

Model-Driven Code Generation for Information Flow Secure Systems with IFlow

Kuzman Katkalov, Peter Fischer, Kurt Stenzel, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Katkalov, Kuzman, Peter Fischer, Kurt Stenzel, and Wolfgang Reif. 2012.
“Model-Driven Code Generation for Information Flow Secure Systems with IFlow.”
Augsburg: Universität Augsburg.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under the following conditions:

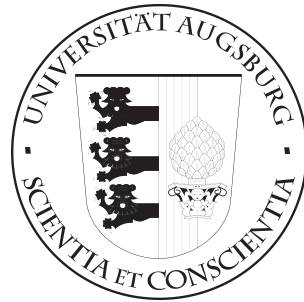
Deutsches Urheberrecht

Weitere Informationen finden Sie unter: / For more information see:

<https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publizieren>



UNIVERSITÄT AUGSBURG



**Model-Driven Code Generation for
Information Flow Secure Systems with
IFlow**

**Kuzman Katkalov, Peter Fischer, Kurt Stenzel
and Wolfgang Reif**

Report 2012-04

March 2012



INSTITUT FÜR INFORMATIK
D-86135 AUGSBURG

Copyright © Kuzman Katkalov, Peter Fischer, Kurt Stenzel and Wolfgang Reif
Institut für Informatik
Universität Augsburg
D-86135 Augsburg, Germany
<http://www.Informatik.Uni-Augsburg.DE>
— all rights reserved —

Abstract

As personal information moves from home computers to mobile devices, protection against information leaks and data theft becomes an increasingly important and current issue. We develop a model-driven approach called IFlow which allows a developer to model mobile applications with complex information flow properties using UML. Using model-to-model and model-to-code transformations we generate code skeletons for those applications and verify noninterference properties using a language-based approach. Further, we will use those properties as lemmas for a formal verification of an automatically generated formal representation of the modeled application. In this report, we focus on automatic code generation, evaluation of language-based information flow control solutions and deployment of generated code to target platforms.

1 Introduction

Smartphones, specifically third party smartphone applications constantly aggregate personal user information like location, private photos or payment data. As smartphones grow more popular and come to replace the personal computer for many everyday tasks, information leaks become a major concern in the mobile setting.

Protection of private data against leaks or misuse is becoming one of the primary research targets in the industry and the academia. Enforcement of such protection provided by mobile operation systems like Android is very limited and does not guarantee safety of personal data. Combined with the variety of unverified third-party applications on the official Android Marketplace, which is the fastest growing software store to date [2] and a smartphone's ability to access the internet, this poses a serious threat to the privacy of the user and his personal or corporate data.

Attacks and vulnerabilities which lead to personal data exfiltration or privilege escalation have already been shown in numerous research papers and also discovered in the wild. Those can lead to unauthorized exposure of personal data to advertisement networks, eavesdropping on phone calls or text messages and data theft [8], and are usually utilized by malware. A selected example for such malware is *DroidDreamLight*, which is actively used to trojanize legitimate applications to then be reuploaded to the Android Marketplace [3]. However, legitimate applications too can leak private data [8], albeit some might do so by accident.

The IFlow project¹ tackles the challenge of providing a development approach for designing applications, focusing on mobile Android apps, while giving formal guarantees about application-specific information flow (IF) properties. We believe that only by considering such security aspects early on during the design phase one can achieve a satisfactory level of protection against data leakage. Thus, we protect the developer against introducing unintended information leaks, while also protecting the user of an application developed using the IFlow approach against malicious developers.

This report focuses on the specifics of IF-secure application modeling and source code generation from a practical point of view. The theory behind infor-

¹This work is sponsored by the Priority Programme 1496 "Reliably Secure Software Systems - RS³" of the Deutsche Forschungsgemeinschaft (DFG).

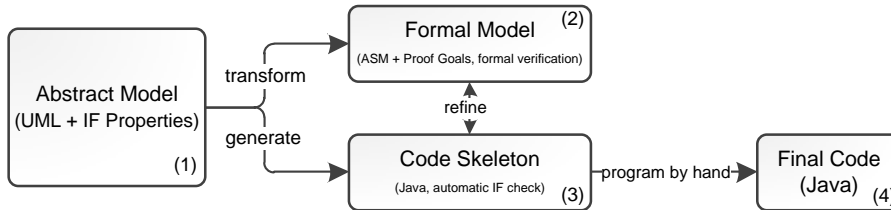


Figure 1: IFlow model-driven approach

mation flow control as well as the description of the formal model and verification will be given in another report. Section 2 gives an overview over the IFlow approach, section 3 presents the IFlow modeling guidelines using our case study as an example, while section 4 explains the automatic code generation utilizing model-to-model and model-to-text transformations. Section 5 concludes this report with a summary of achieved results and an outlook.

2 IFlow Approach

IFlow is a model-driven approach to develop secure applications with regard to information flow as illustrated in Figure 1. It starts with an abstract UML model of an application (1), which can include several applications communicating with each other or a number of web services. This model utilizes a subset of standard UML diagrams as well as IFlow UML profiles and is expanded with information flow property annotations. Such annotations are used to express policies for confidentiality of information by assigning security levels to information sources or sinks. The modeling of actual functionality of applications and services is kept abstract, allowing the developer to focus on high-level security properties and agent communication.

The platform-independent IFlow UML model then undergoes several model-to-model and model-to-text transformations to produce a formal model (2), readable by the interactive verifier KIV [4], as well as an intermediate and a platform specific model (PSM) for Java. This PSM is then used to generate a Java code skeleton (3), which can be checked for illegal information flows by Jif [17] or Joana [12]. The code skeleton can be completed manually by the developer in order to realize actual internal functionality of IFlow agents, checked again with Joana for newly introduced information flows and then deployed on an Android phone or a Java EE web service (4).

Since some application-specific information flow properties cannot be guaranteed using Joana, IFlow integrates the interactive theorem prover KIV in order to prove such properties which extend beyond simple non-interference. This is achieved by employing automatic model-to-text transformations to generate a formal model using Abstract State Machines (ASM) representing the abstract model of the IFlow application. After lifting non-interference properties checked with Joana at the level of Java code to the formal model, the modeler is then able to use them in order to prove additional properties with KIV. This is possible because of the 1:1 IF-preserving refinement relation between the formal model and the skeleton code.

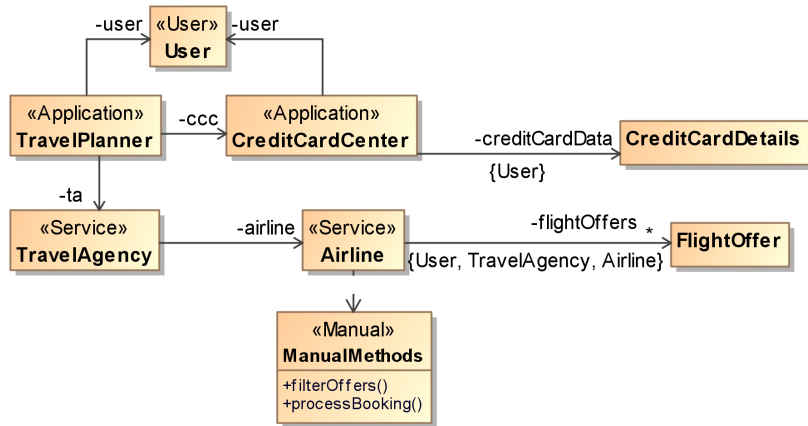


Figure 2: Agent classes

3 Modeling IF-Secure Applications

3.1 Travel Planner Case Study

We explain the IFlow approach with the case study *Travel Planner*, which is a travel booking system consisting of a *travel agency* service (TA) providing flight offers to the user of a mobile *travel planner* (TP) application, developed by the TA. The user is able to select a favored flight offer from a list of offers received from the TA and pay for the flight ticket directly at the *airline* service using the credit card data stored inside a *credit card center* application on his mobile device. The TA then receives a commission from the airline. Secure information flow within this system has to be ensured to provide the user with the guarantee that his credit card data is only ever received by the intended airline, and only after his explicit confirmation.

3.2 Static View

IFlow uses UML to model both static and dynamic views of an application, which can consist of several application agents communicating with each other. We use a class diagram to model smartphone applications, web services or user interfaces, representing each of those agents with a UML class marked with a **Application**-, **Service**- or **User**-stereotype from the predefined IFlow UML profile. Agent class attributes denote data storage of the agent, with their data types also modeled as classes within the class diagram. A **Manual**-stereotyped class contains the signatures for all manual methods (i.e. methods that can later be implemented manually by the developer in order to realize certain application functionality) to be used in sequence diagrams for this application. Figure 2 shows an excerpt from the class diagram for the *TravelPlanner* case study, picturing all involved agents².

