# Consistency Management in Web Application Development

## Masterarbeit

zur Erlangung des Grades eines Master of Science
im Studiengang Web Science

vorgelegt von

Ruslan Kvashchanka

| | |
|---|---|
| Erstgutachter: | Prof. Dr. Ralf Lämmel<br>Institut für Informatik |
| Zweitgutachter: | Msc. Marcel Heinz<br>Institut für Informatik |

Koblenz, im Juni 2016

**Erklärung**

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

| | Ja | Nein |
|---|---|---|
| Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden. | ☐ | ☐ |
| Der Veröffentlichung dieser Arbeit im Internet stimme ich zu. | ☐ | ☐ |

.....

(Ort, Datum) (Ruslan Kvashchanka)

## Abstract

Web-applications evolution often coerced with tasks that are targeted to maintain the application component relationships. Lack of this maintenance negatively affects loosely connected elements of the system. For instance, evolution of one component (e.g., the model in the MVC sense) may lead to inconsistency in another component (e.g., the view in the MVC sense). This research is intended to study such loose coupled component inconsistencies by examining development processes of existing application that use Python-based Django web development framework. The goal is to understand, study and describe ways that help to detect such inconsistencies and prevent them from leaking to the production system. The research is comprised of two parts: getting the experience of industrial developers by conducting a number of interviews and prototyping and implementing a tool that would help to pinpoint the relationships between application components prone to the inconsistencies. The latter in turn includes application of the tool on an open-source project from github.com.

## Acknowledgements

I would like to thank Prof. Dr. Ralf Lämmel for assistance and support during the work on the research.

# Contents


**1 Introduction** **1**

1.1 Problem Context . . . . . 1
1.2 Research Problem . . . . . 2
1.3 Research Questions . . . . . 4
1.4 Research Contribution . . . . . 4
1.5 Thesis Structure . . . . . 4

**2 Background** **6**

2.1 Data Type . . . . . 6
2.2 Type System . . . . . 6
2.3 Object-Oriented Design . . . . . 7
2.4 Web Application . . . . . 7
2.5 Web Application Framework . . . . . 8
2.6 Model View Controller . . . . . 8
2.7 Loose Coupling . . . . . 8
2.8 Version Control System . . . . . 9
2.9 Software Development Processes . . . . . 9
2.10 Software Testing . . . . . 9
2.11 Issue Tracking System . . . . . 10
2.12 Python . . . . . 10
2.13 Django . . . . . 10

**3 Related Work** **12**

**4 Opportunities and Solutions** **14**

4.1 Understanding the Issue . . . . . 14
4.2 Issue Identification . . . . . 18

4.2.1 Issue Tracker and Code Repository Data Mining . . . . . 18
4.2.2 Emulating Development Process . . . . . 18
4.2.3 Gathering the Opinions . . . . . 19
4.3 Issue Prevention . . . . . 19
4.4 Summary and Directions . . . . . 19

**5 Interviews** **21**
5.1 Interview Methodology . . . . . 21
5.1.1 Goals and Theoretical Background . . . . . 22
5.1.2 Questionnaires . . . . . 22
5.2 Interview Process . . . . . 25
5.3 Interviews Output Discussion . . . . . 25
5.4 Analysis and Summary . . . . . 28
5.4.1 Testing . . . . . 28
5.4.2 Standards . . . . . 29
5.4.3 Methodologies . . . . . 29

**6 Prototype Method** **30**
6.1 Requirements . . . . . 30
6.2 Discussion and Challenges . . . . . 31
6.3 Design . . . . . 32
6.3.1 Coverage Plugin API . . . . . 33
6.3.2 Django Template Coverage . . . . . 33
6.3.3 Tool Design . . . . . 34
6.4 Implementation . . . . . 34
6.4.1 Execution Phase . . . . . 35
6.4.2 Analysis Phase . . . . . 35
6.4.3 Report Phase . . . . . 36
6.5 Summary . . . . . 37

**7 Case Study** **39**
7.1 Goals . . . . . 39
7.2 Available Applications . . . . . 39
7.3 Django Activity Stream . . . . . 41
7.3.1 Dependencies . . . . . 42
7.3.2 Structure . . . . . 42

7.3.3 Statistic . . . . . 42
7.4 Preconditions . . . . . 43
7.4.1 Setup . . . . . 43
7.4.2 Configuration . . . . . 43
7.4.3 Artificial Changes . . . . . 44
7.5 Application . . . . . 44
7.5.1 Launch - Execution Stage . . . . . 44
7.5.2 Launch - Report Stage . . . . . 44
7.6 Outputs . . . . . 45
7.7 Validation . . . . . 47
7.8 Summary . . . . . 49

**8 Conclusions and Future Work** **50**

# List of Figures

1.1 Model View Controller Architecture with Components . . . . . 2

2.1 Django-based web-application architecture . . . . . 11

6.1 Coverage.py cycle flow . . . . . 33
6.2 Code analyzer plugin prototype functioning principle . . . . . 34
6.3 Algorithm of processing a coverage frame and filling the model-template mapping . . . . . 35
6.4 Template-Model mapping . . . . . 36
6.5 Template-Variable mapping . . . . . 37

# Chapter 1

# Introduction

This chapter introduces the component inconsistencies that may arise during web application development, states the research problems and questions and outlines the research contributions.

## 1.1 Problem Context

Web application development entails many aspects that have to be taken into account in order to successfully implement a product. One of these aspects is the proper tool set choice. Currently a huge number of ad-hoc technologies is available to simplify and speed up the development processes. These technologies, or frameworks, vary from single page web application helpers to thorough systems that allow to build and support complex enterprise services.

Application of one or another framework requires for the developer not only to master use of the tool but also to be able to overcome and solve possible issues caused by the tool/language specifics that may arise on the implementation stage. One of such issues is the framework component synchronization during application evolution process.

Components of a web framework are "building blocks" that, being logically and structurally independent, comprise the tool. One of the most effective architectures of web application frameworks is Model View Controller (MVC). The MVC architecture is reflected on Figure 1.1.

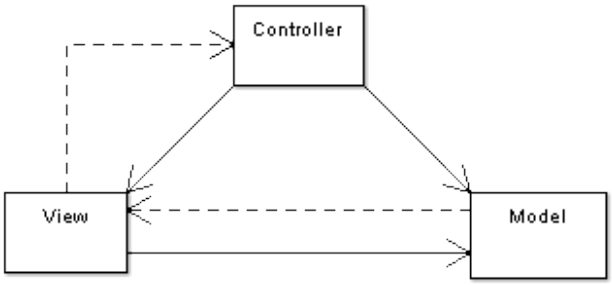


**Figure 1.1** Model View Controller Architecture with Components

Briefly, the approach is to divide the system into three parts in order to achieve a separation between internal representations of data from the ways this data is presented to or processed by the user [19].

The application evolution process in turn is a development process, during which an application is updated with new features and functionality. During this process one has to take care that the logically and structurally independent components of the application are created and updated consistently. In other words, the consistency of the loosely-coupled components has to be ensured.

Most of inconsistencies, particularly in case of MVC components, can be detected and resolved during the development and testing stages. However, in some cases the inconsistencies may leak up to the production system, potentially creating functionality or representation issues.

The particular case of such synchronization, is presented in the next *Research Problem* section.

## 1.2 Research Problem

The loosely coupled components synchronization problem (further in this work referenced as component inconsistency issue/problem or just the issue) may arise when Model component changes are not taken into account in the View component. The following Model changes are potential reasons for the View inconsistency:

- type of a class attribute changes
- method return type changes

- type inheritance changes
- attribute or method is removed

First of all, the class attribute class change. This case is relevant for primitives like *int, long, boolean* as well as for complex types. Next, the class method return type. The potential inconsistency scenario is when a view calls the method directly. Consequently, the type inheritance tree might be changed. The corresponding changes has to be handled in the view accordingly. Finally, one can remove a property / method from the type which is used by the view. Affected view, like in the previous cases, has to be made consistent with the model.

Consider revisions of an application *(R,...,R1,...,R2,..,R')*. Where *R* is an initial revision, *R'* is the final revision in sense that the system version is deployed to productive servers. Suppose, one of the Model changes described above took place between *R* and *R'*, in the revision *R1*. In case if the view was not updated accordingly, the system components became inconsistent. Of course, in the real world scenario there exists certain mechanisms that allow to detect the mistake and fix it in revision *R2* or even prevent such case before *R1* goes to the version control system. Typically, such mechanisms include

1. unit tests
2. functional tests
3. type-checking systems of the underlying programming language
4. development environment tool (IDE) plugins
5. manual QA tests procedures

The mechanisms 1-4 allow one to prevent the issue before or directly after the revision *R1*, so that the issue can be correspondingly handled in *R1* or *R2*.

Naturally, the point 3 - type-checking systems of the underlying programming language would not work in weakly typed languages and the corresponding frameworks which are based on such languages. This work is intended to check how relevant might be the issue, study the issue prevention mechanisms and outline the recommendations on how to prevent or if the prevention mechanisms are not the case, detect and fix the component inconsistencies in time.

