1. Introduction

<Describe the purpose of the document and give a brief overview. Include a guide for different types of stakeholders.>

2. Background

<Give a brief statement of the problem and the intended solution. Describe the main stakeholders and their concerns. Describe any existing systems, especially those that have interfaces with your system. Include a shortened version the very highest level points from the Problem Statement, and give a reference to that.>

3. Functional Requirements

<Describe the main functional requirements. You do not need to include all of the use cases, but a few examples might be helpful, as noted in the Guidelines, *above*. Refer the reader to the Requirements Document for more details. Pick at least one use case and

detail it with design information, as a prime example of how the system will work doing its main function.>

4. Quality Attributes

<Use the scenario table format to describe all of the quality attributes that must be included in the final system. Rank these architectural drivers, in terms of their importance. As noted in the guidelines, above, the most important of these quality attributes, for the system to succeed, should be given more space.>

5. Patterns and Tactics

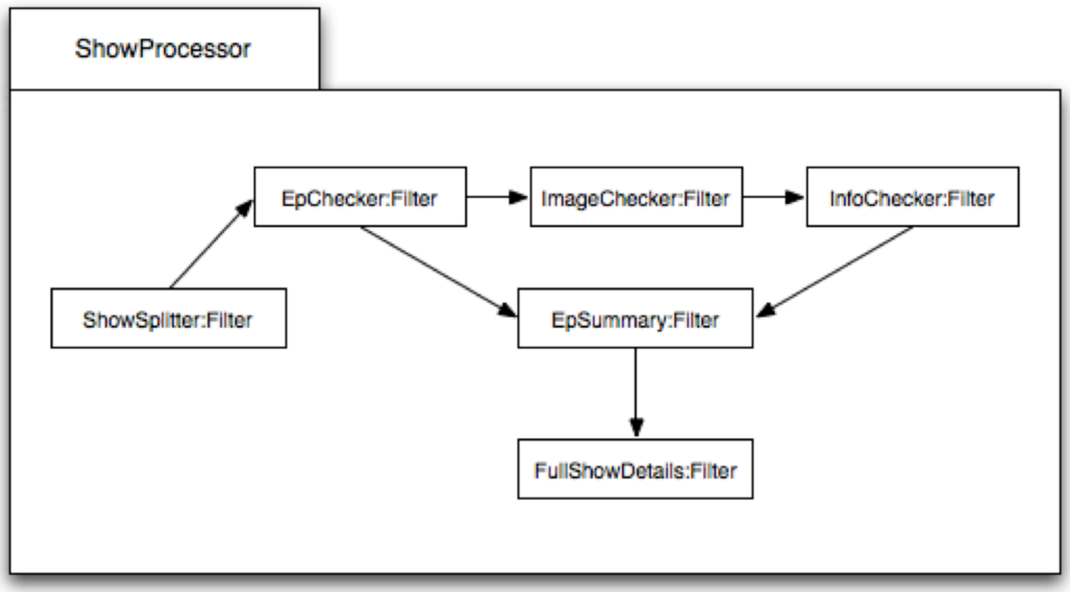
< Describe the tactics used to achieve each quality attribute. Describe any patterns that are used to package tactics. If you don't have any patterns, describe how different forces influence the choice of tactics. You should give the most attention to what is believed to be the most important of these quality attributes – how to achieve that.

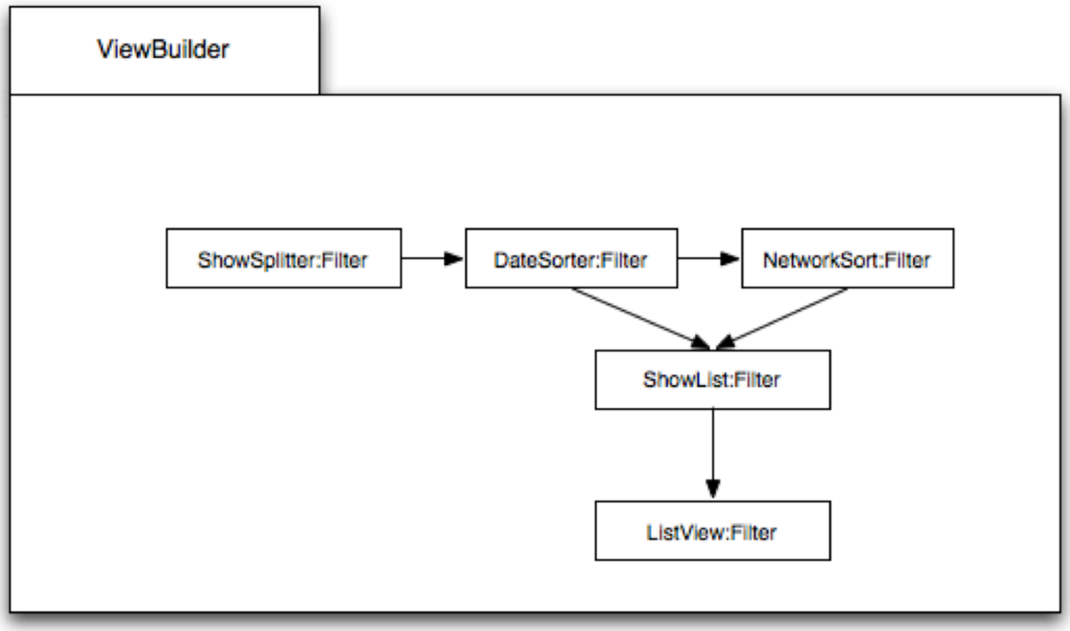
This section especially should record your rationale for the design. It should say what alternatives you considered, and why you chose the one you did. Your instructors will be paying special attention to what you say here!