²The full model can be found in Appendix A and on our website, <http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/iflow/>

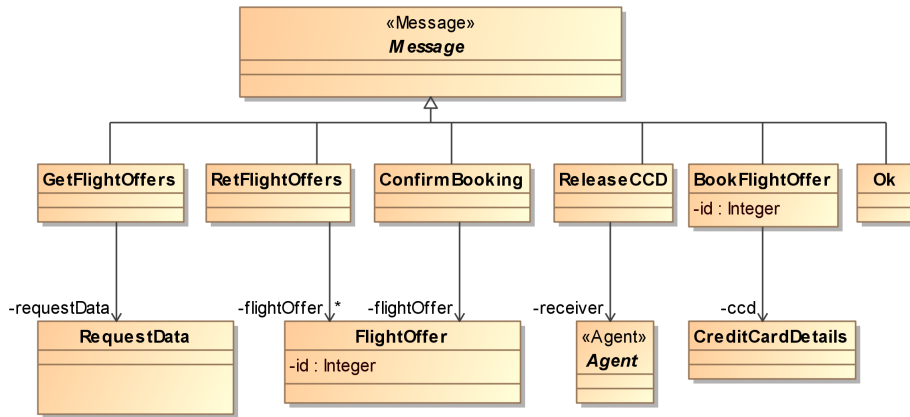


Figure 3: (Excerpt of) message classes

The IF annotations are modeled as UML constraints on class attributes to define their security level and establish a noninterference property (or several). An annotation consists of a list of IFlow agents being able to observe the annotated attribute. Figure 2 shows the *User* annotation applied to the `creditCardData`-attribute of class `CreditCardCenter`, implying that only the *User* agent is able to read it, while the attribute `flightOffers` of agent *Airline* is annotated with the list *User, TravelAgency, Airline*, making flight offers essentially public to all agents. Such annotations imply a lattice, with information being able to flow only to equally or more restrictively annotated model elements. Agent communication takes place by agents exchanging instances of special message classes, defined by the modeler inside a class diagram by inheriting from an abstract class with the IFlow `Message`-stereotype (as seen in Figure 3 and Figure 12) or predefined in an IFlow UML module (see Figure 14).

3.3 Dynamic View

Agent communication is modeled with sequence diagrams, with lifelines representing communicating agents. IFlow sequence diagrams utilize a modified version of the Model Extended Language (MEL)³ to express message instantiations, method calls or local variable definitions. In order to fix the MEL syntax and the subset of UML usable in IFlow models we defined a metamodel, which is also used to instantiate intermediate IFlow models in the automatic model transformations.

Sequence messages carry the name of appropriate message classes followed by a list of parenthesized lifeline class attributes or local variables, which are used to instantiate the message. The order of those variables is determined by the order of attributes defined in the message class. Since non-primitive attributes of a class are denoted with a named association, and UML class associations are not ordered, we define the following guideline: if a class has exactly one association,

³The MEL language was initially designed for the SecureMDD project [15] in order to fully express the functionality of an interactive agent in a UML activity diagram

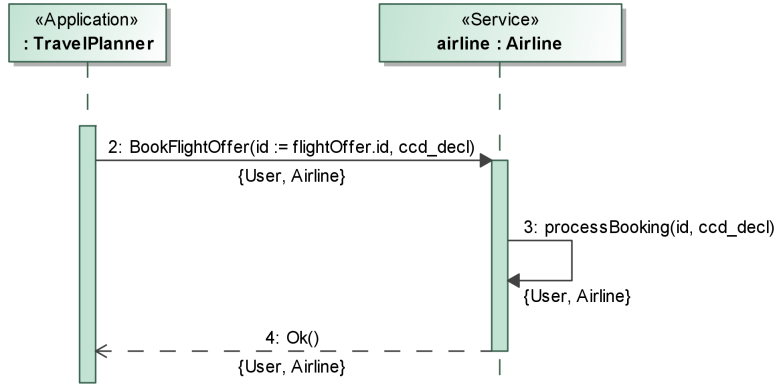


Figure 4: Booking a flight with an airline

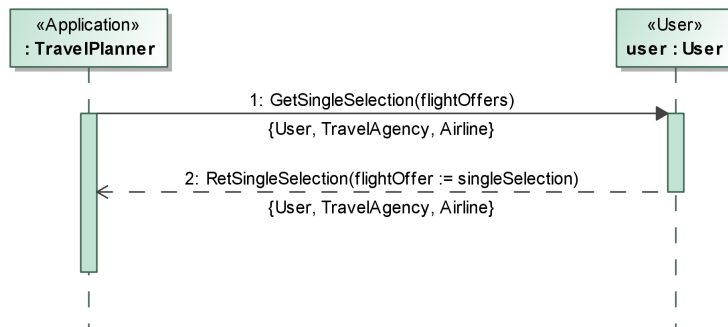


Figure 5: Asking the user to choose an offer from a list

it is to be instantiated last. If a class has more than one association, the modeler needs to define a class constructor with an ordered list of parameters carrying the same name and type as instantiable class attributes.

The receiving lifeline is then able to access those variables or attributes. Each message must be modeled to receive an answer message, while communication with a user is executed by sending a predefined user message to the user lifeline. Messages received by the user lifeline are meant to represent a dialog- or confirmation messages on the user interface. In order to increase the readability and reusability of certain communication fragments, it is also possible to reference other sequence diagrams by utilizing the UML **Interaction Use** element pointing to the appropriate diagram. Each application task, modeled as the communication between agents must begin with a IFlow-predefined user message from a user lifeline to an application. Manual methods are modeled as UML **Self Messages**; their return value can be assigned to an implicitly declared variable and then be reused as input for other manual methods or as part of a message to another agent.

Figure 4 shows an excerpt from the communication between the TP and the airline, booking a flight by providing the airline with the ID of the flight offer and the user's credit card data. The assignment `id := flightOffer.id` implies that the ID of the offer is being extracted from an attribute of a previously selected flight offer stored in the local variable `flightOffer`. The airline can access the ID via the newly created local variable `id` of type `Integer` (as implicitly derived from the type of the field `flightOffer.id`). It then calls the manual method `processBooking` via a UML **self-message**; the method is defined in the `Manual` class (see Figure 2) in order to process the booking of the flight. It receives both the ID and the credit card data, since no manual method can access local variables or attributes of its callee in order to prevent the developer from introducing new information flows by reading or writing to them.

We annotate each message between IFlow agents in order to specify the security level of data any agent is able to receive. In Figure 4, each message is annotated with `{User, Airline}`, which is more restrictive than the security level of flight offers (see Figure 2) and therefore allows us to send a flight offer ID to the airline. However, the user's credit card data is even more restrictive, which is why it has to be explicitly declassified prior to being sent. This is done in a different sequence diagram after confirming the declassification with the user (see Figure 18).

Figure 5 shows an interaction with the user, with the TP requesting the user to select a flight offer from a previously received list of offers (see Figure 16). To accomplish this, the TP sends the predefined user message `GetSingleSelection` (see Figure 17) containing the list to the user interface. It answers with the user message `RetSingleSelection` containing one element from the list. We model the user selection with a generic attribute of the user class `singleSelection`, which is assigned to an implicitly declared variable `flightOffer` upon receiving of the answer by TP. Its type is derived from the type the elements in the sent list, here `FlightOffer`.

To improve readability and modularity of sequence diagrams, we allow them to reference each other via UML **Interaction Use** elements pointing to the used diagram. We identify the initial diagram used to denote the beginning of an interaction stretching over several sequences by sending a user message

from the user lifeline to any IFlow application (**Start** message in Figure 15). To access the sensors of a mobile device we will introduce several predefined methods, to be used in sequence diagrams as **self messages** just as any manual method. For now, we support the GPS sensor with the predefined methods `getCurrentGPSPos()` and `getAllGPSPos()`; the first returns the current location of the device, while the second provides the application with a list of all locations visited since it has been activated by the user. They have to be annotated in order to identify the initial security level of this data. We support the declassification of data by allowing MEL statements like `y := declassify(x)`, which declassifies data `x` and assigns it the declassification results to `y`. Such statements have to be explicitly annotated with the source and target security levels of the declassification, separated with an arrow (e.g. `User->User,Airline` in Figure 18).

4 Mapping to Code

4.1 Motivation and Outline

The IFlow approach not only allows the developer to model an IF-secure application, but also aids him with designing actual, deployable code for specific target platforms. We thus implement a transformation pipeline from the abstract application model to code, using Java as our target code language. We deliberately omit application-specific implementation details in the abstract UML model, focusing only on agent communication in order to simplify the modeling process, since such communication is generally enough to model IF. Thus, our transformations automatically generate a code skeleton which implements the agent communication and abstracts from actual application functionality by invoking methods that are to be implemented manually by the developer.