## 1.3 Research Questions

The research is intended to analyze, study and answer the following questions:

***RQ1*** *Can a component inconsistency go unnoticed to the version control system or even to the production deployments?*

It is necessary to check weather it can cause invalid functionality of release deployments. Three related scenarios are:

1. The issue was detected during the development process.
2. The issue went unnoticed for the developer and leaked to the version control system.
3. The issue affected the deployment deliverable (QA and Production releases).

***RQ2*** *(How) Is it possible overcome or make the detection of component inconsistencies feasible in scope of the technologies vulnerable to them?*

This part of the research is targeted at the development and practical application of a prototypical method that could help the developers to identify the issue by highlighting the potential consistency issue spots.

## 1.4 Research Contribution

During the research the following contributions were made:

1. A set of recommendations on how to prevent the issue during the development process has been compiled from the outputs of a number of interviews with industrial Django developers.
2. A prototypical method for highlighting loosely coupled component relationships vulnerable to the issue has been developed and applied on practice on a Django-based project.

## 1.5 Thesis Structure

This thesis is organized as follows: after the introductory chapter the necessary terminology that is used across the work is presented in *Background* chapter. After that the related researches are highlighted in the *Related Work* chapter. Next,

in the *Opportunities and Solutions* section the directions of the research are presented and discussed. The following *Interviews* chapter describes the interviews with commercial application developers in details and presents the recommendations for the issue prevention. Consecutively, in chapter *Prototype Method* the requirements, design and implementation of the analysis method is presented. Next *Case Study* section describes practical application of the developed method on a real project from github.com. Finally, the research is summarized and recommendations for future work are given in the last *Conclusions and Future Work* chapter.

# Chapter 2

# Background

In this chapter the necessary research background is described along with the technologies that were used for the research.

## 2.1 Data Type

In order to understand what is the *Type System* of a language the *Data Type* has to be defined first. In Computer Science type (*data type*) is defined as a combination if the following [23]:

- determination of the possible values or data classification (including integer, real and boolean)
- operations allowed for the type
- interpretation of the data
- value storing mechanisms

## 2.2 Type System

The *type system* of a programming language declares the way of management and construction of data types of the language. More concrete, the way of mapping of values and expressions to a type, possible manipulation and interaction of types. Or from the other hand, a type system is represented by a collection of rules which

assign a type property to different blocks of program like variables, expressions, functions or modules [17]. In scope of this work the following type systems are considered:

- dynamic (the type can be changed at runtime [27])
- strong (hard imposition of the type rules; all types are defined and known at the compile time [27])
- weak (the type is a value can be altered at runtime [27])
- gradual (a combination of *weak* and *strong* typing "with the programmer controlling the degree of static checking by annotating function parameters with types, or not." [25], [29])
- duck (where type of an object is determined by its runtime capabilities contrary to its definition by the class [16])

Type system definition contributes to understanding underlying reasons of the component inconsistencies.

## 2.3 Object-Oriented Design

*Object-Oriented (OO) Design* typically defined as a way of building the software systems architecture by combining the basic units (*classes*), where each unit "is a possibly partial implementation of some abstract *data type*, and may be connected to other classes by two relations: *client*, enabling the implementation of a *class* to rely on the facilities provided by another through its official interface, and multiple inheritance, where a class is defined as extension or specialization of one or more others." [13]. Only OO designed systems will be considered in this work.

## 2.4 Web Application

Under *Web Application* the following is understood in this research: *Web application* is a client-server application that uses a Web browser as its client program, and performs an interactive service by connecting with servers over the Internet (or Intranet). One has to differentiate between *web application* and *web site*. A *Web*

*site* just sends contents of static files, weather a *Web applicatio*n operates with dynamically created content, taking request parameters as an input, "tracked user behaviors, and security considerations" [24].

## 2.5 Web Application Framework

Knowing what is a web application, *Web Application Framework* has to be defined. *Web Application Framework* is a collection of tools (libraries) intended to reduce or avoid code redundancy and to maximize and promote code reuse [14]. It allows one to create web applications and services minimizing or completely eliminating low level tasks like management of processes and data transfer protocols. A particular web application framework will be chosen for analysis.

## 2.6 Model View Controller

Model View Controller (MVC) is software design pattern that has a tree layer structure (see Figure 1.1). The main goal of the pattern is to isolate business logic from the UI. The *model* typically represents the domain information. The view in turn is responsible for the model display on the UI. Finally, the *controller* using the users input, processes the model and updates the view respectively [8]. In this research MVC is considered as a basis for the subjected web application framework.

## 2.7 Loose Coupling

Loose coupling is a set of action patterns that are specific, or isolated from one another, but at the same time are aware of each other in some way [15]. In other words, loosely coupled components being logically and structurally independent relate to each other composing a solid system. Minimizing interdependence of such system components helps to derive issues in case of incorrect functioning and streamline development, testing and maintenance. Example of loosely coupled components may be the constituents of a *Model View Controller* paradigm based web application framework. Understanding the loosely coupled frame-

work components and the ways they are connected helps to identify the research target issue.

## 2.8 Version Control System

A *Version Control System* (VCS) is an information management system used to control and maintain the source of a project. The main idea is to store versions of artifacts (artifact revisions) making them accessible for later use [18]. Each artifact version is created by a *commit* of a *diff* (difference) between existing version and the changes that were made. There exist various implementations on of VCSs, but in this work the git[1]-based tool github.com[2] is considered.

## 2.9 Software Development Processes

*Software Development Process* is a collection of practices, technologies, methods and activities which are used by people and companies "to develop and to keep related software and products" [26]. This helps in identifying and studying the component consistency issue.

## 2.10 Software Testing

*Software Testing* is a set of activities intended to evaluate an aspect or effectiveness of a system and check if it meets the requirements [20]. Application testing approaches vary from simple manual checking test cases and comparing the outputs with expected results to complex automated testing tools that are able to simulate scenarios like high system load, different system configurations and user interaction. During the interview part of the research, software testing will be referenced as a part of generalized development recommendations to avoid the issue.

[1]https://git-scm.com/ accessed on 21.04.2016 at 10:50
[2]https://github.com/ accessed on 21.04.2016 at 10:58

## 2.11 Issue Tracking System

*Issue (Bug) Tracking System* is a (set of) application(s) that intended to organize maintenance and optionally development processes of software. In this research integrate github.com issue tracker has been employed (refer to *Version Control System* section). In the system an issue (bug) is typically represented by a *ticket* that in turn comprised of such fields as title, description and status of the issue (various various issue tracking systems may have additional fields and diverse statuses of the issue depending on workflow). This term is useful when considering data mining approaches to the issue identification.

## 2.12 Python

The author of the Python[3] programming language refers to is as "an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms" [28]. The Pythons *type system* is comprised of strong and dynamic typing.

## 2.13 Django

*Django*[4] is a web application framework written in Python, that implements MVC architectural pattern. It "provides high-level abstractions of common Web development patterns, shortcuts for frequent programming tasks, and clear conventions for how to solve problems" [9]. The Django framework-based application diagram is reflected on Figure 2.1.

[3]https://www.python.org/ accessed on 21.04.2016 at 10:59
[4]https://www.djangoproject.com/ accessed on 21.04.2016 at 11:50

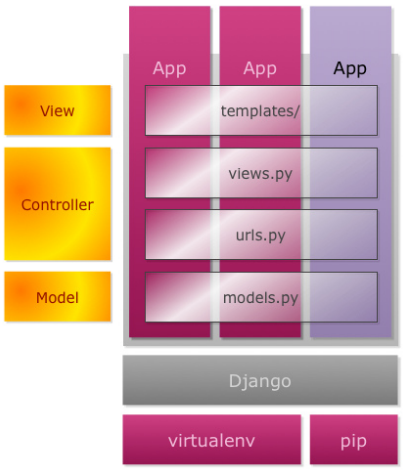


**Figure 2.1** Django-based web-application architecture (license CC BY 2.0)

# Chapter 3

# Related Work

This chapter presents related works that are relevant in scope of the Thesis research problems. The works have been taken from software evolution and software development artifacts theory.

Coupled software transformations (CX), are analized in [11]. The work describes relationship of software components and presents a number of patterns of coupling. The patterns, described using predicate logic, were proven to be applicable on practice, in particular, in "the context of testing implementations of CX".

The multi language software applications refactoring problem was presented in [21] in scope of the hibernate applications. The applied refactorings include object-oriented and database refactorings. The authors have shown that changes in one artifact can lead to "semantic-changing modifications in artifacts of other artifact types".

Co-evolution of Metamodels and Models is studied in [2]. The research presents an approach for managing (namely detecting and fixing) inconsistencies between metamodel and model, which is based on "incremental management of metamodel-based constraints".

Consistency management is researched in [12]. Namely, incremental consistency checking is presented in order to ensure "that models, metamodels, and their compositions conform to (i.e. meet all the constraints of) their metamodels and meta-metamodels".

A software repository mining approach was presented in [1]. The work gives insights on how such mining could be automated and applied on practice. An-

other research from the field of software repository mining [10] studies the relationships between various artifacts from a version control system. The approach is based on getting "highly frequent co-changing sets of artifacts" and establishing the traceability link between them based on such sets.