If you used "patterns" in your system, in the sense of the Gang-of-Four patterns, and want to elaborate on those, that probably should go in Section 7, as a part of your Framework discussion.>

6. Views

A. Component and Connector View





1. Elements

Above is our component-and-connector diagram. This diagram shows the run time entities of our system, detailing how it is supposed to work. It is made up of various elements. The ShowSplitter element divides the shows into parallel processes, allowing a pipe and filter architecture to be applied. The EpChecker element checks to see if the episode itself is available locally, if not it downloads from TVDB. The ImageCheck element checks to see if the episode images are stored locally, it downloads the from TVDB if it is not. The InfoChecker element checks to see if the information pertaining to a certain episode is available locally, and downloads it from TVDB if not. The EpSummary element provides a quick summary of an episode, and if it is not available locally it downloads it from TVDB. The FullShowDetails element provides full information (including actors, summary, credits, etc...) for a given show. The DataSorter element provides a way to take a set of TV episodes or shows and apply a logical ordering to them. The NetworkSort element does the same but relies on the TVDB api for information. The ShowList element takes data from the sorter, and formats it so it can be displayed in a logical order to the user. The ListView element provides the formatting for the list of shows, the ShowList.

2. Relations

The relations exist in the ShowProcessor and ViewBuilder. The filters in these elements depend on each other to pass data throughout the system. Similar to a Pipe and filter model.

3. Context

The views show the high level breakdown of software components internal to SickBeard, in terms of data flows. This is an alternative to an OO view. It is lower level than an allocation view, yet a higher level then code.

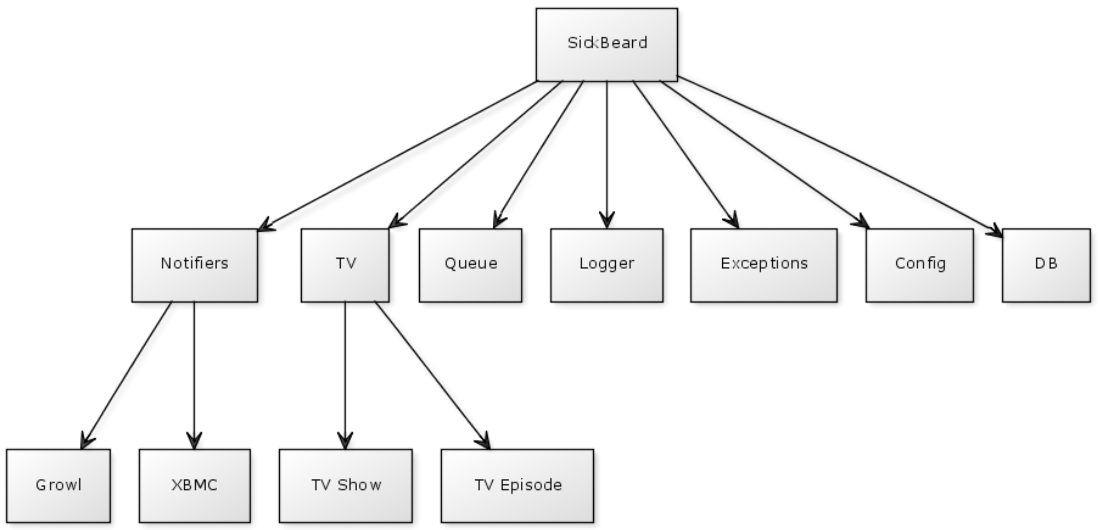
4. Variability

Altering the behavior of an existing filter, or creating new filters and inserting them into the internal workings of SickBeard is how to exercise any variation points in the system.