Section 4.2 outlines the model transformation steps and routines required to arrive at a Java code skeleton for an IFlow application. In order to check the resulting Java code skeletons for illegal IF violating the modeled IF properties, we employ a IF checking tool which operates on the language level. Section 4.3 describes our efforts in integrating the Jif IFC language into the IFlow approach, while section 4.4 explains our plans to focus on Joana as an IFC checking tool. In order to arrive at deployable code, the developer needs to implement the manual methods, realizing application-specific functionality. As the resulting code base needs to be checked for newly introduced IF leaks again, compartmentalized and deployed to specific target platforms, IFlow utilizes an abstraction layer as described in section 4.5. Our current target platforms are Android OS⁴ and Java EE⁵, since Java is the native language for both those platforms, which is a requirement for both considered IFC tools. Incidentally, Android is also the top platform with a market share of 46.9% in the US [1], while the Android Market is the fastest growing software store to date [2]. However, for most of its applications, security or privacy of their users is not a primary concern, as seen in a rising number of security reports on private data leakage (see e.g. [5, 8]).

⁴Android OS, a mobile platform by Google, <http://www.android.com/>

⁵Java Enterprise Edition, <http://www.oracle.com/technetwork/java/javaee/>

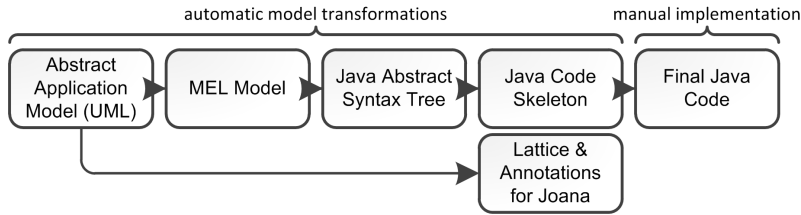


Figure 6: IFlow model transformation pipeline

4.2 Model Transformations

In order to produce deployable application code from an abstract UML model, we introduce several intermediate meta-models in order to separate different stages of model to code refinement. Figure 6 shows our model transformation pipeline, starting with the abstract model and ending with the Java code skeleton, which must be manually enriched with the implementation of manual methods in order to arrive at final code for each IFlow agent.

We use MagicDraw⁶ as our UML modeling tool, and the Eclipse IDE with its modeling tools⁷ as our model transformation platform. In order to import the MagicDraw UML model into our model transformation pipeline, the developer has to export it as an Eclipse UML2 v2.x or v3.x XMI⁸ file, which is readable by the Eclipse Modeling Tools framework. In order to simplify the invocation of the several model transformation steps within Eclipse, we provide an Eclipse plugin which adds a context menu for UML XMI files as well as all IFlow intermediate models. This context menu allows to invoke the whole transformation pipeline for the selected model, or any of the pipeline steps separately. The resulting code is generated into a predefined Eclipse project and can be immediately edited within the Eclipse IDE, e.g. in order to implement manual methods.

The first step of the IFlow model transformation pipeline is converting the abstract UML model into an instance of the MEL-metamodel. This metamodel is based on the MEL-language used in IFlow sequence diagrams; its instances contain representations of each IFlow class and agent and define agent communication interfaces as methods, including method bodies containing variable declarations, assignments and calls to manual methods represented as typed MEL syntax trees. Thus, every `synchCall` UML message in a sequence diagram is mapped to a method with the same name and one parameter of same type as the message. This method is declared inside the MEL class representing the IFlow agent which sent the message. The return type is determined by the appropriate `reply` UML message from the receiving agent. Communication with other agents within method body declarations is denoted with instances of MEL-metamodel-objects `Send`- and `Receive`, which, in turn, contain a list of sent or received variables; thus, each method body declaration begins with an instance of a `Receive` element and concludes with a `Send`.

The transformation from a UML- to a MEL-model is implemented in Java as an Eclipse plugin by creating a volatile Java representation of the abstract

⁶MagicDraw UML, <https://www.magicdraw.com/>

⁷Eclipse Modeling Project, <http://www.eclipse.org/modeling/>

⁸XMI: XML Metadata Exchange, <http://www.omg.org/spec/XMI/>

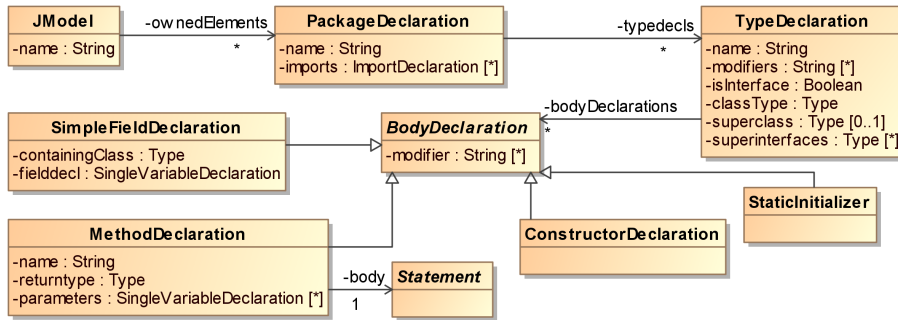


Figure 7: Excerpt from the JAST metamodel

UML model prior to generating a MEL model, utilizing Eclipse EMF (Core)⁹ as well as the ANTLR parser¹⁰ for parsing and annotating MEL expressions and statements embedded in the abstract UML diagram as text.

MEL-models are still platform independent, thus the next step is transforming such a model into an instance of a platform specific metamodel (PSM). We defined a Java Abstract Syntax Tree (JAST) metamodel, which is capable of a direct representation of the final Java code, and use it as our PSM metamodel (excerpt from the JAST metamodel is shown in Figure 7); this model-to-model transformation is implemented using QVTO¹¹, which is part of the Eclipse EMF. MEL classes representing IFlow classes and agents are being mapped to Java classes (see Listing 1), while the MEL syntax tree is being transformed into a Java syntax tree by translating each MEL statement to an appropriate Java statement or expression.

Listing 1: QVTO-code implementing the mapping of a MELClass to a JAST TypeDeclaration

```

1 mapping MELClass::transformClass() : J::TypeDeclaration {
  init{
3     info("Creating class_" + self.classname);
    var decls : Sequence(MethodDeclaration) := self.methoddecls->
      map transformDeclaration();
5  }
  name := self.classname;
7  modifiers += if self.isAbstract then 'abstract' else null endif;
  modifiers += 'public';
9  isInterface := self.isInterface;
  name := self.classname;
11 classtype := classType(self.classname);
  if (self.superclass <> null) then {
13     superclass := classType(self.superclass);
    debug("\tsuperclass:_" + self.superclass)
15 } endif;
  bodyDeclarations += self.attributes->collect(a | a.map
    transformField(self.classname));
  
```

⁹<http://www.eclipse.org/modeling/emf/?project=emf#emf>

¹⁰ANTLR (ANother Tool for Language Recognition) parser generator, <http://www.antlr.org/>

¹¹Operational QVT, <http://www.eclipse.org/m2m/>

```

17 |   bodyDeclarations += decls;
    | }

```

The last model to code transformation step is pretty printing the actual Java skeleton code from the JAST model of the IFlow application using XPand¹². The resulting code for the sequence diagram pictured in Figure 4 is given in Listing 2¹³.

Listing 2: Excerpt from Java-code implementing the sequence diagram shown in Fig. 4

```

1 | public OK bookFlightOffer(BookFlightOffer inmsg){
  |   int id = inmsg.id;
  |   CreditCardData ccd_decl = inmsg.ccd;
  |   Manual.processBooking(id, ccd_decl);
  |   return new OK();
  | }

```

In order to realize application-specific functionality, such as the function filtering flight offers depending on user request data, the developer has to manually implement it by replacing the automatically generated stub of the static method `filterOffers(...)` in the `Manual` Java class.

4.3 IFC with Jif

In order to check the generated code of IFlow applications for information flow leaks, we considered using Jif¹⁴. Jif is a security-typed extension of the Java programming language based on the Decentralized Label Model (DLM) theory [16], a language-based approach to IFC. It allows the developer to create IF-secure code by annotating it with Jif-specific language constructs and labels, and would therefore seemingly fit into our IFlow approach. By generating Jif instead of Java-code and deriving Jif-labels from modeled IF-properties, the IFlow developer could utilize the provided Jif compiler to check his application for compliance with desired static IF policies, while the Jif runtime would enforce those policies for the Java code produced by the Jif compiler dynamically. Jif (and DLM theory in particular) were specifically designed for applications with several participants who could all define their own security policies.

The Jif extension of Java seems lightweight; the developer is mainly required to provide security policies for application variables, classes, attributes and method parameters; internally, each program statement is being labeled by the Jif compiler as well. Such policies are called *labels* and have the form $\{o1 \rightarrow r1, r2; o2 \leftarrow w1, w2\}$, with $o1$, $o2$, $r1$, $r2$, $w1$, $w2$ denoting *principals*, i.e. authority entities of the application. The developer specifies a lattice of such principals, so that one principle may act for several others. In the example above, $o1$ and $o2$ are the *owners* of the policy, $r1$ and $r2$ indicate the *readers*, $w1$ and $w2$ the *writers* of the policy. Labels also form a lattice, with principle readers defining the confidentiality and writers the integrity of the annotated data type; information may therefore only flow to equally or more restrictive program

¹²XPand, <http://www.eclipse.org/modeling/m2t/?project=xpand>

¹³The full Java skeleton code for the Travel Planner application can be found at <http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/iflow/>. The skeleton code for Travel Planner agents can be found in Appendix B.