In [22] issue tracker mining is studied. The mining approach described in this work is comprised of information retrieval and natural language processing. Besides that, the issue tracker mining data was used in [7] to develop an approach of categorization and tracing software system features and hidden relationships between such features.

# Chapter 4

# Opportunities and Solutions

This section presents the research directions after discussing possible approaches to understanding, identifying and preventing the component consistency issue.

## 4.1 Understanding the Issue

In order to understand the component inconsistency issue one has to bear in mind the type system of web application framework implementation language. Having strongly enforced types imposes strict limitations on the program variables and their dependent structures. Weakly imposed types in turn provide developers with more flexibility and freedom when working with program components. Such benefits often come in conjunction with necessity to keep the program component relationships consistent. This means that all the corresponding type checking tasks normally performed by strongly typed languages on compile time have to be handled if required by the developer. In this case human error may cause the component inconsistencies that may slip in the version control system and potentially in the production deployment. This fact allows to assume that the research subject issue would be more common for frameworks based on languages without strict type checks. Among the great variety of such languages, the Python language has been chosen for the research. The language is used in a wide range of modern applications [6],[5], [30],[4], [3] and is flexible and easy to learn.

Having identified the language, a Python-based web application framework that will be used for the application component inconsistencies research, has to

be selected. The framework has to be a high level framework, in sense that it has to provide implementations for all underlying processes, allowing the developer to concentrate on matters directly involved in the application (like pages and services and not sockets and thread management). Python wiki[1] mentions three most popular high-level web frameworks:

- Django
- TurboGears[2]
- web2py[3]

After taking a closer look at each of the frameworks, the Django framework has been chosen as the one that corresponds to the research topic most. Django strictly follows the MVC paradigm having the clear separation between models listed in *models.py*, views represented by templates (typically **.html* files) and controller - *views.py* and *urls.py*. The architecture of Django based web application is depicted on Figure 2.1.

To understand the inconsistency issue and how it may arise, one has to know how the Django framework works and how its components are related. Consider the following example:

**Listing 4.1** Django Application Sample: models.py

```
1 from django.db import models
2
3 class Person(models.Model):
4     name = models.CharField(max_length=50)
5     dateOfBirth = models.DateField()
```

The *models.py* script describes the database tables by representing them as Python classes or *models*. Hence the name this component corresponds to the *Model* of MVC paradigm. Extending Django *models.Model* allows one to manipulate (create/read/update/delete) records in the database by writing exclusively Python code and not the related SQL statements. However, it is also possible to use native SQL if necessary.

[1] https://wiki.python.org/moin/WebFrameworks accessed on 23.04.2016 at 21:28
[2] http://www.turbogears.org/ accessed on 23.04.2016 at 19:55
[3] http://www.web2py.com/ accessed on 22.04.2016 at 16:30

**Listing 4.2** Django Application Sample: views.py

```
1 from django.shortcuts import render
2 from models import Person
3
4 def person_list(request):
5     persons = Person.objects.order_by('-name')
6     return render(request, 'persons.html', {'person_list':
          persons})
```

The *views.py* script includes the logic of request processing and compiling the view context. It corresponds to the *Controller* of the MVC. To be more precise, to a part of the Controller. The other part, *urls.py* is presented bellow.

**Listing 4.3** Django Application Sample: urls.py

```
1 from django.conf.urls.defaults import *
2 import views
3
4 urlpatterns = patterns('',
5     (r'^persons/$', views.person_list),
6 )
```

As it has been stated above, the urls.py is a part of MVC *Controller*, and is responsible for directing user requests to one or another *views.py* method. In this particular case the urls having /persons/ context will be directed to and processed by the *person_list* method of views.py.

**Listing 4.4** Django Application Sample: persons.html

```
1 <html>
2    <head>
3       <title>Persons</title>
4    </head>
5    <body>
6       <h1>Persons</h1>
7       <ul>
8       {% for person in person\_list %}
9          <li>{{ person.name }} - {{ person.dateOfBirth
               }}</li>
```

```
10          {% endfor %}
11          </ul>
12      </body>
13  </html>
```

Finally, the *persons.html*, representing the MVC *View* lists persons ordered by their name.

Now, consider the system component relationships. The relationships are not protected by type system of Python, so the *persons.html* view and *vews.py* controller would not be automatically aware of potential changes like field type change (field *name* may become a complex object of a type *Name* - affects *person.name* in the view and *-name* sorting in controller, see Listing 4.1), or just renaming (*name* may become *fullName*, affected components are the same as in the previous case) or removal (for instance *dateOfBirth* can be removed; the affected component is view with *person.dateOfBirth*) in type *Person*. Not taking into account such model changes in the view and controller will lead to minor reflection issues as well as to complete rendering attempt fails. Consequently, the maintenance of such relationships is the responsibility of developing party. Lack of the development process synchronization may lead to the component inconsistencies in case of changes similar to the ones described above.

**Listing 4.5** Modified class Person with a new class Name: models.py

```
1  from django.db import models
2
3  class Person(models.Model):
4      name = models.OneToOneField(
5          Name,
6          on_delete=models.CASCADE,
7          primary_key=True,
8      )
9      dateOfBirth = models.DateField()
10
11 class Name(models.Model):
12     firstName = models.CharField(max_length=50)
13     lastName = models.CharField(max_length=50)
```

## 4.2 Issue Identification

Having described the component inconsistency issue, the next step is to check weather it may indeed arise during a real development process and make its way to VCS and possibly to a deployment deliverable. This could be done in the following ways

- By mining VCSs and Bug Trackers of real projects
- By emulating a real development process
- By consulting people directly involved into the commercial development activities

### 4.2.1 Issue Tracker and Code Repository Data Mining

Mining Issue Tracker and Code Repository has been studied in number of works [1], [10], [7]. The approaches described in these works could be checked and adapted to the research problem. There is a huge number of Django-based applications publicly available at github.com that can be used for the study. Data from VCS and issue trackers can be processed with algorithms from the researches.

### 4.2.2 Emulating Development Process

Emulating the development process can be done with the following scenario:

1. introduce changes to the project version in scope of the research context
2. run test suite supplied with the project
3. verify if tests were successful

First point has to be done like it would have been a real task of adding a new feature of fixing a bug. The task for example can be formulated as follows: "*Add audit logging to the admin interface*" or "*Fix user list display issues*". The changes have to be of a "meaningful" nature to provide fair analysis.

For the second part it is important to make the updates to the test suite that corresponds to the introduced changes. The goal is to follow "an ideal" development process that typically requires the developer to cover every logical part of

the program he or she writes with tests. The idea is to check how probable is the problem occurrence.

Finally, if tests will be passed without braking on the introduced changes, would mean that the issue may be unnoticed and make its way to the VCS. Moreover, the test coverage of applications has to be obtained and checked for potential places and scenarios where/how it is possible for a problem to leak into VCS or production system unnoticed.

### 4.2.3 Gathering the Opinions

Another option is to contact people that are dealing with the Django-based application development on daily basis. Having the perspective of real developers would give important insights on the issue. Due to the fact that commercial application developers have tight schedule, the communication process has to be organized as efficient as possible. The most efficient way is to compile a questionnaire that would narrow down the discussion to scope of the research.

## 4.3 Issue Prevention

During the work on the issue identification step using one or another approach from the ones described above, one can derive a set of recommendations that could help to prevent the issue. Moreover, a prototypical method of highlighting potential issue spots to the developer has to be developed in order to enforce the obtained results.

## 4.4 Summary and Directions

From the great variety of the research directions the following were chosen to fit into the thesis constraints at the same time addressing the research questions:

1. Prepare and conduce a number of interviews with commercial Django application developers. The interviews has to have a form of face-to-face or skype communication, narrowed down to the research by the pre-compiled questionnaires. The interviews have to give the ouptuts that would help in addressing ***RQ1*** and lay grounds for ***RQ2***.

2. Develop a method of identifying potential spots of the component inconsistencies and check its efficiency on practice. This would address the ***RQ2***.

Rest of the research will be devoted to these two ways.

# Chapter 5

# Interviews

In order to get deeper insights to the specifics of the production software development it is necessary to consult people directly involved in the process. The primary goal is to check how common is the component inconsistency issue for commercial Django web application development processes. Besides that, it is necessary to analyze the methods and tools that are used or potentially can be used to prevent the inconsistency problems in these processes.

Section *Interview Methodology* sets the goals of this part of the research and describes how the interview questionnaires were prepared and compiled. This section also presents the theoretical points on how the interviews were conducted and analyzed along with the sample interview questionnaire.

In the following section *Interview Process* the interviews with the participants are described along with the strong and weak parts of the conversations.

The last section, *Interviews Output Discussion* presents in-depth analysis of data obtained during the interviews. It section also describes the essential elements required for this research.

## 5.1 Interview Methodology

In order to proceed with the interviews it is necessary to set goals and develop an approach which will be used to compile the questionnaires. First of all, the objectives have to be presented and essential matters that have to be taken into account in scope of the research highlighted. The participant choice, being one of the key factors of successful and versatile interview must be also outlined.

