****

CS673S15 Software Engineering

Group 1 Team 3 Project

Software Design Document – Issue Tracker

**Team**

|  |  |  |  |
| --- | --- | --- | --- |
| **Team Member** | **Role** | **Signature** | **Date** |
| Joe Driver | Team Leader/QA Leader | Joe | Feb-15-2015 |
| Ya-Lan Tsao (Amy) | Requirements Leader | Amy | Feb-15-2015 |
| Lin-Kei Tseng (Ted) | Design Leader | Ted | Feb-15-2015 |
| Chun-Kai Huang (Kenny) | Implementation Leader | Kenny | Feb-15-2015 |
| Jerrold Ansman | Group Leader |  | Feb-15-2015 |
| Samer Abu-Nasser | Environment and Integration Leader | Samer | Feb-15-2015 |
| Joshua Darrieulat | Configuration Leader | Josh | Feb-15-2015 |

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Author** | **Date** | **Change** |
| 1.0 | Joe Driver | Feb-15-2015 | Kick-off |
| 2.0 | Joe Driver | April-2-2015 | Iteration 2 |
| 3.0 | Joe Driver | April-21-2015 | Iteration 3 |
| 4.0 | Samer Abu-Nasser | May-3-2015 | Final |

Contents

[Introduction 3](#_Toc418504620)

[Software Architecture 3](#_Toc418504621)

[Requirements Design 3](#_Toc418504622)

[Sequence Diagram 3](#_Toc418504623)

[Use Case Diagram 4](#_Toc418504624)

[State Diagram 5](#_Toc418504625)

[System Design 6](#_Toc418504626)

[Design Patterns & Principles 7](#_Toc418504627)

[Singleton pattern & DRY principle 7](#_Toc418504628)

[Factory Method Pattern 7](#_Toc418504629)

[Model View Controller (MVC) Pattern 7](#_Toc418504630)

[Key Algorithms 8](#_Toc418504631)

[Classes and Methods 9](#_Toc418504632)

[Model Class (Mapper Package) 9](#_Toc418504633)

[Views (Domain Package) 10](#_Toc418504634)

[Django Views 10](#_Toc418504635)

[Django Forms 11](#_Toc418504636)

[Testing Design 11](#_Toc418504637)

[Test Modules 11](#_Toc418504638)

[References 12](#_Toc418504639)

[Glossary 12](#_Toc418504640)

# Introduction

Group 1 of CS673S15 has been assigned to create a project management tool on the web named 3blueprints (<www.3blueprints.com>). The purpose of this document is to outline the Software designs of the Issue Tracker Application done by Team 3. The systems is an issue and bug tracker. It is designed to work as part of the larger project management suite. Our primary goals are interface design usability, collaboration simplicity and communication efficacy.

# Software Architecture

The web tool will be built using the Django framework. Django is a python based MTV (a variation of the classical MVC – described in depth later in the document) web framework built to simplify Web app creation through common default tools, modules and plugins. Furthermore, the team will use Bootstrap for the Front-end. Bootstrap is an HTML, CSS and JS framework which provide pre-built features to assist in the development of Mobile-First (Progressive Enhancement) and Responsive Web Interfaces.

## Requirements Design

The client has requested a project management web application. The web application is comprised of three sub applications: a requirements input system, a communication chat system and an issue/bug tracker system. The following section diagrams the requirements for the issue/bug tracking sub application via a use-case diagram and sequence diagram.

### Sequence Diagram

The following sequence diagram highlights the main functionalities and modules within issue tracker, the issue/bug application system.