¹⁴Jif: java + information flow, <http://www.cs.cornell.edu/jif/>

statements and variables, e.g. a variable labeled with $\{o \rightarrow r\}$ is allowed to be assigned to $\{o \rightarrow\}$ but not vice versa.

Obviously, any language-based IFC solution which aspires to be used in real-world scenarios must support data declassification. Jif provides a built-in *declassify*(e, L_1 to L_2) statement, which declassifies the provided data e from security level L_1 to L_2 . Each security level has one or several owners, and this declassification statement can only succeed if called from within a class with the assigned authority principle which is equal to or can act for all label owners.

Given the support for IFC provided by Jif, we considered it to be a viable candidate for integration with the IFlow approach. We achieved the mapping from abstract UML to Jif code by using the mapping to Java code as a basis and enriching the resulting code with Jif annotations. We assumed each IFlow agent to be a Jif principle, with all agents forming a flat IF lattice if none is specified. As described in section 3, we applied security annotations to UML class attributes, indicating their security level, as well as to messages within IFlow sequence diagrams using UML constraints. We derived Jif labels from those annotations by assuming a list of agents to be a list of Jif readers, omitting the integrity and ownership aspects of Jif labels. We justify omitting the label ownership by arguing that we model and check the whole application and thus do not have an environment of mutual agent distrust as assumed by Jif. To achieve this, we assume the owner of each label to be an automatically generated, `Default` principle which each agent has the authority for, so that any agent is able to declassify any given piece of data. The set of writers is assumed to be empty, since we currently don't consider case studies with integrity properties.

Security annotations from an IFlow class diagram can be mapped directly to Jif labels for Jif class attributes. A Jif label derived from a sequence message annotation can be assumed to be the method begin label (denoting the lower bound for the security level of the callee program counter) as well as the label of the argument of the method corresponding to this message. For a reply message, the label is being applied to the return value of the Jif method. Declassification annotations can be mapped directly to Jif declassification labels. The manual methods of the application code can be implemented by the developer using Jif code sans the *declassify* statement.

Using Jif policies we were able to express a non-interference application property of our case study, namely *“travel agency never learns the user’s credit card data”*¹⁵.

Listing 3: Excerpt from Jif-code implementing the sequence diagram shown in Figure 4

```

2  public OK {Default->User , Airline;*<-}
   bookFlightOffer {Default->User , Airline;*<-} (
       BookFlightOffer [{Default->-,*<-}]{Default->User , Airline;*<-}
           inmsg){
4  int id = inmsg.id;
   CreditCardData ccd_decl = inmsg.ccd;
6  Manual.processBooking(id , ccd_decl);
   return new OK();
8  }

```

¹⁵The Jif skeleton code for a more complex version of our Travel Plannel case study including ownership and integrity aspects and using slightly different model to code mapping passed the IF check successfully

Listing 3 shows the Jif code for the sequence diagram shown in Figure 4, implemented using model to code mapping described in this report. As detailed in subsection 4.2, sequence diagram messages are mapped to method calls; thus, the `BookFlightOffer`-message is mapped to the method `bookFlightOffer` of the `Airline` class. Owner of each label is a default principle, the list of writers is empty.

The Jif code passed the IF check by the Jif compiler and verified our assumptions of Jif being a viable IFC tool for IFlow. However, we did encounter several problems which made us look for other alternatives. For one, the Jif runtime environment is not yet implemented for Android, and it is thus not possible to natively run Java code produced by the Jif compiler. However, as Jif is only an extension of Java, it is possible to omit all Jif-specific language constructs to arrive at valid Java code, which we manually verified for our case study. This could be done automatically by either generating both Java and Jif code (identical w.r.t. functionality and IF), checking the Jif code for IF leaks and deploying the Java code, or implementing a dummy Jif runtime environment for the Android platform; in both cases one loses the dynamical enforcement of IF policies. It could also be possible to port the Jif runtime to Android.

The bigger problem is the overall complexity and unintuitive design of Jif applications, as it has already been pointed out in other studies [11, 14]. Jif applications require many explicit security annotations and exception case handling (such as explicit `NullPointerException`-handling; also note the parameterizing of the `BookFlightOffer`-class with a Jif label in Listing 3, absent from the model in Figure 4), which makes manual programming with Jif a painful experience; an experience a future IFlow modeler and developer would have to live with when implementing manual methods. This also increases the complexity and required maintenance efforts for IFlow model transformations. Furthermore, Jif offers little label inference capability while enforcing IF very strictly, which results in many explicit security. Finally, the development of Jif seems to have stagnated over the years, probably due to difficulties for the developer listed above as well as little interest for the language outside of academic circles.

4.4 IFC with Joana

We identified Joana as a viable alternative for Jif as a IFC tool for Java applications and successfully evaluated it for our case study. The Valsoft/Joana project¹⁶ aims to automatically construct precise program dependence graphs (PDGs) for bytecode of moderately big Java applications (about 100kLOC [6]). Such PDGs model the information flow within applications, as they show the dependence between program statements w.r.t. IF, whereas each program statement is represented by a PDG node and edges between nodes indicate the possibility of data or control flow between them. If no such edge between two nodes exists, no information can flow between them; PDGs can therefore be used to check Java application for information flow leaks [12]. Joana aids the developer in finding such leaks by allowing him to provide an IF lattice and to identify PDG nodes serving as sources or sinks of private information, effectively labeling them with a security level. All other nodes inside the PDG are then being labeled automatically by Joana, following the rules of security level propagation.

¹⁶Valsoft/Joana project (KIT), <http://pp.info.uni-karlsruhe.de/project.php?id=30>

Joana is then able to identify whether information from a source node ever reaches a sink node with a lower or incompatible security level, violating the IF policy. Joana also supports declassification by allowing the developer to mark PDG nodes as declassification nodes, specifying the source and target security levels. Generated PDGs can have too many edges due to analysis imprecisions, but never too few; the IF analysis therefore could contain false positives, but will not miss a possible IF leak.

A connection between the Gorguen/Meseguer-style noninterference and PDGs has been established and proven [19]; it shows that statement c interferes a iff there is a path from node c to a .

We found Joana to be a viable alternative to Jif, as it requires much less effort from the modeler and the developer, allowing us to only generate Java code instead of both Java and Jif. The actual analysis is performed on Java bytecode, while the IF annotation is performed with the Joana tool and can easily be automated. In [11, 12] the authors argue that the IF analysis using PDGs is more precise than with Jif as well as easier, since the developer only has to annotate the information sources, sinks and declassification statements. We evaluated Joana by generating Java code for our case study using the mapping described in previous chapters and omitting the Jif specific constructs. We annotated the resulting bytecode with Joana using a lattice and security levels derived from the abstract model, which can later also be automated using our model transformations. To verify the security property of our case study “*travel agency never learns the user’s credit card data*” we only needed four annotations (which is even less we had to include in our model in order to derive security annotations for the formal model and the Jif code), while the original Jif implementation contained over a hundred label annotations to show that the same property holds.

Joana annotations can be methodically derived from the IFlow security annotations by assuming each unique annotation to be a security domain and use those to infer an IF lattice. Each annotated IFlow model element would therefore be a Joana source and a sink at the same time. Class attributes annotations can thus be adapted directly, while message annotations could be applied to either message classes or the appropriate method parameters respectively. IFlow model transformations would also have to generate a unique declassification method for each cross product of all security domains. Those can be annotated as Joana declassification statements using the modeled source and target security domains.

Notably, the Joana tool is currently unable to check the integrity aspects of IF, which, however, is not a limitation for our project at this point, since our case studies do not consider integrity properties. Furthermore, the Joana tool is being actively developed and supported, and we are actively collaborating with the Joana team within the scope of the RS³ priority programme. The current version of Joana tool is very much work in progress, and we plan to contribute to improving it.

4.5 Deployment to Target Platforms

4.5.1 Motivation and Goals

As we have to generate Java code in order to employ automatic IF checking tools like Jif or Joana, we chose Android as our target mobile platform, and Java EE as the platform for IFlow services. Android is currently enjoying a growth in popularity, with supported hardware ranging from smartphones to tablets to even TVs running on Android. It employs several security mechanisms like application sandboxing and an application permission system, which, however, are not enough to protect the user's private information stored on his device. Reports on applications leaking confidential data like address books, phone IDs and location data accidentally or on purpose to third parties appear frequently in the media and academic papers (e.g. [8]). The main reasons for this is the open nature of the Android application marketplace, with anyone being able to submit his own application without prior security screening, as well as the rather coarse-grained, inflexible permission system. A set of permissions defining whether the application is allowed to access the internet, device sensors etc. is determined by the developer and has to be approved by the user prior to the installation of the application. However, application permissions do not define which data can be sent over the internet, or how the application deals with the sensor readouts; an application requiring both internet access and access to the GPS sensor may use the location data to submit anonymized location statistics to the developer, or leak the location of the user to any third party. Besides, applications with seemingly harmless permissions can collaborate with each other in order to create covert information flows in order to exfiltrate confidential user information [13, 18].

