RemoteIpFilter`, `MinaSessionFilter` and `FileFilter`, or with "`File`" such as `NativeFtpFile` and `File`.

4.2 Horizontal Conformance

To answer RQ1, we analyze if the OOG achieves horizontal conformance with the State-Logic-Display architectural pattern, where `DATA` maps to the State tier, `LOGIC` maps to the Logic tier and `UI` maps to the Display tier.

4.2.1 Observation 6: The refined OOG conforms to a State-Logic-Display architecture.

In a State-Logic-Display architecture, no edge should exist from the State tier to the Display tier and vice-versa [11] (Fig. 18). Similarly, in the OOG, no points-to edge should exist between an object in `UI` and an object in `DATA`. We refined the OOG to satisfy these constraints imposed by the pattern.

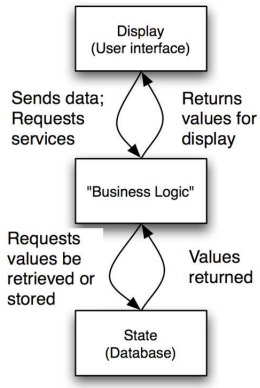


Figure 18: The State-Logic-Display architectural pattern. Source: [11, Fig. 4-3].

After the refinements, the points-to edges in the OOG are from objects in `UI` to objects in `LOGIC`, and from objects in `LOGIC` to objects in `DATA`. We were able to move the objects between the top-level domains in such a way that the OOG did not show edges between `UI` and `DATA`.

4.3 Vertical Conformance

To answer RQ2, we analyzed if the OOG achieves vertical conformance with a reference architecture available in the documentation. Through vertical conformance, we checked that the OOG and the reference architecture have a similar number of tiers, a similar number of objects in each tier, and a similar hierarchical decomposition.

The Response for Comments (RFC) document is the official specification of a FTP server and it contains a diagram representing an FTP model. The model has two types of boxes with rounded corners and square corners. Since the diagram does not have a legend, we interpreted the square corners as tiers and the rounded corners as components.

For example, the protocol interpreter (PI) or data transfer protocol (DTP) are both components in the Server-FTP and USER-FTP tiers. The edges represent data or control communication paths. For example, the edge between User PI and Server PI represents a communication path used to exchange FTP Commands and FTP Replies. The edge between User DTP and Server DTP represents a communication path used to exchange Data: a part of a file, an entire file, or several files.

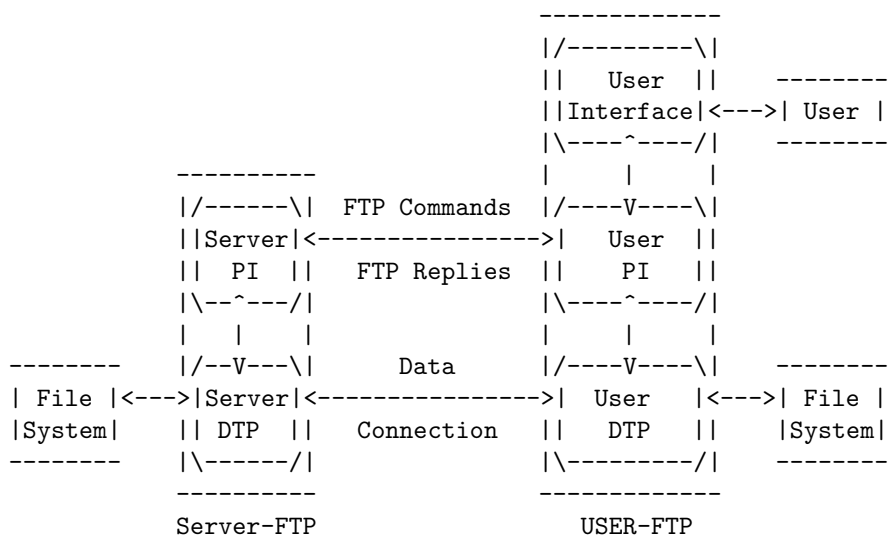


Figure 19: The reference architecture of an FTP Server, as drawn in RFC 959 [10].

Table 3: Relating the reference architecture of an FTP server to the AFS OOG.