### 5.1.1 Goals and Theoretical Background

Primary goal of this part of the research is to get essential insights from people that are or were directly involved in the industrial Python, and in particular Django-based application development. Their experience would provide understanding on how relevant is the issue and is it possible for the issue to leak into CVS and / or in a deployment deliverable. The interviews have to be narrowed down to the research scope.

During the interview planning and questionnaire development process the following points have to be taken into account:

1. format of the interviews, including the interview duration and questionnaire volume
2. participant choice: skills/positions/availability of the participants
3. questionnaire/interview coverage in scope of the research goal
4. participant anonymity

The interview has to be planned as a conversation related to the issue. The conversation has to be narrowed down to the research scope. This can be achieved by compiling a set of questions that besides the underlining the issue scope would give the broad view on the problem and potentially discuss the actual or theoretical solutions. The conversation duration has to be no longer than 20-30 minutes due to potential tightness of the participants schedule. The questionnaires also have to have the limited number of questions, to make sure that the participants do not lose the interest in the middle of the questionnaire. Finally it is necessary to ensure the participant anonymity and explain how the interview outputs will be used.

### 5.1.2 Questionnaires

As a supplementary material for face-to-face and Skype interviews a questionnaire pdf form was compiled, The form could be easily filled and saved with any pdf reader, so that it is not necessary to create "hard" copies of the questionnaire for the participants. The questionnaire divided in three parts. First is the introductory statements which briefly set the problem context. The statements shall

also be provided with oral explanations and examples if necessary. The second part is the quiz-like question list with total of 7 multiple choice questions. Finally the discussion based part comprised of questions that can not be addressed by quizzes and require more extended answer. The form is supplied by text filed for remarks and notes.

**Introductory statement**

MSc. Thesis topic: Research consistency issues due to the evolution of a web application such that evolution of one component (e.g., the model in the MVC sense) may lead to inconsistency in another component (e.g., the view in the MVC sense)
Case study: Use of Django framework in popular open-source web applications:
- Classify the kind of changes that may be applied to model and may affect view
- Study how existing tests shield against inconsistencies between model and views
- Find evidence that inconsistencies could go unnoticed

**Questions**

1. You are/were employed as (position):
   ☐ junior dev
   ☐ senior dev
   ☐ team lead
   ☐ freelance
   ☐ other

2. Total experience with Django-based projects:
   ☐ less than 1 year
   ☐ 1-3 years
   ☐ 3-5 years
   ☐ more than 5 years

3. Scale of the application you are working (worked) on:
   ☐ small system, with basic functionality
   ☐ more-less sophisticated platform
   ☐ large enterprise solution
   ☐ other

4. With what components of the application you have to deal most of the time (multiple choice)?
   ☐ model
   ☐ view
   ☐ controller
   ☐ UI
   ☐ database
   ☐ other

5. What testing approaches are/were you using (regardless of the frameworks)?
   ☐ no test of any kind
   ☐ unit tests
   ☐ integration tests
   ☐ web-browser emulations
   ☐ QA team
   ☐ other

6. How would you estimate test coverage of the project you are working/worked on (multiple choice)?
   ☐ no coverage at all
   ☐ basic (some of the) functionality is covered (model)
   ☐ basic (some of the) functionality is covered (view)
   ☐ basic (some of the) functionality is covered (controller)
   ☐ essential use-cases are covered (model)
   ☐ essential use-cases are covered (view)
   ☐ essential use-cases are covered (controller)
   ☐ advanced functionality is covered (model)
   ☐ advanced functionality is covered (view)
   ☐ advanced functionality is covered (controller)
   ☐ other

7. How would you estimate relevance of the Django model/view consistency problem?
   ☐ largely irrelevant because inconsistencies are resolved routinely during coding
   ☐ largely irrelevant because inconsistencies are identified routinely by testing
   ☐ somewhat relevant as in the development approach may miss some inconsistencies

□ quite relevant as in serious inconsistencies could "easily" make it into production
**Discussion Part**

- What techniques do you employ to deal with the issue?
- What techniques could you imagine to be useful in this context? For instance:
- Does an IDE help (at least to some extent) to detect inconsistencies (I use Pycharm and seems that it does not really helpful in this situation)?
- Can you imagine a static or dynamic code analysis to detect inconsistencies?

## 5.2 Interview Process

The interviews were conducted with prior scheduling of either Skype or person to person meetings. In total five developers with various levels of skills, projects they are working (worked) on and methodologies on these projects were involved. At the beginning of the meeting each of the participant was informed about the privacy and interview outcomes usage issues. Namely, it has been told that the participant identity will not be disclosed and the interview outcomes will not be published without the respondents approval. Next an introductory statement was presented along with explanation and examples of the issue. After that the participant was asked to complete the quiz part of the questionnaire, providing brief comments on the particular questions. Finally, after the quiz questions were answered, the participant was asked to proceed to the discussion part and express the opinion on the questions presented in the section above. The main difficulty during the interviews was to hold the discussion in scope of the questionnaire to make sure it does not run out of defined time.

## 5.3 Interviews Output Discussion

The interviews were conducted with people on the following positions: Junior, Senior and Freelance developers. The experience level varied from 1-3 years to more than 5. The projects the people worked on range from small system to large

enterprise solutions for content management, multiple system monitoring and e-commerce domains. Overall, the experience of working with MVC components, database and UI was gathered. Next, unit tests, integration tests and QA were the most common techniques employed to the particular process. The corresponding test coverage ranged from essential to advanced coverage of MVC components. Most of the time respondents described the problem as somewhat relevant due to the inconsistencies during the development. However, the issue in most of cases was described as something that can be covered and detected by tests or even earlier, on the coding phase. Outputs of the quiz part are following:

1. *You are/were employed as (position)* - 3 junior dev, 3 senior, 2 freelance, 2 other
2. *Total experience with Django-based projects* - 2 1-3 years, 2 3-5 years, 1 more than 5 years
3. *Scale of the application you are working (worked) on* - 2 small system with basic functionality, 4 more-less sophisticated platform, 4 large enterprise solution
4. *With what components of the application you have to deal most of the time* - 4 model, 5 view, 3 controller, 1 UI, 1 database
5. *What testing approaches are/were you using (regardless of the frameworks)* - 5 unit tests, 2 integration tests, 2 web browser emulations, 3 QA team, 2 other (model fixtures)
6. *How would you estimate test coverage of the project you are working/worked on (multiple choice)* - 2 basic (some of the) functionality is covered (model), 1 basic (some of the) functionality is covered (view), 2 basic (some of the) functionality is covered (controller), 4 essential use-cases are covered (model), 3 essential use-cases are covered (view), 4 essential use-cases are covered (controller), 2 advanced functionality is covered (model), 1 advanced functionality is covered (view), advanced functionality is covered (controller)
7. *How would you estimate relevance of the Django model/view consistency problem* - 3 somewhat relevant as in the development approach may miss some inconsistencies, 2 largely irrelevant because inconsistencies are identified routinely by testing, 1 largely irrelevant because inconsistencies are resolved routinely during coding, 1 quite relevant as in serious inconsistencies could "easily" make it into production

During the discussion part people underlined that the main way of detecting and preventing problems (not particularly the research issue, but in general) is the test coverage, reliable communication within the team (or between the teams) and following process conventions both external and internal. Consecutively, the potential reasons for the research issue to appear were defined as deficiencies (or lack of) in one of these problem detection/prevention ways. Besides that, the participants underlined, that the issue is much more probable for mid and large scale projects, due to the fact that small projects are normally developed by smaller teams, and for a smaller team it is much easier to communicate and plan the development process in details. Contrarily, for mid-large scale projects, developed by large teams, the communication may be not that reliable and the development process may easier become out of sync.

Particularly, one of the respondents described a scenario in which the model-view relation inconsistency has made it to the projects qa deployment. The problem was caused by lack of the communication within the development team (more precisely due to vacation of one of the team members and inconsistent working schedule of another one) and not covered piece of the functionality. The inconsistency took place when the underlying query for obtaining a list of objects of type $A$ in a model method was modified in a way that it returns list of objects of type $B$. The view where this list was processes was not aware of the changes. This led to the broken UI that was detected only during manual tests procedures. The issue was fixed and the related piece of code was covered with tests.

Next, all the participants emphasized that IDE does not help in such scenarios due to the Django frameworks specifics described in *Identifying the Technology* section. The IDE used are Pycharm, Eclipse and Atom. As for the helper tool, most of the respondents underlined that it might be feasible only for mid-scale projects, due to the fact that for smaller project the issue is not much relevant, and for a larger scale ones the tool would have to process too many project files, which can cause serious development process performance issues. As has been told by one of the respondents, such processing would have to take place dynamically due to the fact that static analysis in case of Django would not give any guarantees regarding the types engaged in the model-view relationship.

The interviews have provided valuable information on practices involved in the commercial application development and confirmed that the issue can indeed take place due to the development process out-of-sync.

## 5.4 Analysis and Summary

Having obtained the outputs of the open source application development analysis and interviews with the commercial application developers, the final step of this part of the research is to summarize the information and derive a number of recommendations and best practices for issue detection and prevention.