With IFlow, we plan to address those shortcomings of the Android platform by providing the developer with a model-driven software engineering approach for IF-secure applications. We encourage him to focus on IF aspects of his application in early development stages, not having to deal with implementation details or rely on faulty Android security systems. By providing means to verify IF properties and generate deployable, runnable code, our model-driven approach is unique and goes beyond projects like XManDroid [7] or TainDroid [9] (realtime privacy monitoring). This section will describe how we plan to deploy automatically generated and manually enriched code to Android and Java EE services.

4.5.2 Mapping to Platform Specific Code

In subsection 4.2 we described the mapping from the abstract UML model to Java code skeletons. However, it is not trivial to translate those skeletons to Android or Java EE specific code. Android applications have a unique structure and rely on Android specific API; we thus need an intermediate abstraction level between generated code and platform specific API. Furthermore, the generated Java code is realized as a monolithic applications, with agent communication implemented as Java class instances calling each other's methods. This was done to be able to analyze the code with IF checkers Jif and/or Joana, but such code cannot be deployed to the target platform directly. We therefore implement a wrapper library, which abstracts from platform-specific functionality like agent communication implementation, device sensor or database access and provides



Figure 8: Application communication using direct method calls for IFC checks

its own API, which we use in our generated code skeleton. The wrapper implementation has exchangeable versions. A prototype version calls agent methods directly for communication and emulates database access in pure Java, while a platform-specific version implements this functionality using platform-specific API, to be used for deployment purposes.

Our modeling guidelines allow the modeler to access various mobile device sensors via calls to predefined MEL methods, e.g. *currentGPSPosition()*. The Java implementation of those methods also internally employs the wrapper library, which will either utilize the Android API or prototype method stubs denoting the IF within those methods.

One challenge is to allow the developer to implement manual methods without giving him the opportunity to introduce additional information flows, accidentally or maliciously. The IFlow approach cannot expose the native platform API to the developer, as it opens up many possibilities of potential information leaks (e.g. directly opening a connection to a third party service from within a manual method). We therefore only give the developer access to a subset of our wrapper API, which is shown not to leak information, and forbid the usage of declassification routines.

Another challenge of deploying the generated code to specific platforms is agent to agent communication. Communication with the user interface is modeled with sequence messages from a predefined set of user messages. An implementation of this interface maps those messages to Android GUI elements like input boxes or confirmation messages. Communication between apps and services, modeled as direct method invocation, is routed through our wrapper library in the generated Java code (the wrapper API for this is shown in Listing 4¹⁷) The prototype implementation will directly invoke method calls of agent instances to simplify IF checks by Jif/Joana as seen in Figure 8. The deployment version utilizes the platform specific API to route the messages to the actual, deployed app on the mobile device or a service on the internet, as seen in Figure 9. Each message is being en- and decoded with JSON.

Listing 4: Wrapper API for sending messages

```

2  /**
   * Sends a message to an agent
   *
4  * @param message The message to send
   * @param agent The agent which will receive the message
6  * @return The response
   */
8  public Message send(final Message message, Agent agent);
  
```

The communication with IFlow services is implemented as HTTP requests to appropriate service URLs as provided by Java EE, managed by the wrapper.

¹⁷Full API of the current version of the wrapper is shown in Appendix C

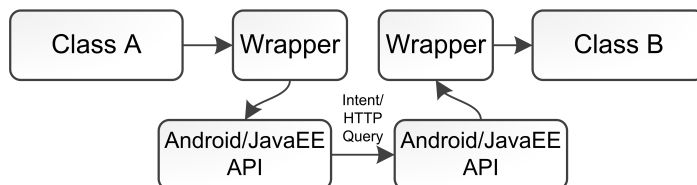


Figure 9: Platform-specific application communication for deployment

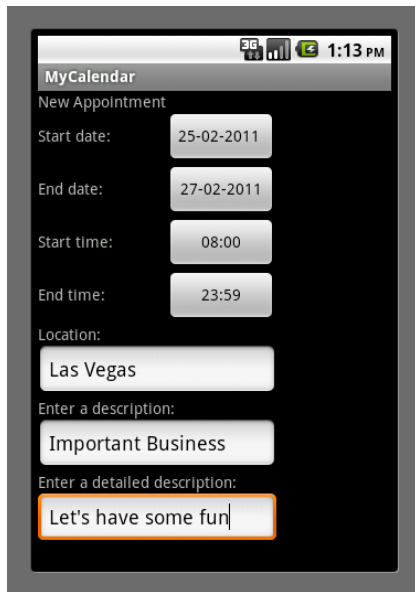
The communication between Android apps can be realized in a number of ways; for one, Android apps can provide external interfaces as *Content Providers*, however those are only used for data queries. The alternative is to implement Android *Broadcast Receivers*, which can receive and react to system-wide messages; but broadcasting sensitive information to all applications on a mobile device is not a secure way to handle user’s data. We therefore realize IFlow application interfaces with Android *Services*, which is an optional component of any Android application. Services can run in the background and receive messages from other applications via the asynchronous inter-application Android messaging system with so-called *Intents*. Since IFlow currently only supports synchronous communication, our wrapper is designed to handle the synchronization of messages between IFlow applications.

Android implements sandboxing and a permission system for its applications to remain secure. As already outlined, those security mechanisms are not enough to protect user’s confidential data from information leaks. We therefore only implicitly rely on the application sandboxing aspect of Android by assuming that IFlow applications cannot communicate with or influence each other besides by using shared resources (e.g. SD card). Android permission play no role in our IFC, and we thus only need to specify the minimal set of permissions required by the generated application. To automate this process we will use Stowaway¹⁸, which is able to analyze Android applications and generate such a set [10].

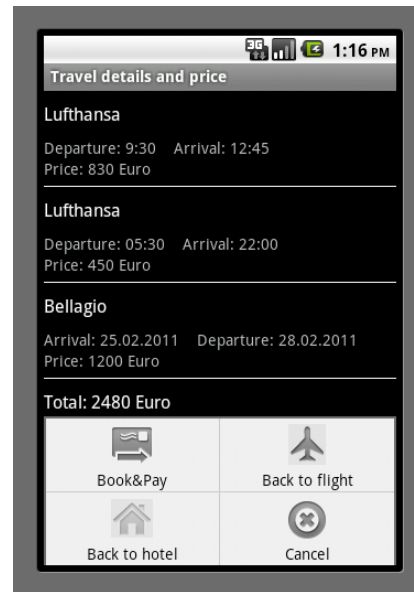
5 Conclusion and Outlook

We were successful in creating a model-driven approach for designing IF-secure applications consisting of smartphone apps and webservices. We accomplished this by creating concrete modeling guidelines and defining a mapping from an abstract model to code skeletons, which in turn can be implemented manually by the developer. Using already available automatic, language-based IF checking tools (we evaluated both Jif and the preferable alternative, Joana) we are able to show noninterference properties for the modeled application. By implementing an abstraction library we will also be able to deploy the resulting code to platforms like mobile devices and webservices. In order to verify IF properties beyond simple noninterference we also defined a mapping between the abstract mapping and a formal model, which can be used with the interactive verifier KIV [4]. We will present achieved results in this field in an upcoming technical report.

¹⁸Stowaway: A static analysis tool and permission map for identifying permission use in Android applications, <http://www.android-permissions.org/>



(a) Travel Planner: Creating appointment



(b) Travel Planner: Selecting flight offer

Figure 10: Screenshots of the Android application “Travel Planner”

We evaluated our approach with the case study “Travel Planner” by creating an abstract model of the application and writing code skeletons based on this model. We also verified a simple noninterference property for this application using Jif and Joana. Automatic model to code transformations are work in progress, as is a full implementation of the abstraction wrapper library. We also implemented full, runnable Android and Java EE service code for the extended version of our case study, which includes a calendar application and hotel booking using the current implementation of the wrapper (application screenshots are shown in 10(a) and 10(b)).