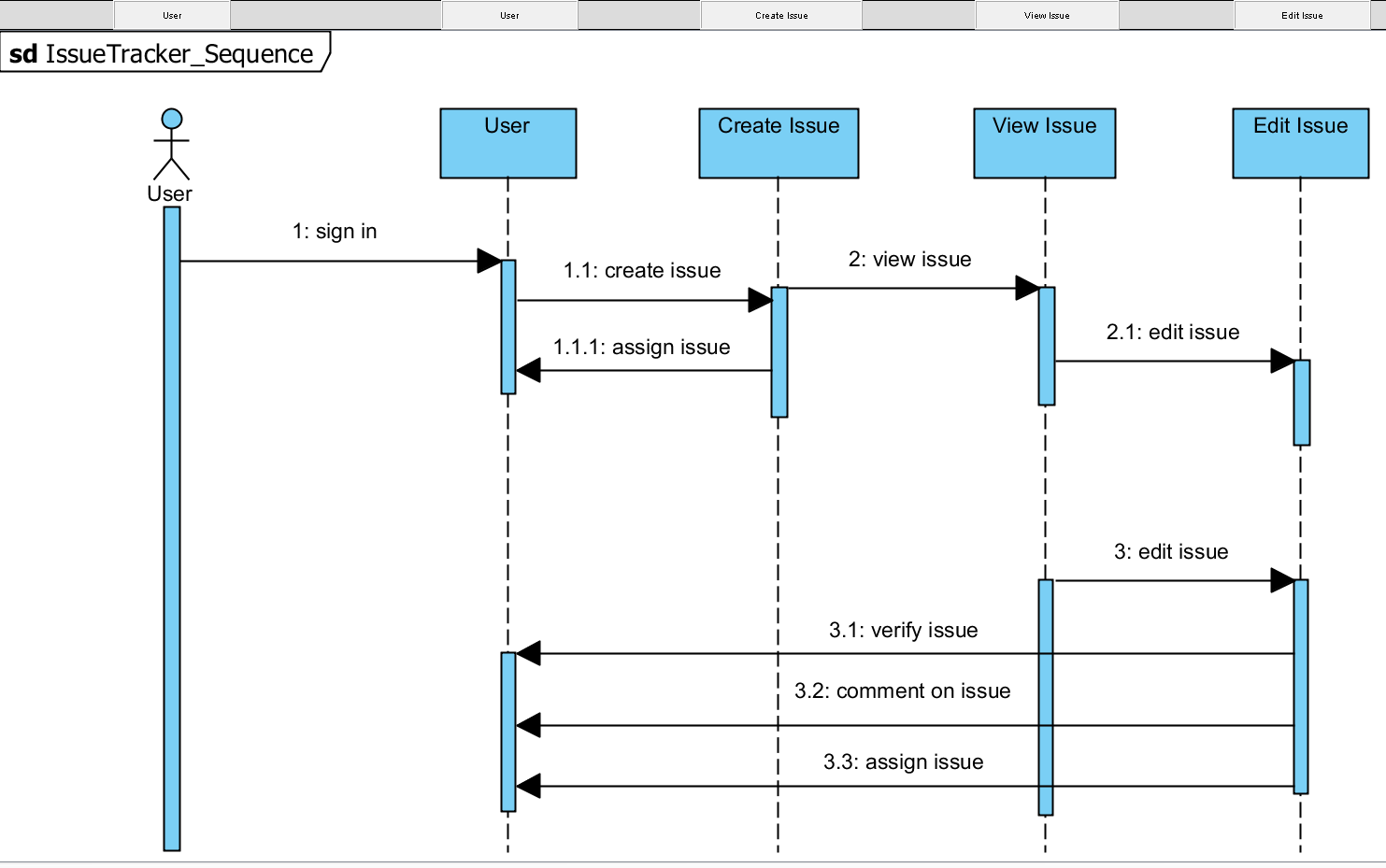


Figure 1- Sequence Diagram

### Use Case Diagram

The diagram below reveals the issue/bug tracking client requirements via a use case diagram.