In order to minimize or even completely eliminate Django component inconsistencies, the following techniques have to be used:

- Tesing: ensure both quality and quantity
- Following conventions and standards
- Employing development methodology

### 5.4.1 Testing

First of all testing. The analysis of the open source applications has shown that due to the lack of testing scenarios and coverage an obvious inconsistencies could potentially be unnoticed. This argument was supported by the commercial application developers. That is why it is important to make sure that at each stage of the development process the application functionality coverage is or close to 100%. This includes not only coverage percentage, but also quality of tests. All possible scenarios have to be considered in order to make sure that any potential problem is noticed and fixed in time. The core testing approach recommendations, taken from both open source and commercial development experiences are the following:

- Unit tests
- Integration tests
- Client browser emulation
- QA team

### 5.4.2 Standards

From the python developer interviews another aspect of minimizing/eliminating issue was derived. Namely, following the development standards. It means that the developer writes and organizes the code according to conventions. In case of Python it could be PEP8[1]. Wile working with Django one has to make sure that the MVC methodology is strictly followed. Such conventions not only help in fast identification and prevention of the problem but, as it has been pointed out during one of the interviews, also would significantly speed-up the process of understanding the application be new-commer developers.

### 5.4.3 Methodologies

From the commercial development experience it has become clear that the clearly organized development process significantly reduces the probability of the component inconsistencies. Particularly, the following techniques are (and can be) engaged:

- Daily stand-ups, where each team member describes what he has done / is doing / going to do
- So called 4EP (4 [E]ye [P]erson) checks, when before each commit the code is reviewed by the developer minimum one another person from the team
- Senior code review, when the code written by less skilled developers is reviewed by senior developers before it goes to the version control system
- Testing phase, where the artifact is deployed to the qa servers and the corresponding QA activities are performed

[1]https://www.python.org/dev/peps/pep-0008/ accessed on 23.04.2016 at 21:58

# Chapter 6

# Prototype Method

The developed method is represented by a prototypical code analysis tool that is able to highlight loosely coupled component relationships helping to detect and eliminate the inconsistencies. More precisely, the tool maps the model-view relationships of a Django-based application engaged in the test runtime and pinpoints potential inconsistency spots. A potential inconsistency spot in turn, is a relationship between model and view (in case of Django model classes and properties used in template) that is not exercised at test runtime. This means that the relationship may easily be broken in case of out-of-sync development process, and potentially leak in VCS as well as to a deployment deliverable. Notifying the developer about such relationships stimulates verifying them, fixing if necessary and covering by the corresponding tests.

## 6.1 Requirements

The following Table 6.1 reflects the basic requirements for the prototype tool

| ID | Title | Description | Priority |
|---|---|---|---|
| REQ1 | Model / View Links | The system has to analyze models that are used in views and infer the model Classes | Must |
| REQ2 | Template variable model mapping | The system has to process all the application templates and build the template variable listings | Must |
| REQ3 | Result intersection | The system has to find intersection between the context variable mapping and template variable lists | Must |
| REQ4 | Command line visualization | The tool must provide basic result visualization functionality. The command-line output has to reflect the model-view relationships in form of a table. | Must |
| REQ5 | Plugability | The tool can be implemented as a plugin for existing instruments (such for example as python coverage.py project) | Can |

## 6.2 Discussion and Challenges

Even though getting the variable names and properties they are referring to is a trivial task, the most problematic part is determining the variable types directly from the template. It is not possible to say looking just at the variable name and properties it is referring to that the variable is of a concrete model type. Given this fact a static analysis mapper is not feasible in context of the problem. To overcome this challenge the mappings can be obtained dynamically at runtime, since the template rendering context can be intercepted and its variables along with their type can be identified and mapped. This would make the tool dependant on the concrete runtime in a way that the mapping is built only for the templates directly involved in the runtime. So for instance if the application is running in the test mode, the mapping building completely relies on the test coverage. That means

that is a certain template is not engaged in the testing activities, it will be not possible to map its variables. However, having a template not mapped during such test execution phase can be detected during static analysis. Then, compared to the dynamic mapping it is trivial to see which templates are not covered and prompt the developer to do so.

## 6.3 Design

The prototype application itself is a *fork* of Django Template Coverage plugin[1] for Python Coverage tool (coverage.py)[2]. Coverage.py is an instrument for measuring and analyzing the Python project code coverage. It interceprts the code execution and records statistics of involved code snipets invocation. After the program execution gathered statistics can be compared to actual source code and the difference (executed lines vs actual lines) is presented. The Coverage tool phases are reflected in the Table 6.3[3].

| Phase | Name | Description |
|---|---|---|
| 1 | Execution | Coverage.py runs the code, and monitors it to see what lines were executed |
| 2 | Analysis | Coverage.py examines the code to determine what lines could have run |
| 3 | Reporting | Coverage.py combines the results of execution and analysis to produce a coverage number and an indication of missing execution |

Moreover, the tool provides an ability to trace differences between the executions, so that onw can see how the project coverage evolves (degrades) from one

[1]https://github.com/nedbat/django_coverage_plugin accessed on 26.04.2016 at 10:30

[2]https://github.com/msabramo/coverage.py accessed on 26.04.2016 at 10:31

[3]According to http://coverage.readthedocs.org/en/coverage-4.0.3/howitworks.html accessed on 26.04.2016 at 11:33

coverage launch to another. The cycled representation of the coverage phases is presented on the Figure 6.1

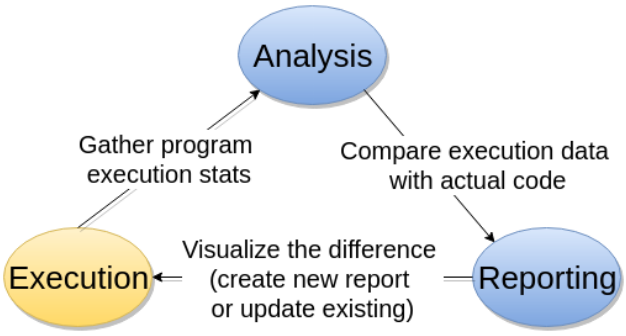


**Figure 6.1** Coverage.py cycle flow. The blue circles mark the phases that are engaged in a single *tool* execution (analysis and report are performed one after next in a single *tool* run). The yellow indicates the execution of a *program* with the *tool* (data gathering)

Coverage tool provides interfaces for developing and integrating helper plugins. At the moment the only known plugin implementations are Django template coverage and *Mako* template coverage[4].

### 6.3.1 Coverage Plugin API

The coverage.py plugin API consists of three classes: *coverage.CoveragePlugin*, *coverage.FileTracer* and *coverage.FileReporter*. The first is a base class for coverage plugins. It has to be extended in order to make it possible to integrate the plugin into coverage.py. The second includes support functionality needed on the Execution phase, providing interface methods for file processing. Finally, the *FileReporter* is for processing the actual source code files and comparing it to the data obtained on the *Execution* step on the *Analysis* and *Reporting* phases respectively.

### 6.3.2 Django Template Coverage

The Django template coverage plugin had grown from a standalone project *dtcov*[5] and was integrated into coverage.py as a plugin. The main idea is by implementing *FileTracer* interface for non-python files (particularly *.html) the plugin can

[4]https://bitbucket.org/ned/coverage-mako-plugin accessed on 26.04.2016 at 11:50
[5]https://github.com/traff/dtcov accessed on 26.04.2016 at 11:50

gather the data from Django templates when they render. Consecutively, the *FileReporter* is adapted to deal with the template files on the *Analysis* phase so that the *Report* phase includes the calculated template coverage into the coverage reports.

### 6.3.3 Tool Design

The solution for the challenge of mapping the model classes to a particular template is to intercept the template context objects and map them to the particular template at runtime. Having the coverage.py plugin API it is natural to integrate a model-template mapper into the *Execution* phase. Then, on the *Analysis* and *Reporting* phases gather the data from the actual application templates and build the diff. Having almost all required implementation in Django Temaplte coverage plugin, it is more advantageous to create a *fork* of the plugin, so that is includes the model-view mappings and the related source code differences into the coverage reports than to build a new artifact from scratch.

The base principle of the tool is reflected on Figure 6.2.