References

- [1] Press Release: comScore Reports November 2011 U.S. Mobile Subscriber Market Share. http://www.comscore.com/Press_Events/Press_Releases/2011/12/comScore_Reports_November_2011_U.S._Mobile_Subscriber_Market_Share, 2011.
- [2] AndroLib: Android Market New Apps and Games Directory. Statistics. <http://www.androlib.com/appstats.aspx>, 2012.
- [3] M. Balanza, K. Alintanahin, O. Abendan, J. Dizon, and B. Caraig. Droid-dreamlight lurks behind legitimate android apps. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 73–78, oct. 2011.
- [4] M. Balsler, W. Reif, G. Schellhorn, and K. Stenzel. KIV 3.0 for Provably Correct Systems. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *Proc. Int. Wsh. Applied Formal Methods*, volume 1641 of *LNCS*, pages 330–337. Springer, 1999.
- [5] L. Batyuk, M. Herpich, S.A. Camtepe, K. Raddatz, A. Schmidt, and S. Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 66–72, oct. 2011.
- [6] D. Binkley, M. Harman, and J. Krinke. Empirical study of optimization techniques for massive slicing. *ACM Trans. Program. Lang. Syst.*, 30, November 2007.
- [7] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, Apr 2011.
- [8] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security, SEC’11*, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.
- [9] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of OSDI 2010*, October 2010.
- [10] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security, CCS ’11*, pages 627–638, New York, NY, USA, 2011. ACM.
- [11] C. Hammer. Experiences with pdg-based ifc. In F. Massacci, D. Wallach, and N. Zannone, editors, *International Symposium on Engineering Secure Software and Systems (ESSoS’10)*, volume 5965 of *LNCS*, pages 44–60. Springer-Verlag, February 2010.

- [12] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, December 2009. Supersedes ISSSE and ISoLA 2006.
- [13] N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988.
- [14] D. Hassan, S. El-Kassas, and I. Ziedan. Developing a security typed java servlet. In *Fourth International Conference on Information Assurance and Security – ISIAS ’08.*, pages 215 –220, sept. 2008.
- [15] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. ’SecureMDD: A model-driven development method for secure smart card applications’. In *Availability, Reliability and Security, 2009. ARES ’09. International Conference on*, pages 841 –846, march 2009.
- [16] A.C. Myers and B. Liskov. Protecting privacy using the decentralized label model. In *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, pages 89 – 116, 2003.
- [17] A.C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. *Software release. Located at <http://www.cs.cornell.edu/jif>*, 2005, 2001.
- [18] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*. The Internet Society, 2011.
- [19] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology*, 15(4):410–457, October 2006.