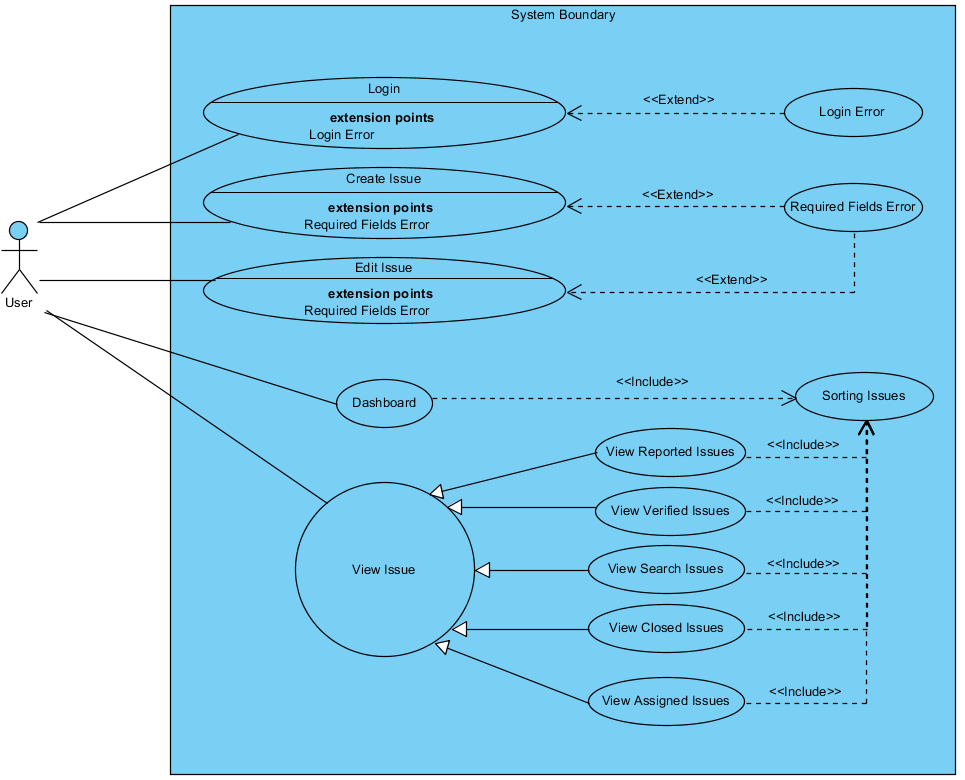


Figure 2- Use Case Diagram

### State Diagram

The state diagram reveals the behavior of the issue tracker system. It reveals the available states and the navigation from state to state at a reasonable level of abstraction.