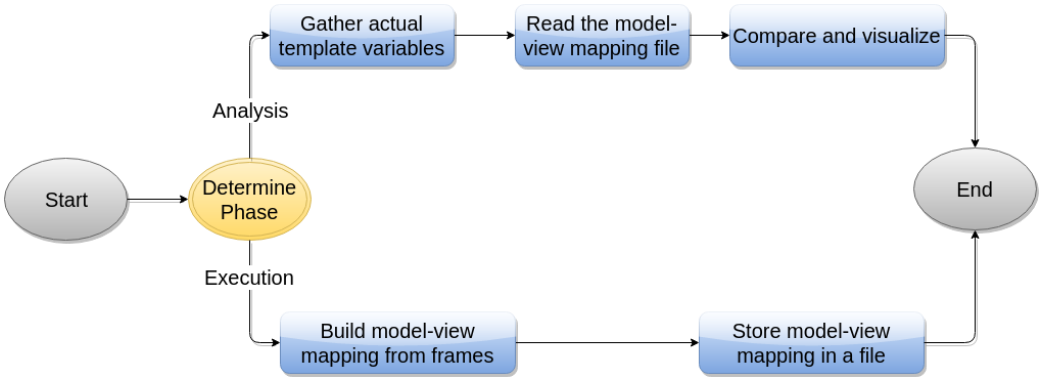


**Figure 6.2** Code analyzer plugin prototype functioning principle

## 6.4 Implementation

As it has been stated before, the prototype of code analysis tool, being a fork for the coverage.py plugin, was implemented in a way that all three phases of coverage.py are involved in the information processing. That means that the Django template coverage plugin was updated with new functionality in tree

places. Namely: the *DjangoTemplatePlugin* that is responsible for model-template mapping on the coverage *Execution* phase, *FileReporter* class which takes care of the actual template variables mapping on the coverage *Analysis* phase and finally a new method that performs the data comparison and visualization was added. The method executes after the *Reporting* coverage phase, right before the program exits.

### 6.4.1 Execution Phase

The code involved in the Execution phase was integrated into the *DjangoTemplatePlugin* class as *dump_context(self, frame)* method. This method is called from *dynamic_source_filename* and *line_number_range* methods. *dump_context* method performs frame processing and fills the template-model mapping. A simplified algorithm of the *dump_context* method is reflected on Figure 6.3

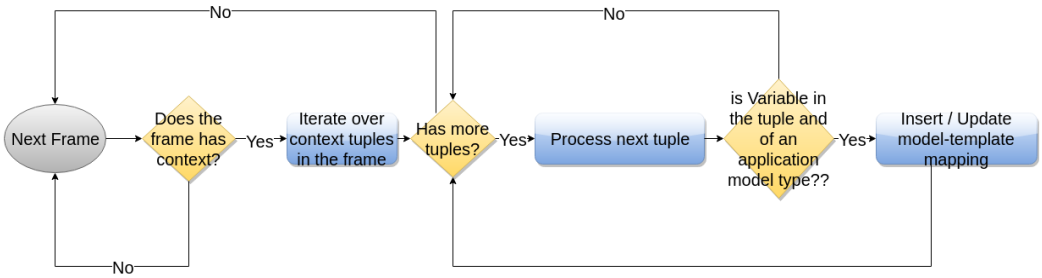


**Figure 6.3** Algorithm of processing a coverage frame and filling the model-template mapping

The template-model mapping is represented by a dictionary object that has absolute template file names as the keys. For each such key there is a set of model class names that are resolved from the frame context. Each model in turn is mapped to set of its variable names that are involved in actual rendering of the particular template. The mapping structure is presented on Figure 6.4.

At the final stage of the *Execution* phase the mapping is stored in a file for usage in the *Analysis* and *Report* coverage.py stages.

### 6.4.2 Analysis Phase

The *Analysis Phase* which employs the *FileReporter* class implementation, is executed in two stages. First of all, the actual application templates are parsed and

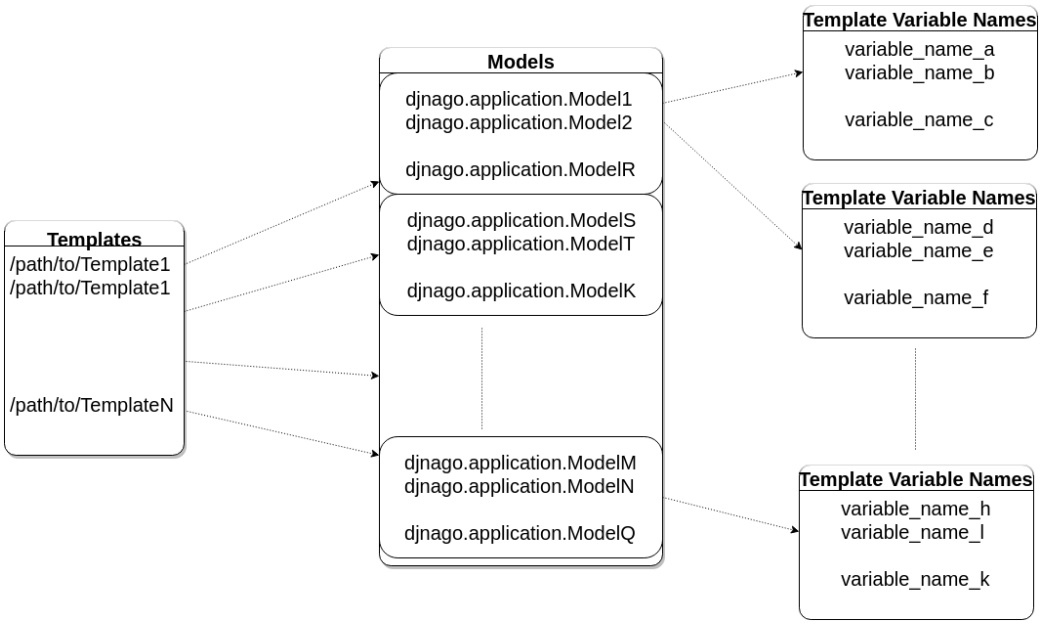


**Figure 6.4** Template-Model mapping

the template-variable mapping is created in the updated Django template coverage plugin *FileReporter* class *lines* method. The mapping consists of template file absolute paths as a keyset and sets of variable names used for particular template for each key. The mapping is shown on Figure 6.5. Next stage firstly loads the template-model mapping, obtained on the previous stage. If *Execution* has never taken place, or if the template-model mapping file is missing, the system informs the user that the visualization of the model-template relations is not possible and exits. If the file contents were successfully loaded, the analysis proceeds further building the related sets intersections.

### 6.4.3 Report Phase

On the report phase the created mappings are intersected in order to pinpoint model attributes that were not exercised during the tests. Having these relationships not covered by tests means that the component consistency may be broken without automatic tests noticing it. The report itself is represented by a console output which includes:

1. Template / model mapping (from runtime contexts)

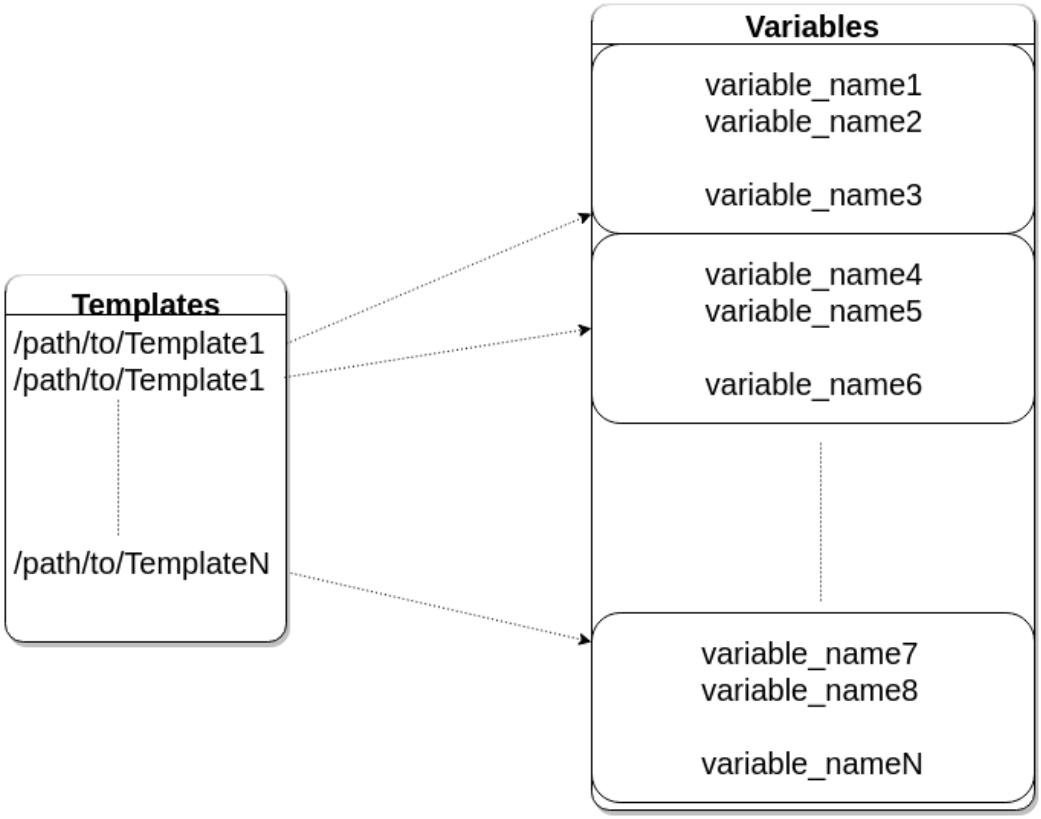


**Figure 6.5** Template-Variable mapping

2. Template / token mapping (actual)

3. Intersection of mapping variables

The first two mappings obtained on test runtime and from the actual Django views (template htmls) data respectively. The last one illustrates what properties were not exercised by the tests and hence are vulnerable to the consistency issue.