A Travel Planner UML Model

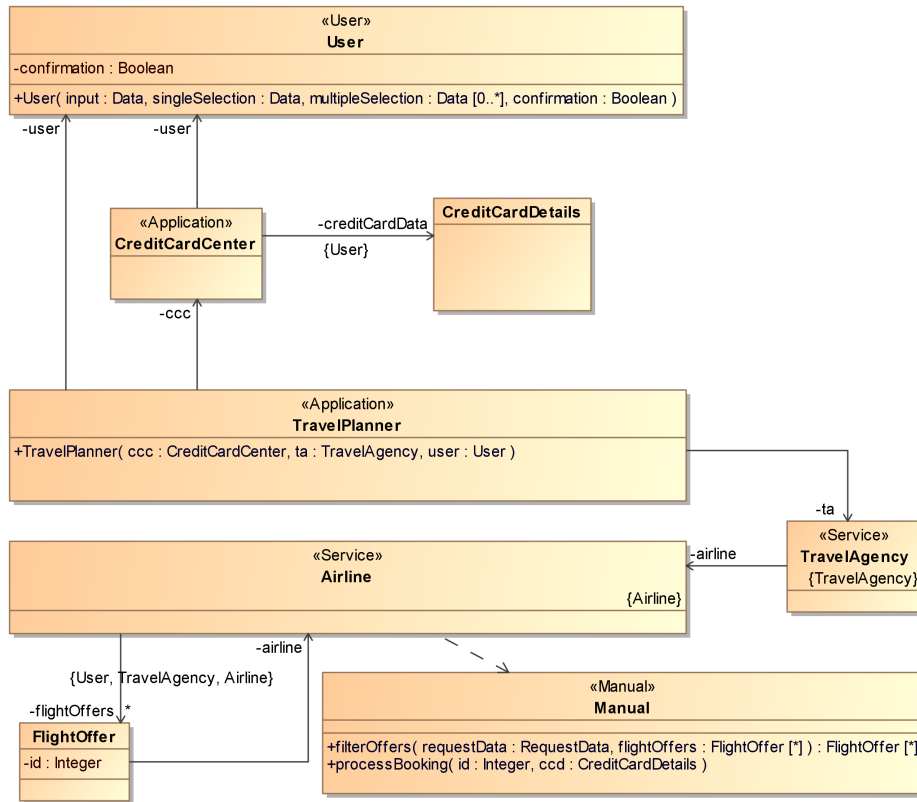


Figure 11: Application component diagram

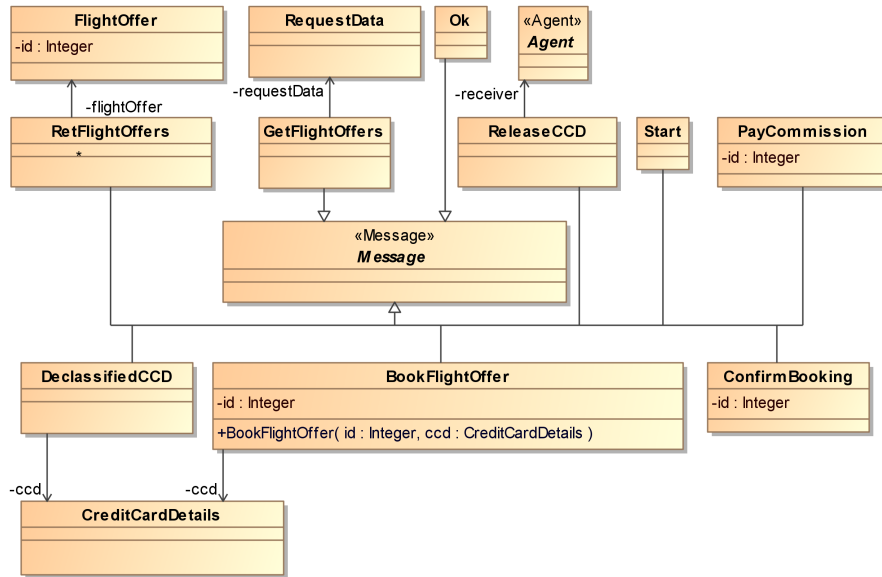


Figure 12: Application messages diagram

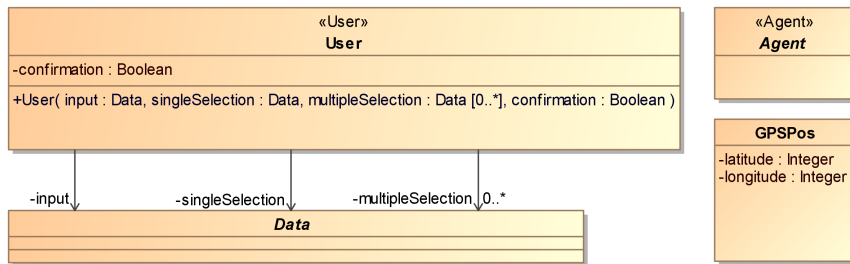


Figure 13: Standard classes

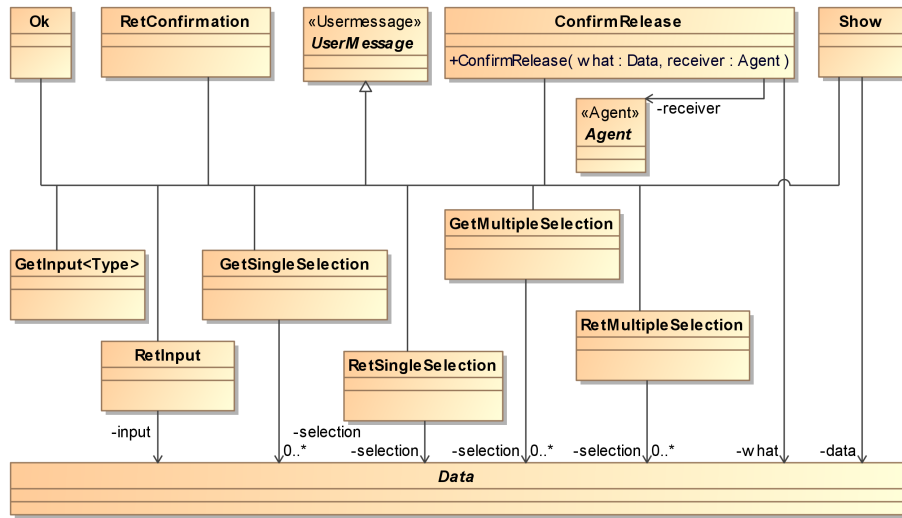


Figure 14: Standard user messages

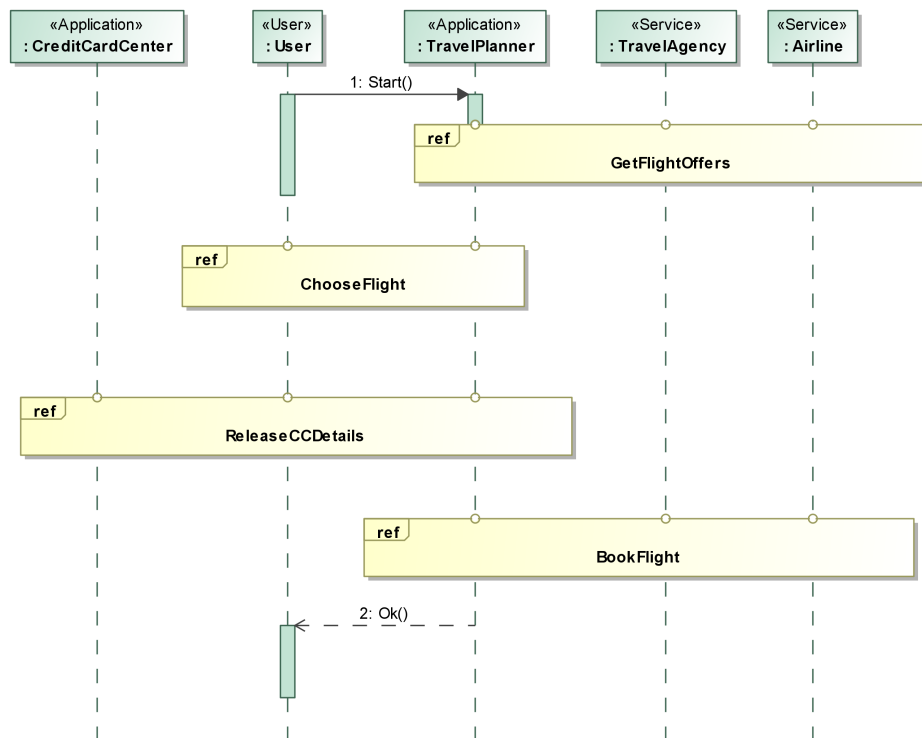


Figure 15: Behavior overview diagram

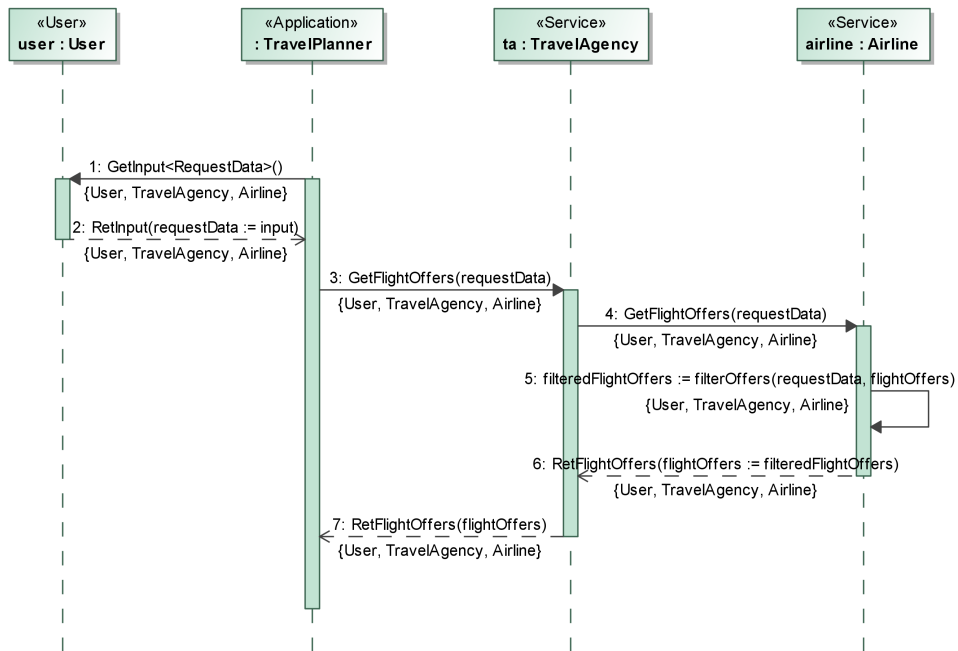


Figure 16: Request flight offers

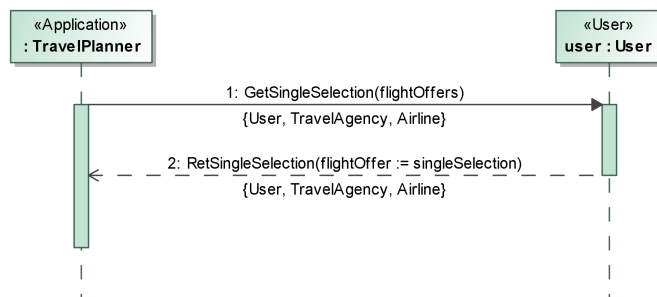


Figure 17: Choose a flight

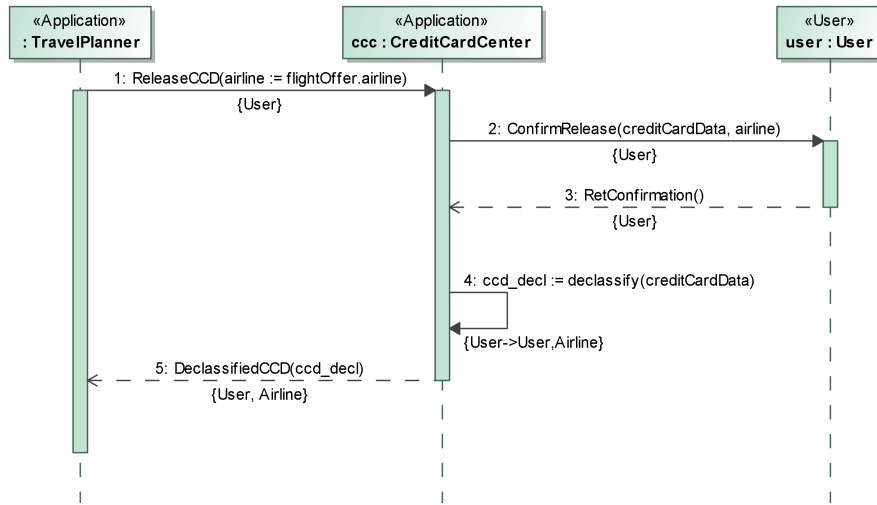


Figure 18: Release credit card details

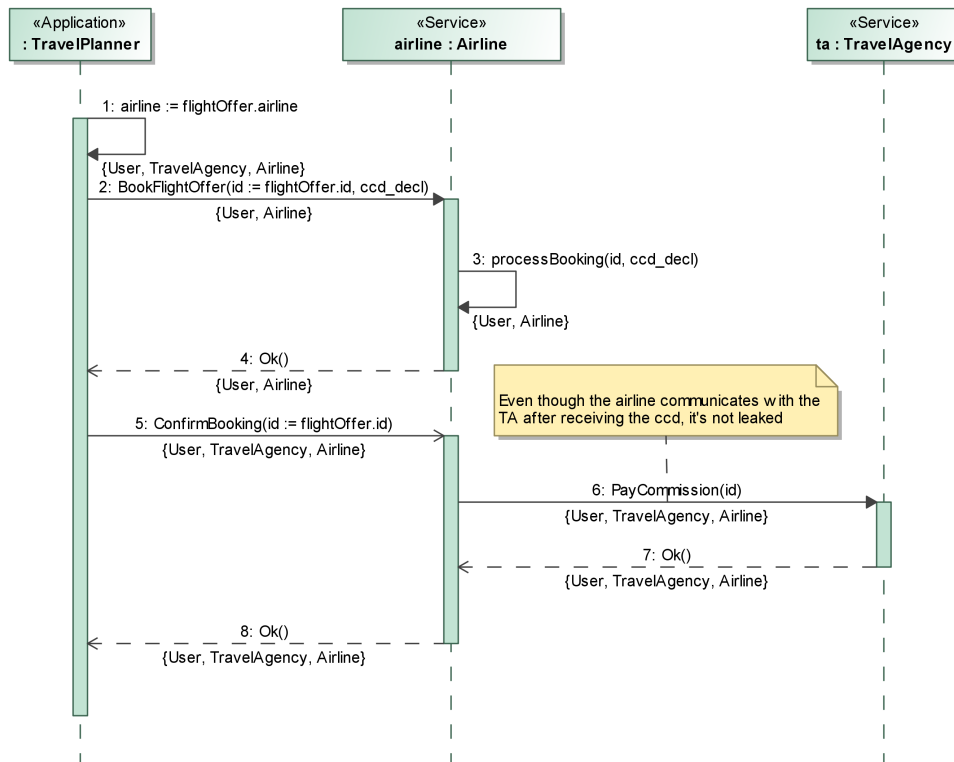


Figure 19: Booking flight

B Travel Planner Agent Skeleton Code

Listing 5: TravelPlanner.java

```
1 package travelplanner;
3 public class TravelPlanner extends Agent {
5     private static TravelPlanner instance;
7     public static TravelPlanner getInstance() {
9         if(instance == null)
10            instance = new TravelPlanner();
11        return instance;
12    }
13    static private UserInterface ui = new UserInterface();
14    static private TravelAgency travelAgency;
15    static private CreditCardCenter ccc;
17    public TravelPlanner() {
18        ccc = CreditCardCenter.getInstance();
19        travelAgency = TravelAgency.getInstance();
20    }
21    public void start () {
22        // GetFlightOffers
23        RetInput inmsg = ui.getInputRequestData();
24        RequestData requestData = (RequestData) inmsg.input;
25        RetFlightOffers inmsg2 = travelAgency.getFlightOffers(new
26            GetFlightOffers(requestData));
27        ListOfFlightOffer flightOffers = inmsg2.flightOffers;
29        // ChooseFlight
30        RetSingleSelection inmsg3 = ui.selectOne(new GetSingleSelection
31            (flightOffers));
32        FlightOffer flightOffer = (FlightOffer) inmsg3.selection;
33        // ReleaseCCDDetails
34        DeclassifiedCCD inmsg4 = ccc.releaseCCD(new ReleaseCCD(
35            flightOffer.airline));
36        CreditCardData ccd_decl = inmsg4.ccd;
37        // BookFlight
38        Airline airline = flightOffer.airline;
39        OK inmsg5 = airline.bookFlightOffer(new BookFlightOffer(
40            flightOffer.id, ccd_decl));
41        OK inmsg6 = airline.confirmBooking(new ConfirmBooking(
42            flightOffer.id));
43    }
}
```

Listing 6: TravelAgency.java

```
1 package travelplanner;
3 public class TravelAgency extends Agent {
5     private static TravelAgency instance;
7     static private Airline airline;
```

```

9   public static TravelAgency getInstance(){
11       if(instance == null)
12           instance = new TravelAgency();
13       return instance;
14   }
15   public TravelAgency() {
16       airline = Airline.getInstance();
17   }
18
19   /**
20    * GetFlightOffers message handling
21    *
22    */
23   public RetFlightOffers
24       getFlightOffers(
25       GetFlightOffers inmsg) {
26       RequestData requestData = inmsg.requestData;
27       RetFlightOffers inmsg2 = airline.getFlightOffers(new
28           GetFlightOffers(requestData));
29       ListOfFlightOffer flightOffers = inmsg2.flightOffers;
30       return new RetFlightOffers(flightOffers);
31   }
32
33   public OK payComission(PayComission inmsg) {
34       int id = inmsg.id;
35       return new OK();
36   }
37 }

```

Listing 7: CreditCardCenter.java

```

1   package travelplanner;
2
3   public class CreditCardCenter extends Agent{
4
5       private static CreditCardCenter instance;
6
7       public static CreditCardCenter getInstance(){
8           if(instance == null)
9               instance = new CreditCardCenter();
10          return instance;
11      }
12
13      private CreditCardData ccd = new CreditCardData();
14      private UserInterface ui = new UserInterface();
15
16      public CreditCardData getCcd(){
17          return ccd;
18      }
19
20      /**
21       * ReleaseCCD message handling
22       *
23       */
24      public DeclassifiedCCD releaseCCD (ReleaseCCD inmsg) {
25          Agent airline = inmsg.receiver;
26          RetConfirmation inmsg2 = ui.confirmRelease(new ConfirmRelease(
27              ccd, airline));

```

```

29     CreditCardData ccd_decl = Declassify.declassify(ccd);
        return new DeclassifiedCCD(ccd_decl);
31     }
    }
}

```

Listing 8: Airline.java

```

package travelplanner;
2
import java.util.ArrayList;
4 import java.util.List;

6 public class Airline extends Agent {

8     private static Airline instance;
        public static TravelAgency ta;

10
12     public static Airline getInstance(){
        if(instance == null)
            instance = new Airline();
14     return instance;
        }

16
18     private List flightOffers = new ArrayList();

19
20     public Airline(){
        FlightOffer fo = new FlightOffer(this, 1);
        try{
22         flightOffers.add(fo);
        }
24     catch(NullPointerException e){}
        catch(ClassCastException e){}
26     catch(IllegalArgumentException e){}

28     ta = TravelAgency.getInstance();
        }

30
32     /**
        * GetFlightOffers message handling
        *
34     */
    public RetFlightOffers
36     getFlightOffers(
        GetFlightOffers inmsg) {
38     RequestData requestData = inmsg.requestData;
        ListOfFlightOffer filteredFlightOffers = Manual.filterOffers(
            requestData, flightOffers);
40     return new RetFlightOffers(filteredFlightOffers);
        }

42
44     /**
        * BookFlightOffer message handling