5. Architecture

SickBeard started with one core functionality, and as it added more functionality it extended its existing set of filters to do more with regards to the TVDB API, file management, etc.

B. Module View



Above is the class diagram for SickBeard. The diagram shows how the different classes in our system interact with each other.

1. Elements

Our system is made up of many different classes, and controlled through one main class called Sickbeard. The main class is used to control the web interface, however it knows very little about how the system actually works. In order to get the implementation details it makes class to all of the different classes. The system functionality is split up into many different classes in order to hid different parts of the system from each other. This makes working on and individual piece of the system much easier to understand, because the developer does not have to worry about the implementation details of the other parts. The system works by first loading the data stored in the database and accessed through the DB object, into tv show and tv episode objects. After loading the data, the system checks each show to find upcoming episodes, if one is found it is the added to a queue for processing. While processing the queue it will log any errors it finds in the data, as well as send notifications to the user about the upcoming episodes.

2. Context

The diagram purposefully does not show how the system interacts with external services because this interaction is shown in the allocation view.

3. Variability

In order to exercise any variation points in the system, a developer can simply create new classes to be used by the main controlling class. These new classes can incorporate new features, or simply come about from a refactoring of the old system.

4. Architecture

This design came to be through a long process of trail and error. When the system was first designed it did not have separate classes for different functionality. As the system became larger it was apparent that these separate classes were necessary to help new developers work with the system, as well as help current developers keep a separation of concerns.

C. Allocation View

1. Elements

Hardware

Client

The client computer can be any desktop, laptop or other Internet capable device. These devices will be the ones viewing information. People will view the software through means of a third party browser, which will allow the end user to see the server's interface. This hardware client can, in rare cases, the server itself. The reason for this is usually the local server is not hooked up directly to a monitor or easily accessible by a user.

Local Server
This is the server that will be running SickBeard. It is required to run python and other required libraries. The server will always be running SickBeard and could possibly have multiple connections to the SickBeard software. We will have full control over the local server and any of the relevant processes needed to run SickBeard.

Remote Server
This is the server that implements the API that we need. We do not have control of anything other than the data that the API provides. This is also the only data we have access to.

Software

Show Fetcher
The Show Fetcher is responsible for going out to TheTVDB and fetching information about television shows. Some of the items fetched include, show descriptions, episode images, air times, network and plot summaries. This is the primary component that is responsible for server-to-server communications.

Display Driver
The Display Driver is responsible for taking the local data and presenting it to the user. It is also capable of pulling information through the show fetcher. This means that the Display Driver can display live information if it is required in certain views.

Browser
This is the external browser that the user will be using. It is simply used to interpret and display the views produced from the Display Driver.

Relations
The primary relation is the one from SickBeard to TheTVDB, which happens through the Show Fetcher. The Show Fetcher communicates through the API provided by TheTVDB. This allows an easy connection for us to get the information we need out of TheTVDB.

2. Interfaces

TVDBAPI
Resources
TV Info
This is the python object that provides methods to query info from TheTVDB. It's actually a class that can be called as a method that allows us to obtain information.

Data Types
TVDBAPI Class
This is the API class that we can make calls against. There are several methods that allow us to query multiple pieces of information such as seasons, series or individual episodes. It also provides an easy interface for obtaining images.

Exceptions
There are very few exceptions in this system, as this is the nature of the system's design. We are using the model that exceptions should be used for exceptional circumstances.

Variability
There is minimal variability due the lack of need for variability. The API provides all the data in an easy to access form so therefore we don't need different forms of this.

Quality Attributes
SickBeard provides a clean pythonic interface that any python programmer can pick up and understand. This is awesome as it was very easy for each of our developers picked it up right away.

Element Requirements
The system requires it's own database and theTVDB webserver. These are the only two software requirements in order to run SickBeard successfully.

Rational and Design Issues
TheTVDB existed before the API did and thus was written to bridge the connection between TheTVDB and any python interface.

3. Context
This system's Environment is the Hardware and Internet. These environmental elements are both required for proper operation, although SickBeard can run without Internet in a limp mode. Many of the functions are disabled in limp mode, but the user is notified about the problem.

4. Variability
In order to provide an easy implementation, we need to minimize our variation. We do this to keep items succinct and simple to keep variation down.