## 6.5 Summary

The developed method is intended to show the developer which and how Django model classes are used in the view (template) and in case of potential inconsistencies produce the corresponding notifications. The mapping can be built only at runtime, due to the fact that the static analysis does not provide necessary information on template variable types. This means that the developer has to make

sure that all templates are rendered during the tests. This, as a side effect, helps to promote a wider template coverage by showing the developer which templates are not engaged in the testing.

Django version supported at the moment is 1.9. To provide the support for lesser versions is a subject for future work. Besides that, due to the fact that Django template engine does not track conditional code in the templates the tool can not check how the variable was engaged.

# Chapter 7

# Case Study

This chapter describes the case study performed using the developed method on the real project from github.com. After the goals are stated, the project is selected and described, the method is applied and validated. Finally the findings are summarized and discussed.

## 7.1 Goals

The goals of the case study are the following:

By applying the method developed in the previous chapter:

- Establish the connection between model and view (in other words identify which classes are engaged in which views)
- Identify potential component inconsistency spots by examining the model-view relationships
- Validate the results by examining actual application and how the method work with "inconsistencies" injected for test purposes

## 7.2 Available Applications

A number of projects from github.com that use Django framework were considered for analysis. Two main criteria were used to select a project on this stage of the research. First of all, the popularity on github.com. The popularity metric

consists of the amount of github.com "stars" and recent activity timestamp. For all selected projects the amount of "stars" has to be more than 1000, and the recent activity timestamp not more than a month in the past (to the time of the research). Next criterion is the ability of the projects test suite to execute without extra code changes. To make the selection process more productive only the projects whose test suite can be executed without errors "out-of-the-box" with minimum to no additional setup steps were selected.

The following Table 7.2 reflects the applications chosen for analysis along with the star amount, activity timestamps and short descriptions.

| Name | Stars | Last Activity | Description |
|---|---|---|---|
| Django ReST Framework | 5568 | 24.03.16 | a powerful set of tools that help to easily and effectively build web APIs [6]. Providing authentication/authorization, serialization, extensive documentation and ease of use Django ReST framework has strong community support and trust of such companies as Mozilla and Eventbrite. |
| Django Oscar | 1918 | 24.03.16 | an open-source e-commerce framework for Django [5]. The framework provides a wide range of tools and plugins for creating and managing e-commerce portal. |
| Wagtail CMS | 3358 | 25.03.16 | a powerful Content Management System framework intended to make the content creation a pleasant experience [30] |
| Django CMS | 4063 | 25.03.16 | Another Content Management System, *django-cms* is a simple but powerful open-source tool "created by developers for developers" [4] |
| Django Activity Stream | 1003 | 06.03.16 | a library that allows one to track user activity on a web site. The library can easily be integrated to any django-based application [3] |

## 7.3 Django Activity Stream

Among the project above the *Django Activity Stream* project appears to be the best candidate for the research. Having a clear structure and separation of concerns, the project is one of the most "generic" from the all considered applications.

### 7.3.1 Dependencies

The project has the following dependencies:

- Django>=1.6,<1.7
- Sphinx==1.2.2
- git+https://github.com/justquick/alabaster.gitegg=alabaster
- django-jsonfield==0.9.12

Besides that it is required to have Python 3+ installed.

### 7.3.2 Structure

Django Activity Stream project consists, apart from the configuration files and migration data, of 2 parts: main application, or the library itself and test application, that is used in the tests. The test application is a simple Django project that uses the library by extending model classes and implementing the views (see next chapter). The library Django Activity Stream further in the research is referred to as the main application (main app), and its test application - test app.

### 7.3.3 Statistic

Main app and test app follow the Django framework specifics and accordingly MVC paradigm. They both have clearly separated mode, view and controller components. Model of main app to the moment of the research consists of the following classes:

- Follow
- Action
- User*
- ContentType*

The classes defined in main apps model.py file. Classes marked with [*] are Django framework classes used in the main app. Test app model in turn additionally extends User class (MyUser).

The views, or templates in terms of Django, are comprised of:

- action.html
- actor.html
- base.html
- detail.html
- followers.html
- following.html

## 7.4 Preconditions

Before operating with the Django Activity Stream application, that can be checked out from github.com, the tool has to be configured. The detailed configuration and setup steps are described in the following chapters.

### 7.4.1 Setup

Having forked plugin.py and coverage.py and the original Django Template Coverage plugin installed on the application Python virtual environment, the original Django template plugin file was replaced with the forked one:

**Listing 7.1** Copying the forked plugin

```
1 $cp /path/to/forked/plugin.py
      /.../lib/python3.4/site-packages\
2     /django_coverage_plugin/plugin.py
```

### 7.4.2 Configuration

Since the prototype is a fork for Django template plugin, all the related configuration[1] and the ways of the configuration remain relevant. At this stage of the prototype no extra configuration options were added.

[1]https://pypi.python.org/pypi/django_coverage_plugin accessed on 26.04.2016 at 10:35

**Listing 7.2** Prototype code analysis tool setup for django-activity-stream project

```
1   $cd django-activity-stream && echo "[run]
2 plugins = django_coverage_plugin" >> .coveragerc
```

### 7.4.3 Artificial Changes

In order to check how the method deals with a theoretical model-view inconsistency, a small change was made in action.html template of the main application. Namely a string " action.verb3 " was added. From Djangos point of view this means that the template action.html have to render non-existing "verb3" property of the "actor" variable which in this particular case is of type User. In other words, the view actor.html has lost its consistency with the model class "User" that does not have "verb3" field or method.

## 7.5 Application

As it has been mentioned in the chapter above, the developed prototype tool has two phases, namely: execution phase and report phase. Accordingly, the tool operates in two modes: execution and report.

### 7.5.1 Launch - Execution Stage

The execution process is analogous to the coverage.py execution with a plugin[2]. The following command is used to launch the tool in the gathering mode.

**Listing 7.3** Prototype code analysis tool excecution for django-activity-stream project

```
1 $DJANGO_SETTINGS_MODULE=actstream.runtests.settings
    coverage run actstream/runtests/manage.py test
```

### 7.5.2 Launch - Report Stage

In the same way, the report stage mode of the tool is activated via the following command:

[2]https://coverage.readthedocs.io/en/coverage-4.0.3/plugins.html accessed on 26.04.2016 at 10:35

**Listing 7.4** Prototype code analysis tool excecution for django-activity-stream project

```
1 $DJANGO_SETTINGS_MODULE=actstream.runtests.settings
    coverage report
```

## 7.6 Outputs

The tool outputs, besides tha standard coverage.py plugin outputs consist of the template-model mapping with models involved in each template, template-token mapping, covered/not covered templates and covered / not covered properties for each template. The related output extract of the plugin is reflected on the Listing 7.7 (long paths are omitted).

**Listing 7.5** Prototype code analysis tool output extract for django-activity-stream project

```
1  ========= Template / model mapping (from runtime contexts)
     ==============
2  /.../../actstream/templates/action.html:
3      actstream.runtests.testapp.models.MyUser:
4          actor
5      actstream.models.Action:
6          action
7          action.actor.get_absolute_url
8          action.actor
9          action.actor_url
10         action.target.get_absolute_url
11         action.target
12         action.target_url
13     django.contrib.contenttypes.models.ContentType:
14         ctype
15 ...
16 ========= End Template / model mapping (from runtime
     contexts) ============
17 ========= Template / token mapping (actual) ==============
18 /.../.../actstream/action.html:
19     action.actor.get_absolute_url
```

```
    action.actor
    action.actor_url
    action.actor
    action.verb
    action.verb3
    action.action_object.get_absolute_url
    action.action_object
    action.action_object_url
    action.action_object
    action.target.get_absolute_url
    action.target
    action.target_url
    action.target
    action.timestamp|timesince
...
========= Intersection of variables from mappings
    ==============
/.../.../actstream/action.html
    Covered:
        ['action.actor_url',
        'action.target',
        'action.target.get_absolute_url',
        'action.target_url',
        'action.actor.get_absolute_url',
        'action.actor']
    Not Covered:
        ['action.action_object_url',
        'action.timestamp|timesince',
        'action.action_object',
        'action.verb3',
        'action.action_object.get_absolute_url',
        'action.verb']
...
```

## 7.7 Validation

As it can be seen from the output during the execution the following three classes were employed in rendering of *action.html* template: MyUser, Action and ContentType (lines 3,5 and 13). Moreover, it can be seen how variables of each type were referenced in the template. For instance, variable *action* was of type Action and the following properties were exercised, lines 5-12 (after the variable name, after the first dot symbol):

- action.actor.get_absolute_url
- action.actor
- action.actor_url
- action.target.get_absolute_url
- action.target
- action.target_url

Besides that, the listing contains *action* variable itself (line 6), which means that the variable was engaged in a conditional operator which Django template engine does not track. It is worth to show such variable as an "uncovered" to inform the developer anyway, so that the fact is not left unnoticed.

Next, one can see the contents of template-token mapping (lines 18-33). This mapping, built on the Analysis phase, reflects which variables are actually used in the template, including one with the invalid property *action.verb3* (line 24) which was added for the testing purposes.