Reference Component	OOG Object	JavaDoc comments supporting the mapping
Server PI (Protocol Interpreter)	Ftplet	A Ftplet is a small Java program that runs within an FTP server. Ftplet objects receive and respond to requests from FTP clients.
Server DTP (Data Transfer Protocol)	FtpletContext	A Ftplet configuration object used by a Ftplet container used to pass information to a Ftplet during initialization. The configuration information contains initialization parameters.
Server File System	FtpFile	This is the file abstraction used by the server
FTP Commands	Command	This interface encapsulates all the FTP commands
FTP Replies	FtpReply	Interface for a reply to an FTP request
Data	InputStream	data to the client (e.g. RETR).
Data	OutputStream	data from the client (e.g. STOR).
Connection	DataConnection	Transfer data
User DTP (Data Transfer Protocol)	FtpSession	Defines an client session with the FTP server. The session is born when the client connects and dies when the client disconnects. Ftplet methods will always get the same session for one user and one connection.
User PI (Protocol Interpreter)	FtpIoSession	Contains user name between USER and PASS commands
User Interface	UserManager	User manager interface
User File System	FileSystemView	This is an abstraction over the user file system view
User	User	Basic user interface

4.3.1 Observation 7: The refined OOG conforms to the reference runtime architecture.

We performed further refinements on the annotations until the OOG conformed to the reference architecture. The reference architecture corresponded to a client server architecture. We placed all the objects on the server side in the `SERVERFTP` domain and all the objects on the client side in the `USERFTP` domain. We performed the following refinement steps:

- We renamed the top-level domains `LOGIC` and `DATA` to `SERVERFTP` and `USERFTP`, respectively. Since the `UI` was not mentioned in the reference architecture, we left it unchanged.
- We moved some top-level objects from one top-level domain to another (Table 4).
- We moved some objects in lower-level domains to top-level domains, e.g., we moved the commands objects from `COMMANDS` and `SITE` to `SERVERFTP`, and `inputStream` and `outputStream` from `STREAMS` to `USERFTP`. These changes undid some of our previous refinements since those were not guided by the reference architecture.

- We pushed some top-level objects underneath other objects, e.g., we pushed instances of type `ConnectionConfig`, `MessageResource` underneath instances of `DefaultFtpServerContext`.

Table 4: Top-level objects moved between top-level domains.

Object	Source Domain	Destination Domain
`parentObject:NativeFtpFile`	`USERFTP`	`SERVERFTP`
`ftpletContainer:DefaultFtpletContainer`	`USERFTP`	`SERVERFTP`
`userManager:UserManager`	`SERVERFTP`	`USERFTP`
`fsView:NativeFileSystemView`	`SERVERFTP`	`USERFTP`
`defaultFtpSession:DefaultFtpSession`	`SERVERFTP`	`USERFTP`
`managedSession:FtpIoSession`	`SERVERFTP`	`USERFTP`

We recognized all the objects from the OOG as components in the reference architecture (Figure 19) based on their labels (e.g., `User`, `DataConnection`) and we established the mapping in Table 3. When the mapping was not straightforward, we traced the top-level object to the code lines and read the description of their corresponding types available as JavaDoc comments. If we were able to establish a mapping, we marked the object in the OOG as a convergence (✓); otherwise we marked it as a divergence (✚). We were able to establish a mapping for all the components in the reference architecture (Fig. 20). The OOG has six additional objects that do not correspond to any component in the reference architecture. These divergences represent additional implementation details that were not considered in the reference architecture, such as the user interface of the server, the factory object used to create data connections, or permission requests to change or move files. Knowing about the additional information is likely to be useful to developers using AFS. For example, the OOG shows the object `defaultSslConfiguration`. Developers might learn from the presence of this object that the Apache implementation supports Secure Socket Layer (SSL) data transfer and they should instantiate the corresponding class while embedding the server in their application.

The fact that we located all the components in the reference architecture and we also found a relative small number of divergences, indicated that in general the AFS developers followed the reference architecture. At the same time, the implementation contains additional details specific to the technology used. For example, `InputStream` and `OutputStream` are specific to a Java implementation, while the objects in the `UI` domain might not be used when the FTP server is a

part of a larger application. Overall, we were able to map top-level domains to the tiers in the reference architecture, and both representations have a similar number of components per tier. The reference architecture showed only the high-level decomposition which is similar to the one in the collapsed OOG. However, one advantage of the OOG is that it has additional details that can be viewed by expanding the ownership substructure of the top-level objects. Such details are not available in the reference architecture.