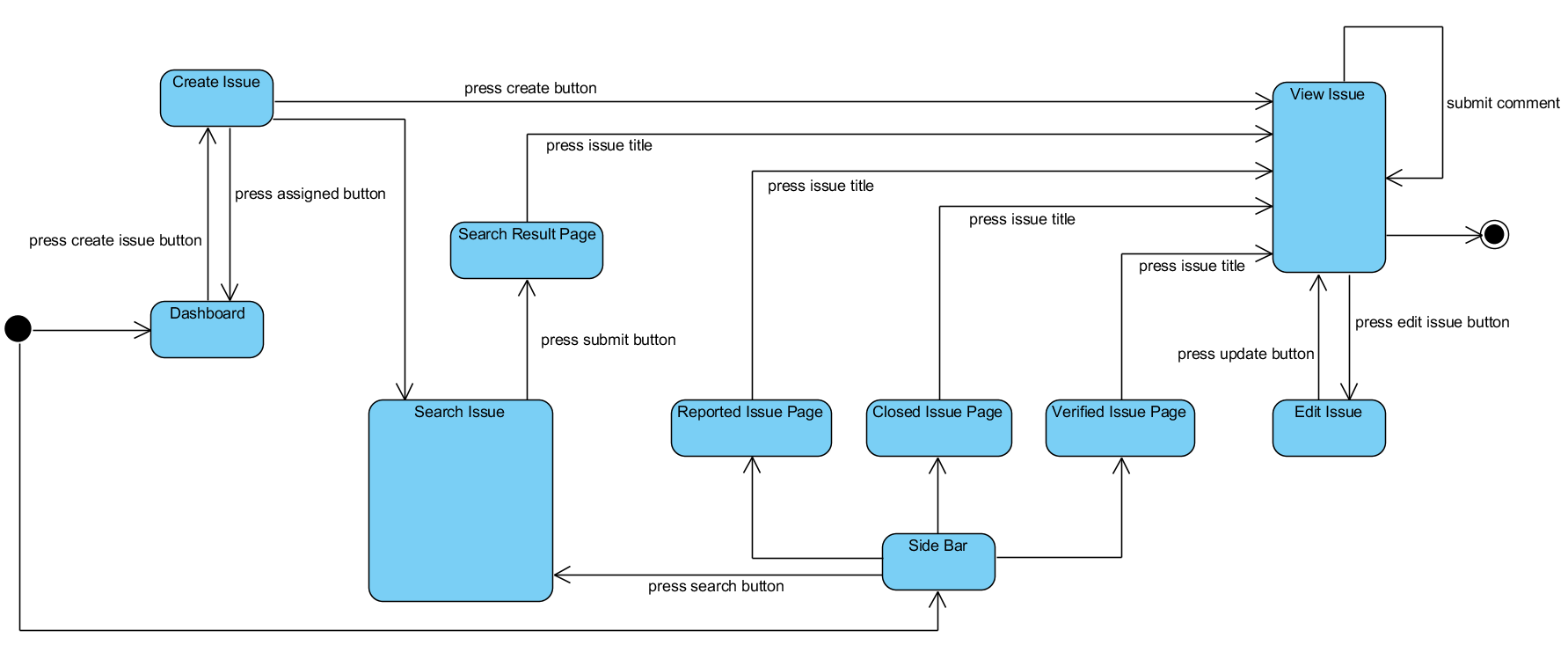
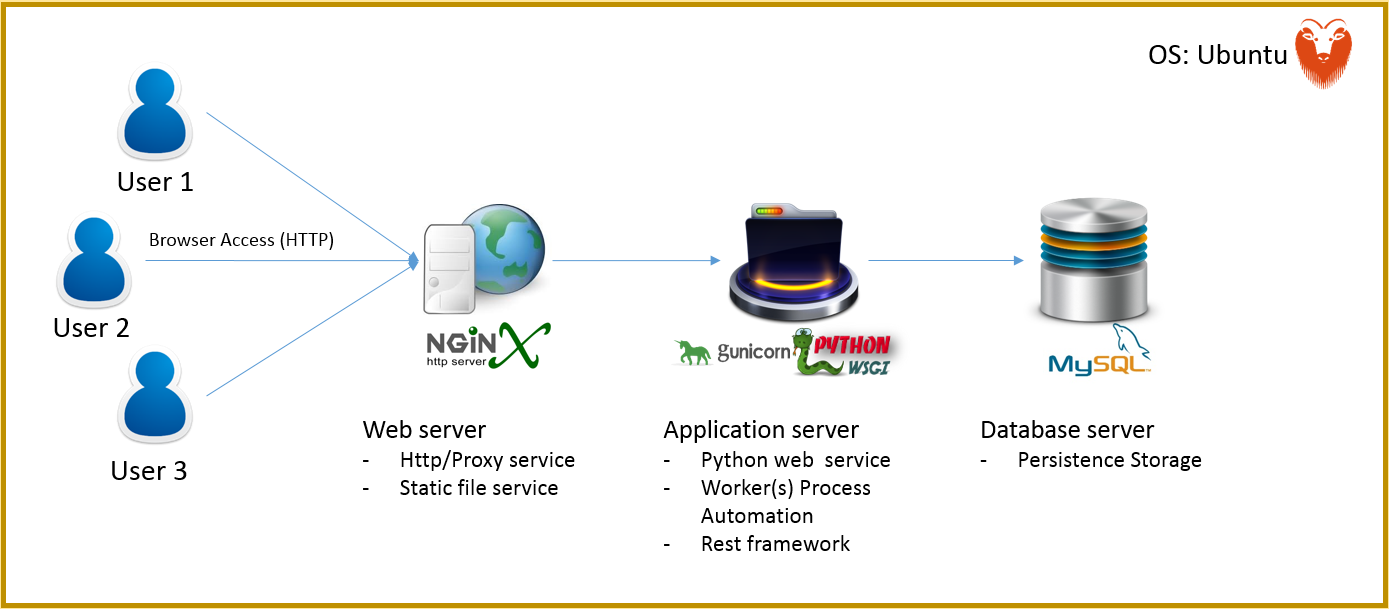


Figure 3 - State Diagram

## System Design

The Django web application will be hosted on a LNMP stack (a slight variation of the LAMP stack). The following software core components housed the 3blueprints website. The diagram below graphically illustrates the software stack.