5. Architecture
The overall structure is built on existing web service technologies. The overall structure of the program will be dead simple for anyone to pickup. It also employs web standard API technology implementations so anyone familiar with standard web API technologies will be familiar with the system.

<These are the pictures and their explanations. Provide an introductory paragraph listing the purpose of each view. What kinds of views do you need? See the Viewtypes discussion, below. Include each view as a lettered subsection of this section. That is, the first view should be 6.A, the second should be 6.B, etc. Each view should have all of the following subsections:

1. Primary presentation – this will often be a diagram
2. Element catalog
    A. Elements - brief description of each
    B. Relations - brief description of each
    C. Interfaces - for views that include them
        1) Interface identity - unique name
        2) Resources provided - syntax and semantics
        3) Locally defined data types - if used
        4) Exception definitions - including handling
        5) Variability provided - for product lines
        6) Quality attribute characteristics - what is provided?
        7) Element requirements - names of required items, assumptions
        8) Rationale and design issues - why these choices
        9) Usage guide - protocols
3. Context diagram -- how the system relates to its environment
4. Variability guide – how to exercise any variation points
5. Architecture background – why the design reflected in the view came to be
6. Glossary of terms used - alternatively, you may place these in a separate glossary for the whole document
7. Other information - if needed

Viewtypes:

There are typically 3 kinds of views you need to show, which translates to 3 or more views in this high-level document:

Modules – These are "design time entities" of your system. The structure of the OO design (or other software design methodology). This is probably the same thing as the "framework" described in the next section. So, Section 7 becomes effectively an extension of the picture and the basic explanation of it given here. Often, there are multiple versions of this viewtype – say, a UML class diagram showing OO decomposition, generalization and uses; and then a view of the software "layers" in the system, perhaps showing how your software uses third-party software.

Component-and-connector (C&C) – This shows the "run time entities" of your system. How is it all supposed to work? For example, how does a "message" or "transaction" get through your system? Or, how does the main use case happen, in terms of pieces of the system interacting? The explanation should be "stream of execution." Often, your system's "style" comes out the most in this figure, and readers can see the pipes-and-filters, or client-server design. Interfaces to other systems would be at least generally described here. There are UML diagrams you can use for these, too – see SEI Article.

Allocation – This one always shows hardware as well as software, a larger picture of the system in its environment. Especially, it shows what software on what box talks to what other software on its box. The links between the boxes should be as informative as possibly in their legend, saying things like what protocols or interfaces are used, and what the capacity of these links will be. However, there are additional kinds of allocation views that you may also need, like one showing how it will be deployed or installed.

Data – These include things like entity-relationship diagrams for the database. Now, in this architecture document, you might only show the most critical parts of that design, since otherwise you could easily have half the document describing the data!>

User interface – Typically these interaction design images appear in a more detailed design spec. However, as with data, an exemplary one which gets across a major point about your design could be appropriate. The software framework supporting user interaction would be described in Section 7.

7. Framework

<It's presumed that, as you write this, or shortly thereafter, you create the high level design for your system. This is its "framework." Describe the components of the framework constructed by the development team. You may want to use a diagram or refer to one of the views in the previous section. Describe how the framework enables/enforces/facilitates the software architecture.

A framework is a general OO design that makes it easy to attach more detailed pieces. So, it has, for example, the most general "parent" classes of the system. And it shows how these connect to others. Typically a framework is full of generics, iterators, interfaces, methods designed to be overloaded, and all that good OO stuff. The description should be supported by a UML class diagram which shows the connections in the whole system (probably in Section 6, above).

This is the section where things like "model view controller" and "Gang of Four" patterns would be a part of the discussion. However, if you were going to fully flesh out the description of even your framework, that would likely be too long for this high-level intro-to-everything.>

8. Acknowledgements

<Acknowledge the help of the customers and developers. Also give thanks for any other sources of material you used, such as authors of patterns.>

9. References

<Include citations here, both internal to the team and external. Use a standard format, such as IEEE (http://standards.ieee.org/guides/style/section7.html) or ACM (http://moon.cse.yzu.edu.tw/acm/docs/ihhuang/citation/). Use a standard format, such as IEEE (http://standards.ieee.org/guides/style/section7.html). >

10. Revision history

<Table containing the revision number, date, who made the change, what the change was>

11. Appendices

A. Glossary

API	Application Programming Interface. A standardized set of methods provided for developers to use when connecting to the application. In the context of the TVDB, refers to a set of Web-based data connections that return information about TV shows.
SickBeard	An open-source application that interfaces with the TVDB in order to provide an interface into a local TV collection with information about each show.
TVDB	Television Database. A freely available collection of information