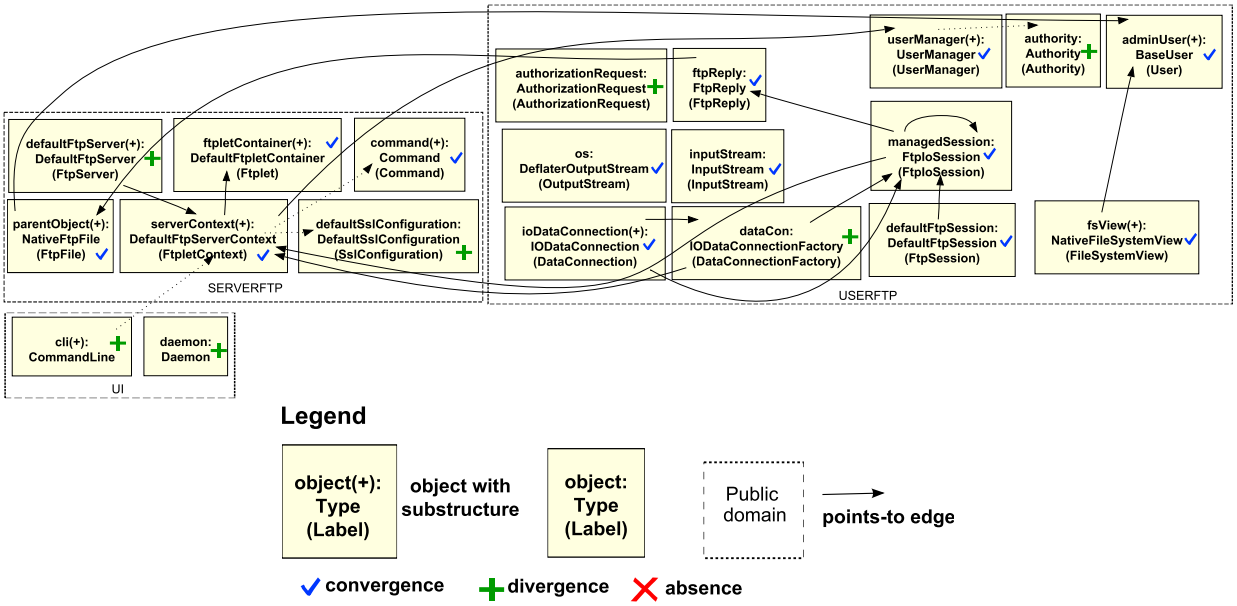


Figure 20: AFS OOG with labeling types.

4.3.2 Observation 8: The OOG shows objects used to incorporate AFS in the Spring framework.

AFS can be used as a stand alone application, or it can also be a plug-in to the Spring framework. For Spring integration, AFS developers defined additional classes that extend the interface `FactoryBean` from the Spring framework. These classes are instantiated using Java reflection based on XML configuration files used by the Spring framework. Since our static analysis does not understand these XML configuration files, we have to manually summarize them. To include these objects in the OOG, we manually modified the root class so that it instantiates the AFS-specific classes that extend from `FactoryBean` (Fig. 21). These objects, or their children in the ownership

hierarchy refer to the `serverContext` and `defaultSslConfiguration` objects from `SERVERFTP` domain. They do not refer to objects in `USERFTP` or in `UI` (Fig. 22).

```
1 @Domains( {"UI","SERVERFTP","USERFTP","SPRING" })
2 public class OOGRootClass {
3
4  public void init(@Domain("lent[shared]") String[] args) throws FtpException {
5  //instances specific to Spring framework
6  @Domain("SPRING<SPRING,SERVERFTP,USERFTP>")
7  FtpServerNamespaceHandler springNamespace = new FtpServerNamespaceHandler();
8  @Domain("SPRING<SPRING,SERVERFTP,USERFTP>")
9  FtpServerFactoryBean ftpServerFactoryBean = new FtpServerFactoryBean();
10 @Domain("SPRING<SPRING,SERVERFTP,USERFTP>")
11 ListenerFactoryBean listenerFactoryBean = new ListenerFactoryBean();
12 @Domain("SPRING<SPRING,SERVERFTP,USERFTP>")
13 SslConfigurationFactoryBean sslConfigurationFactoryBean = ...;
14 @Domain("SPRING<SPRING,SERVERFTP,USERFTP>")
15 DataConnectionConfigurationFactoryBean dataConnectionConfigurationFactoryBean = ...;
16 @Domain("SPRING<SPRING,SERVERFTP,USERFTP>")
17 ConnectionConfigFactoryBean connectionConfigFactoryBean = ...;
18
19  //instances specific to a standalone application
20  @Domain("UI<UI,SERVERFTP,USERFTP>") CommandLine cli = new CommandLine();
21  @Domain("UI<UI,SERVERFTP,USERFTP>") Daemon daemon = new Daemon();
22  @Domain("UI<UI,SERVERFTP,USERFTP>") AddUser addUser = new AddUser();
23 }
24 public static void main(@Domain("lent[shared]") String[] args) throws Exception {
25  @Domain("shared") OOGRootClass oog = new OOGRootClass();
26  oog.init(args);
27 }
28 }
```

Figure 21: AFS root class that instantiates Spring objects.