Finally, there is an intersection of values that were employed at the template test runtime and the actual template variables (lines 36-50). One can see which properties were used during the tests and appear on the template, and which were not. This should encourage the developer to check the potential inconsistency spots and ideally cover them with tests.

Removing the introduced changes (*action.verb3*) and executing the plugin again produces the following result:

**Listing 7.6** Prototype code analysis tool output extract for django-activity-stream project with removed inconsistency

```
1  ========= Template / model mapping (from runtime contexts)
       ==============
2  /.../../actstream/templates/action.html:
3      actstream.runtests.testapp.models.MyUser:
4          actor
5      actstream.models.Action:
6          action
7          action.actor.get_absolute_url
8          action.actor
9          action.actor_url
10         action.target.get_absolute_url
11         action.target
12         action.target_url
13     django.contrib.contenttypes.models.ContentType:
14         ctype
15 ...
16 ========= End Template / model mapping (from runtime
       contexts) ============
17 ========= Template / token mapping (actual) ==============
18 /.../.../actstream/action.html:
19     action.actor.get_absolute_url
20     action.actor
21     action.actor_url
22     action.actor
23     action.verb
24     action.action_object.get_absolute_url
25     action.action_object
26     action.action_object_url
27     action.action_object
28     action.target.get_absolute_url
29     action.target
30     action.target_url
31     action.target
32     action.timestamp|timesince
33 ...
```

```
34 ========= Intersection of variables from mappings
       ==============
35 /.../.../actstream/action.html
36     Covered:
37         ['action.actor_url',
38         'action.target',
39         'action.target.get_absolute_url',
40         'action.target_url',
41         'action.actor.get_absolute_url',
42         'action.actor']
43     Not Covered:
44         ['action.action_object_url',
45         'action.timestamp|timesince',
46         'action.action_object',
47         'action.action_object.get_absolute_url',
48         'action.verb']
49 ...
```

As can be seen the inconsistent relation *action.verb3* does not appear in the output anymore.

Finally, it was manually checked that the outputs are correlated with the actual application components. The check gave positive result and confirmed that the tool is indeed able to pinpoint loosely coupled component relationships, namely the model and view and help in detecting and preventing the issue.

## 7.8 Summary

The case study has shown that using the development method one can identify potential spots of the component inconsistencies using the model-view mapping created by the tool. The tool highlights relationships that were not exercised by the application tests. Examining these highlights helps one to recognize the inconsistency, like in the case of "verb3" of the Action class. The outputs produced by the tool and their manual validation have proved that the method works fairly on the real life project and can potentially be used during the application development.

# Chapter 8

# Conclusions and Future Work

During the research the component inconsistencies were highlighted and studied. Obtaining the perspective of commercial Django application developers allowed to confirm that the issue is indeed relevant and can make its way up to deployment deliverables, particularly QA. This perspective has not only helped in addressing ***RQ1***, but also in deriving common recommendations to people involved in the development processes. These processes, involving multi-faceted testing, following standards and methodologies, have to be clearly organized to prevent out-of-sync scenarios which lead to the issue. Further addressing ***RQ2***, these recommendations were enforced by the method, that allow to pinpoint model-view relationships of Django-based application. The method was developed and applied on Django Activity Stream project as a case study. The case study has shown that the method builds correct model-view mapping and highlights the model-view relationships that are vulnerable to the component consistency issue.

The component inconsistency requires multi-sided research. It is necessary to conduct a research involving repository, issue tracking mining and emulating a real development process. Besides that it would be helpful to conduct more interviews with Django application developers, which would allow to obtain more opinions for analysis. The methodology developed for this research could potentially be used for such analysis. Also, the prototypical method developed for this thesis requires improvements like multiple Django version support, multi-threaded runtime support, additional visualization ways and possibly integration into an IDE.

# Bibliography


[1] P. Anbalagan and M. Vouk. "On mining data across software repositories". In: *2009 6th IEEE International Working Conference on Mining Software Repositories*. May 2009, pp. 171–174. DOI: 10.1109/MSR.2009.5069498.

[2] Andreas Demuth et al. "Co-evolution of metamodels and models through consistent change propagation". In: *Journal of Systems and Software* 111 (2016), pp. 281–297. DOI: 10.1016/j.jss.2015.03.003. URL: http://dx.doi.org/10.1016/j.jss.2015.03.003.

[3] *Django Activity Stream*. URL: https://github.com/justquick/django-activity-stream (visited on 04/23/2016).

[4] *Django CMS*. URL: http://www.django-cms.org/ (visited on 04/23/2016).

[5] *Django Oscar*. URL: http://oscarcommerce.com/ (visited on 04/23/2016).

[6] *Django ReST Framework*. URL: http://www.django-rest-framework.org/ (visited on 04/23/2016).

[7] Michael Fischer, Martin Pinzger, and Harald C. Gall. "Analyzing and Relating Bug Report Data for Feature Tracking". In: *10th Working Conference on Reverse Engineering, WCRE 2003, Victoria, Canada, November 13-16, 2003*. 2003, pp. 90–101. DOI: 10.1109/WCRE.2003.1287240. URL: http://dx.doi.org/10.1109/WCRE.2003.1287240.

[8] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321127420.

[9] A HOLOVATY and J KAPLAN-MOSS. "The Django Book: Version 2.0". In: *The Django Book* 16 (2009).

[10] Huzefa H. Kagdi, Jonathan I. Maletic, and Bonita Sharif. "Mining Software Repositories for Traceability Links". In: *15th International Conference on Program Comprehension (ICPC 2007), June 26-29, 2007, Banff, Alberta, Canada*. 2007, pp. 145–154. DOI: 10.1109/ICPC.2007.28. URL: http://dx.doi.org/10.1109/ICPC.2007.28.

[11] Ralf Lämmel. "Coupled Software Transformations—Revisited". Under submission. Published online 6 March 2016.

[12] Roberto E. Lopez-Herrejon et al. "Using Incremental Consistency Management for Conformance Checking in Feature-Oriented Model-Driven Engineering". In: *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*. 2010, pp. 93–100. URL: http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf.

[13] Bertrand Meyer. *Object-Oriented Software Construction*. 1st. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN: 0136290493.

[14] Rimjhim Mishra and Pankaj Kumar. "Web Application Framework: Review". In: ().

[15] J Douglas Orton and Karl E Weick. "Loosely coupled systems: A reconceptualization". In: *Academy of management review* 15.2 (1990), pp. 203–223.

[16] John K Ousterhout. "Scripting: Higher level programming for the 21st century". In: *Computer* 31.3 (1998), pp. 23–30.

[17] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8.

[18] J. Pinnix et al. *Version control system*. US Patent App. 11/107,145. Oct. 2006. URL: https://www.google.com/patents/US20060236319.

[19] Trygve Reenskaug and James Coplien. "The DCI Architecture: A New Vision of Object-Oriented Programming". In: *artima.com* (2009).

[20] Matthew N. O. Sadiku and Mohammad Ilyas. *Simulation of Local Area Networks*. Boca Raton, FL, USA: CRC Press, Inc., 1995. ISBN: 0849324734.

[21] Hagen Schink et al. "Hurdles in Multi-language Refactoring of Hibernate Applications". In: *ICSOFT 2011 - Proceedings of the 6th International Conference on Software and Data Technologies, Volume 2, Seville, Spain, 18-21 July, 2011*. 2011, pp. 129–134.

[22] P. Schugerl, J. Rilling, and P. Charland. "Mining Bug Repositories–A Quality Assessment". In: *Computational Intelligence for Modelling Control Automation, 2008 International Conference on*. Dec. 2008, pp. 1105–1110. DOI: 10.1109/CIMCA.2008.63.

[23] Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithms Analysis, Java Edition*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998. ISBN: 0-13-660911-2.

[24] Leon Shklar and Rich Rosen. *Web Application Architecture: Principles, Protocols and Practices*. 2nd. Wiley Publishing, 2009. ISBN: 047051860X, 9780470518601.

[25] Jeremy G. Siek and Walid Taha. "Gradual Typing for Functional Languages". In: *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*. 2006, pp. 81–92.

[26] Ian Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004. ISBN: 0321210263.

[27] Nicola Tomatis et al. ""May you have a Strong (-typed) Foundation" why Strong-typed Programming Languages do Matter". In: *Proceedings of the 2004 IEEE International Conference on Robotics and Automation, ICRA 2004, April 26 - May 1, 2004, New Orleans, LA, USA*.

2004, pp. 3429–3434. DOI: 10.1109/ROBOT.2004.1308784. URL: http://dx.doi.org/10.1109/ROBOT.2004.1308784.

[28] Guido Van Rossum and Fred L. Drake. *An Introduction to Python*. Network Theory Ltd., 2011. ISBN: 1906966133, 9781906966133.

[29] Michael M. Vitousek et al. "Design and evaluation of gradual typing for python". In: *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SLASH 2014, Portland, OR, USA, October 20-24, 2014*. 2014, pp. 45–56. DOI: 10.1145/2661088.2661101. URL: http://doi.acm.org/10.1145/2661088.2661101.

[30] *Wagtail CMS*. URL: https://wagtail.io/ (visited on 04/23/2016).