Linux: Operating System to house manage and house the software components. The group used a Debian variant Linux system named Ubuntu, version 14.04, codename Trusty Tahr.

Nginx: HTTP server and web cache to serve pages and reverse proxy the application server.

Gunicorn/WSGI: Application server to run the python web service, parallel worker processes and REST framework.

MySQL: Relational Database for Persistence storage.

# Design Patterns & Principles

Web frameworks are becoming increasingly popular because they develop the common modules and allow teams to focus on their application code. Given the relative simplicity of the project, our application logic did not require extreme external design and rather our team opted for the utilization of Django’s existing modules and architecture. As a standard and powerful web framework, Django utilizes design patterns in every module.

## Singleton pattern & DRY principle

*Singleton* classes are used in the get\_absolute\_url() and reverse functionality, which essentially obtain URLS in their final form and/or allow for redirection. The alternative would be to use static URLs and violate the *DRY (don’t repeat yourself) principle* thus increasing *essential difficulties*. The team used these functions in the models.py on the Issue and IssueComment models (See class model section for more information) to direct the web app to the view\_issue view respectively.

## Factory Method Pattern

Factory methods are common in the Django internal architecture, a common example is form fields. Forms in Django allow for the creation of HTML form field but provide many of the basic functionalities such as field validation. For Example, the team utilized forms in forms.py for the SearchForm and CommentForm functionalities. The factory method is used because of the different field data types. A CharField factory contains its own validation regarding issues such as length, input type, error messages and so on.

## Model View Controller (MVC) Pattern

Django was developed based on an “MVC Like” Architecture named Model View Template (MVT). The MVT pattern follows MVC closely enough that it is an MVC framework. The framework still separates the three common functions: data access logic, business logic and presentation logic. However, the controller portion is handled by the framework itself using the URLconf which selects the appropriate python view function. Therefore, models are still models, templates act more like views and views more like the controller (through URLconf or urls.py).

The package diagram below illustrates our teams’ use of MVT

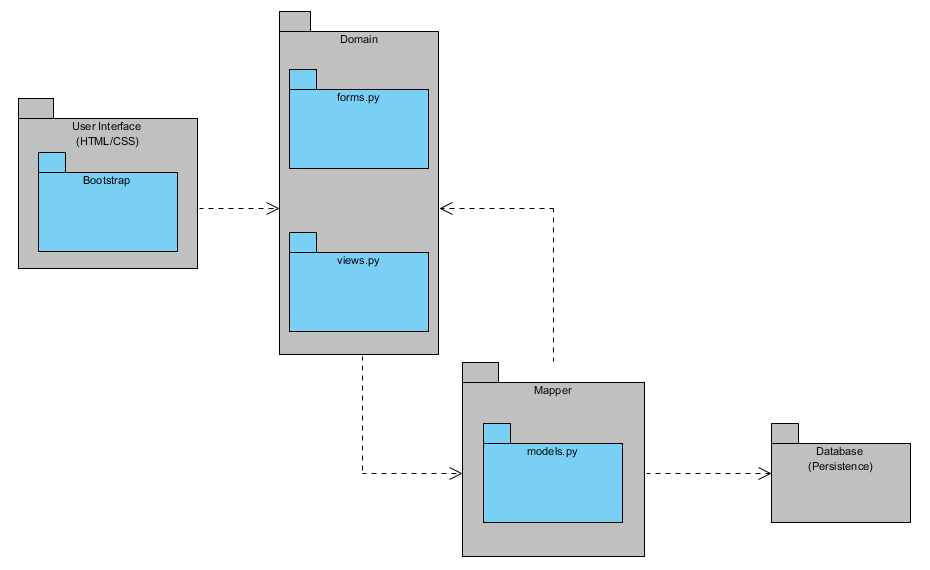


Figure 5 - Model View Template/Package Diagram

* Model (M in MVC & MVT)
  + Mapper Package (models.py)
    - Entities and Records
* Domain (C in MVC, V in MVT)
  + Domain Package (URLconf via views.py, forms.py)
    - Application Logic
    - Validation Logic
* User Interface (V in MVC, T in MVT)
  + User Interface Package (Templates, Bootstrap)
    - Templates, HTML/CSS