5 Discussion

Overall, the results provided explicit positive assurance for the conformance of the OOG with the State-Logic-Display architectural pattern. There are no points-to edges between objects in the `UI` domain or `SPRING` domain and the `USERFTP` domain. Regarding the conformance of the OOG with the reference architecture, we observed that each component in the reference architecture had a corresponding object in the OOG. The six divergences and the objects in the `UI` and `SPRING` domains can be viewed as implementation details specific to AFS. Before making refinements to

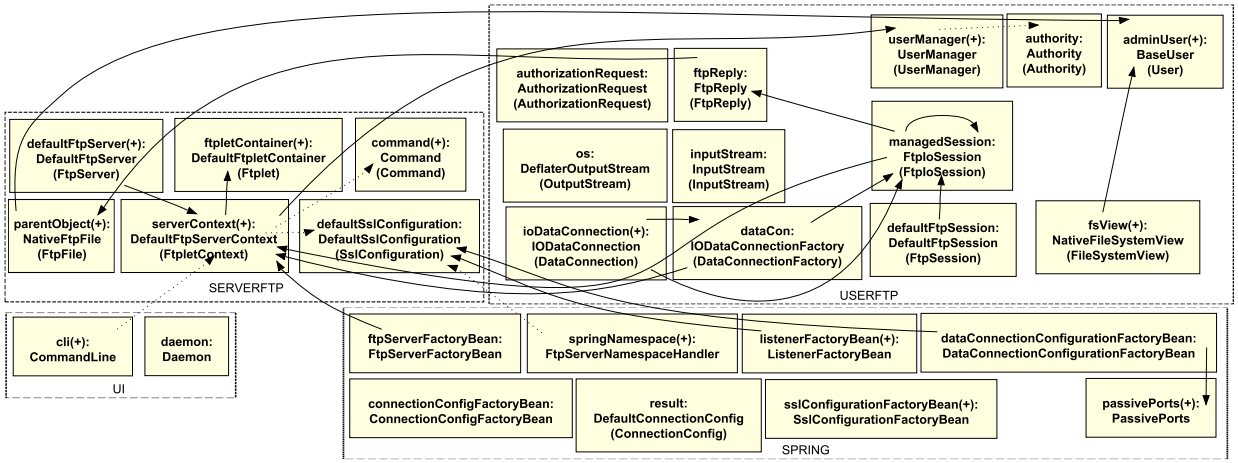


Figure 22: AFS OOG with Spring objects.

analyze vertical conformance, there were three component absences: `Command`, `InputStream` and `OutputStream` and a higher number of divergences. To reduce the number of absences, we reverted the refinement of pushing commands to lower level public domains. To reduce the number of divergences, we pushed top-level objects in lower-level domains. For example, there were more factory objects in the top-level domains increasing the number of divergences. For example, objects of type `DefaultCommandFactory`, `NativeFileSystemFactory`, and `Md5PasswordEncryptor` were pushed to lower-level domains.

5.0.3 Limitation 1: To analyze conformance integrity, we need to enhance the OOG with communication edges.

When a target architecture exists an OOG can be used to analyze the communication integrity between the code and the intended architecture. By definition, communication integrity means that each component in the implementation may only communicate directly with the components to which it is connected in the architecture [5]. So far, we are able to establish a mapping between the objects in the OOG and the components in the reference architecture. We also need to analyze the conformance of the edges. The current OOG edges represent points-to relations between objects, and we cannot establish a correspondence to the edges in the reference architecture, which were communication edges. We plan to enhance the OOG with dataflow communication edges [12].

5.0.4 Limitation 2: Some objects in the OOG correspond to edges in the reference architecture, rather than components.

We observed that there are some edges in the reference architecture that corresponded to objects in the OOG (e.g., `Command`, `FtpReply`, `InputStream`, `OutputStream`, `DataConnection`). Ideally, we want to show these objects as edges in the OOG as well. Such an edge would indicate that there are objects flowing between different domains. But our approach assumes a single ownership model, where an object is assigned to one domain, and the domain does not change.