        *
46     */
    public OK
48     bookFlightOffer(
        BookFlightOffer inmsg) {
50     int id = inmsg.id;
        CreditCardData ccd_decl = inmsg.ccd;
52     Manual.processBooking(id, ccd_decl);
        return new OK();
    }
}

```

```
54     }
55
56     /**
57      * ConfirmBooking message handling
58      *
59      */
60     public OK confirmBooking(ConfirmBooking confirmBooking) {
61         int id = confirmBooking.id;
62         OK inmsg2 = ta.payComission(new PayComission(id));
63         return new OK();
64     }
65
66 }
```

C Wrapper API

Listing 9: Interface API for agent communication

```
1 public interface CommunicationService {
3     /**
4      * Sends a message to an agent
5      *
6      * @param message The message to send
7      * @param agent The agent, that will receive the message
8      * @return The response
9      */
10    public abstract Message send(final Message message, Agent agent);
11 }

```

Listing 10: Interface API for database interaction

```
2 public interface DatabaseService {
4     /**
5      * Closes the connection to the database
6      */
7     public abstract void close();
8
9     /**
10    * Adds an object to the database
11    *
12    * @param objectToSave The object to add
13    * @return The primary key id, generated by the SQLite
14    *         Database
15    */
16    public abstract long addObject(Object objectToSave);
17
18    /**
19    * Returns a cursor, that has the information of the queried
20    * database
21    * The information of the object referenced by the objectId
22    * The Cursor and thereafter the Database have to be closed
23    * manually after usage
24    *
25    * @param objectId The id of the object to query
26    * @param objectToQuery The obj that references the table to
27    *         query
28    * @return The cursor with the information
29    */
30    public abstract Cursor queryDatabase(int objectId, Class
31    objectClassToQuery);
32
33    /**
34    * Returns a cursor with all objects, that are saved in the db
35    * Discouraged, because query is on all columns
36    * @param objectToQuery The object, of which table to query
37    * @return A cursor with all objects
38    */
39    public abstract Cursor queryForAll(Class objectClassToQuery);
40
41    /**

```

```

38     * Returns all objects from the database that fit to the
        specified class
        *
40     * @param objectClassToQuery The class, that specifies the table
        to query
        * @return An ArrayList containing all objects of the
        table
42     */
public abstract ArrayList<Object> queryForAllObjects(
44     Class objectClassToQuery);

46     /**
        * Returns a cursor with the objects from the database, that
        match the selection
48     * @param objectToQuery The object, of which table to query
        * @param selection A selection string, formatted as an sql-
        where-clause without the 'where'
50     * @param selectionArgs Specifies the '?' in the selection
        argument for the query
        * @return A cursor with the selected objects
52     */
public abstract Cursor queryWithSelection(Class
        objectClassToQuery,
54     String selection, String[] selectionArgs);

56     /**
        * Returns the cursor containing the object with the given id
58     * Cursor and connection have to be closed after usage
        *
60     * @param objectId The id, that references the database entry
        * @param tableName The name of the table
62     * @return A cursor containing the object
        */
64     public abstract Cursor queryDatabase(long objectId, String
        tableName);

66     /**
        * Queries the database and returns the object with the given id
68     * Only works if all objects have default constructor
        *
70     * @param objectId The id of the queried object
        * @param className The fully classified name of the class to
        query
72     * @return The returned object
        */
74     public abstract Object query(long objectId, Class objectClass);

76     /**
        * Returns a list object of the given class and referenced by the
        id of the object having the
78     * list as field
        *
80     * @param listTypeClass The class of the list
        * @param objectOfTheReferenceId The id of the object
        referencing the list
82     * @return The list object
        */
84     public abstract Object queryDatabaseForList(Class<?>
        listTypeClass,
        long objectOfTheReferenceId, String paramType);
86     /**

```

```

88     * Updates the object in the database by deleting it and adding
      * it new
      *
90     * @param objectToUpdate    The object to update
      * @return                    True, if update was successful
92     *                            False, instead
      */
94     public abstract boolean updateObject(Object objectToUpdate);

96     /**
      * Delete an object from the database
      *
98     * @param objectToDelete    the object to be deleted from the
      *                            database
100    * @return                    true, if the deletion was successful
      *                            false, otherwise
102    */
      public abstract boolean deleteObject(Object objectToDelete);
104

106    /**
      * Creates a table for the given object, if it does not already
      * exist
      *
108    * @param objectToSave    The object for which a table is needed
      */
110    public abstract void createTableIfNotExists(Object objectToSave);
112 }

```