# Key Algorithms

The key algorithm in the project was the search algorithm. Essentially the search algorithm represents the application as a whole (see state diagram). By allowing the user to search by any issue attribute we have provided the functionality necessary to meet our application goals of interface design usability, collaboration simplicity and communication efficacy.

The search issue algorithm spans across most of the applications modules.

There are three main parts to the search algorithm:

1. Issue model
   1. Data Object and database interface that stores all issue attributes (Title, Description, Assignee, Verifier…)
2. FilterIssueQueryset object
   1. Filter methods
      1. Methods which take search form data and filters based on all the form submission parameters in the Issue Model then returns a query.
      2. Example filter:
         1. filter\_description(query)

If (description <> NULL)

query = Issue(description=query.description)

return query

* 1. filter\_issue\_results(data)
     1. Main method which uses all the filter methods (description, title, assignee…..) to return an aggregate query that returns each query set until the issue results of the objects the user has filtered by are fully listed.
     2. Example use:
        1. For each field in form

search = filter\_description(%value%)

if (search == true)

query = search

final\_query = IssueModel.filter(query)

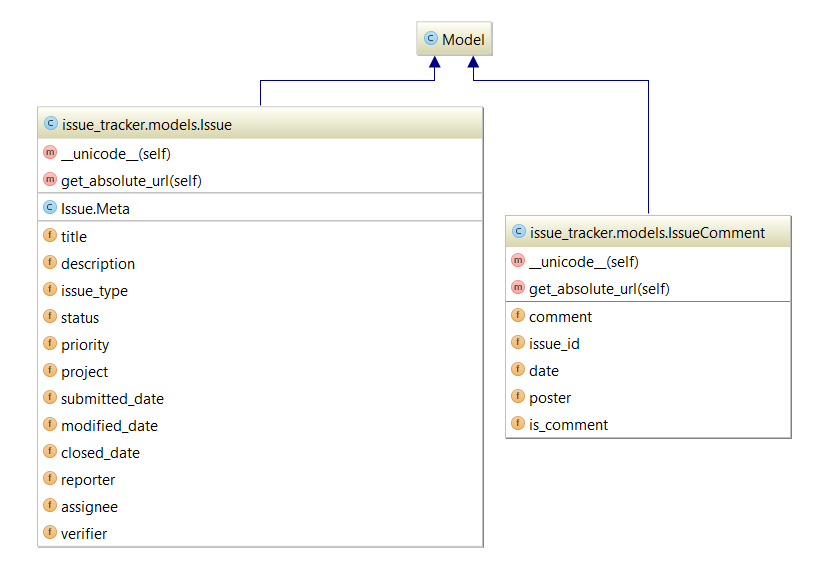
* 1. *The above is pseudo code to illustrate how the algorithm works please refer to filters.py under the issue\_tracker Django app in the project to see actual code.*

# Classes and Methods

The Django framework provides many of the common base classes required to get a web application running. It is the responsibility of the developer to leverage these classes with their custom logic to create an application which provides a unique service. The MVT (Model-View-Template) architecture Django uses compose the base of the project and the subclasses created by the teams application developers compose the application.