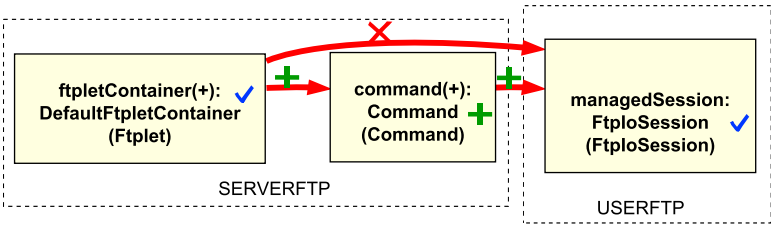
In such cases, it may be useful to abstract an object in the OOG as communication summary edge. Consider for example, the OOG with dataflow communication edges that shows that the `Ftplet` object communicates with the `Command` object, which in turn communicates with the `FtpIoSession` object (Fig. 23a). This transitive communication is summarized in the reference architecture as a direct edge between Server PI and User PI labeled as Commands, where Server PI maps to `Ftplet`, and User PI maps to `FtpIoSession`. If we allow a similar summarization in the OOG, it would show the `Command` object as a summary communication edge between `Ftplet` and `FtpIoSession`, and the conformance analysis would mark this edge as a convergence (Fig. 23b). Otherwise, the conformance analysis marks as divergences the dataflow communication edges to and from the `Command` object, and as an absence the communication between `Ftplet` and `FtpIoSession` (Fig. 23a).

5.0.5 Threats to validity.

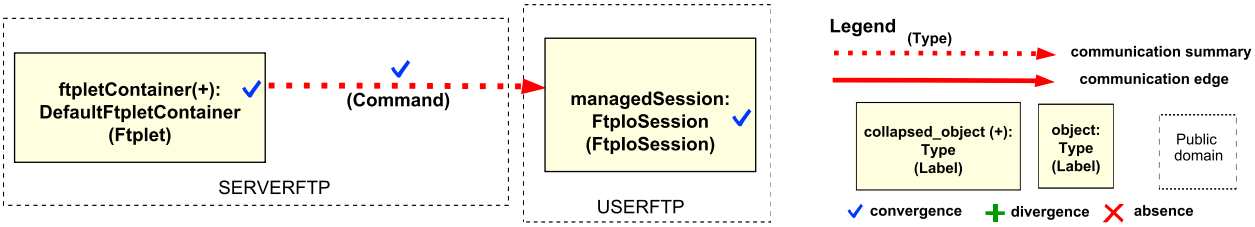
One threat to validity is that 228 annotation warnings remain unsolved; therefore, we cannot conclude that the OOG is sound. Further work is needed to address these warnings.

6 Conclusion

In this report, we described the process of annotating the legacy code of AFS using ownership annotations. Based on the annotations, we extracted an initial OOG that we then iteratively refined. We analyzed the horizontal conformance between the OOG and the State-Logic-Display



(a) partial OOG with Command object



(b) summarized edge

architectural pattern. We then analyzed the vertical conformance between the OOG and the reference runtime architecture available in the official documentation of AFS, and we described the challenges we encountered. Overall, the results provided explicit positive assurance that the AFS implementation conforms with the State-Logic-Display architectural pattern and with the reference architecture.

Acknowledgements

The authors thank Anwar Mohammadi and Zeyad Hailat for their help with adding initial annotations to the AFS system.

References

[1] Apache FtpServer. `http://mina.apache.org/ftpserver/`.

[2] M. Abi-Antoun. *Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure*. PhD thesis, Carnegie Mellon University, 2010. Available as Technical Report CMU-ISR-10-114.

[3] M. Abi-Antoun and Z. Hailat. A Case Study in Extracting the Runtime Architecture of an Object-Oriented System. Technical report, Wayne State University, 2011.

[4] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object-Oriented Programming (ECOOP)*, 2004.

[5] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *ICSE*, 2002.

[6] Apache Maven Project. `http://maven.apache.org/`, 2011.

[7] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *TSE*, 35(4), 2009.

[8] Eclipse Metrics Plugin. `http://metrics.sourceforge.net/`, 2010.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] J. Postel and J. Reynolds. File Transfer Protocol (FTP). (RFC 959), 1985. http://mina.apache.org/ftpserver/rfc959.html.

[11] R. N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[12] R. Vanciu and M. Abi-Antoun. Extracting Dataflow Communication from Object-Oriented Code. Technical report, WSU, 2011. `www.cs.wayne.edu/~mabianto/tech_reports/VA11_TR.pdf`.