## Model Class (Mapper Package)

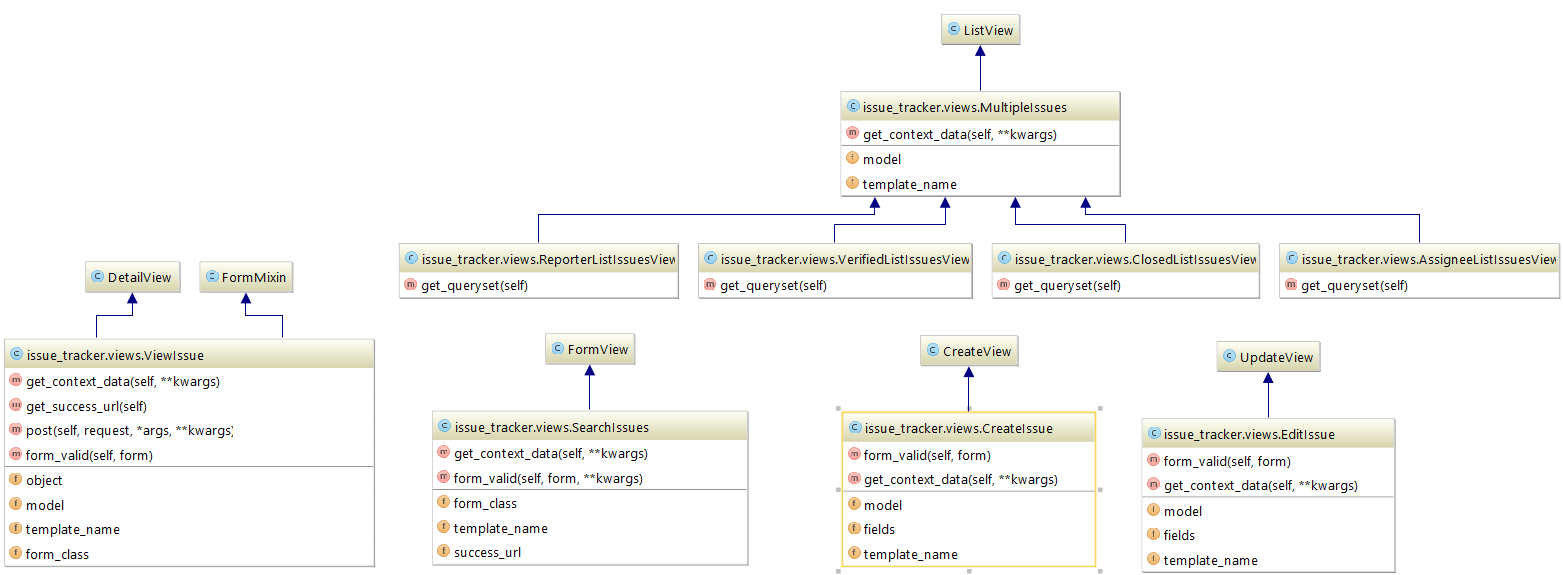
First the Model class, sub classed into the Issue and IssueComment models, used to represent the data of the application’s main component issues and their respective comments. The default Django user model was used for the project by the three different applications and does not appear in the model file as it is a part of the Django internals.



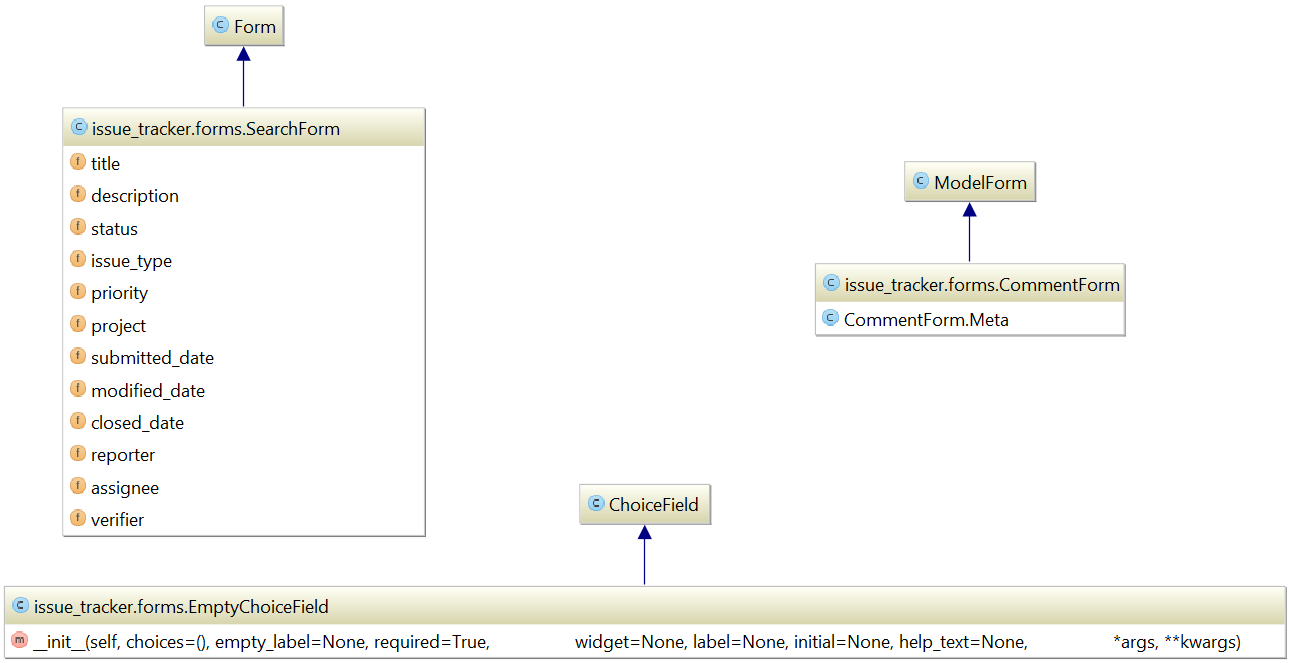
## Views (Domain Package)

### Django Views

The class diagram below reveals the view class, which when combined with the URLconf module of Django acts as the MVC controller and contains much of the application logic.



### Django Forms

The class diagram below reveals the form classes, which contain the validation logic thus being a facilitator to the Django views (controller in MVC) and templates interface (View in MVC). 

# Testing Design

Owing to the late stages of integration, all of our existing test suite was run on a locally generated instance of the issue tracker application and can be found under the tests folder in the issue tracker app.

## Test Modules

1. Login
2. Create Issue
3. Edit Issue
4. Search Issue
5. View Issue
6. Side Bar

By implementing the page object design we created a fairly flexible infrastructure that allowed us to modify tests quickly and only according to specific templates when they were altered. This also allowed us to develop some very clear test cases. Essentially each subset of tests found in a file is linked to a page that we operate on the front end, and while they may interact with each other, the goal of those tests is to test that page specifically.

Our coverage of the app’s functions and processes is near-100%. We had full coverage of error functionality, search, navigation, login, issue creation, issue edits, and sorting features. We had high coverage of interactive visible elements, but were not complete there (we did not have a test confirming quantity displays in the side navigation were accurate, for example).

All of our current tests were 100% satisfied at the time of integration, and based on their reports we were confident of a functioning product when integrating with the other teams.

# References

<http://www.django-rest-framework.org/>

<https://www.djangoproject.com/>

<http://getbootstrap.com/>

<http://www.seleniumhq.org/>

<http://www.agilemodeling.com/artifacts/stateMachineDiagram.htm>

# Glossary

1. DRY Principle
   1. Don’t Repeat yourself: a principle of software development that focuses on the reduction of repeated code and thus encouraging high cohesion and even loose coupling
2. Essential difficulties
   1. Complexity: Software is nonlinear and more complex than any other human construct. Therefore, it is extremely difficult for any one person to understand all parts at all times.
   2. Conformity: Different timelines, designs and interfaces creates essential difficulties.
   3. Changeability: Software continuously changes with pressure and requirements.
   4. Invisibility: Software cannot be understood and discussed by all members.
3. Model-View-Controller
   1. An Architectural design pattern, specifying the separation of data access logic, business logic and presentation logic.
4. Singleton Pattern: A pattern which restricts the instantiation of an object to only one instance, used for coordination of actions. Usually used when a single global instance of an object is needed to monitor actions in order and is that job is critical, such as logs, database locking technologies…
5. Factory Method Pattern: A creational pattern used to create objects dynamically without knowing the object type it needs to create ahead of time. This allows a client to call for an object from the factory and the subclass to create it for you keeping it decoupled from concrete types.