Software Lab EE 461L
Phase IV Design Report

Team A7: Board Game Database

Cedrik Ho, Allegra Thomas, Grant Ross, Sanne Bloemsma

11.30.2020

INFORMATION HIDING

Searching's more modular design leaves room for adding searching fields, adding models, and restricting search precision (i.e. if all fields must contain the searched string). Search is also set up so that fields not available for a model cannot be searched and the search function does not alter available filters on runtime. (See Design Pattern section for more information.)

Filtering takes a string from a function in listbase.html, this string is a unique identifier for the dictionary that is selected in the filter.py. This dictionary is passed to a function call Apply filter, that uses the dictionary to execute a mongodb find call that returns a cursor of elements that match the filter declared by the dictionary. This cursor is passed back to listbase.html and iterated through to populate the page with all elements.
Filtering functions were moved to another file. This allows filtering, parsing and cursor object creation to be self-contained and built separately from the routing of the front end UI. With parsing based on the string of the filter, filters can easily be added and additional features such as filter stacking can be developed independently from other features such as sorting or searching.
The main disadvantage of this design is that filtering must be called by each individual model that we have. We have three models and thus three new calls of the same functions. If the modules could share one instance of filtering, then it would save memory and increase efficiency as the stack has less methods to keep track of.

On the list pages, the filters are generated based on an array of strings that is passed in. Each model has its own filter array, which is a global variable, and adding or removing from the array doesn't create problems when loading the list pages. This means that new filters can be implemented without having to modify the list html file.

For the database code, change primarily occurs when adding new documents to the database. The database code is split into 3 files, one for each database collection (board games, genres, and publishers). Each file only affects documents in one collection. For example, if you run the script in the publisher database file, it only affects the publisher collection and not the genre or game collections. Additionally, using any of the scripts to add new documents to collections cannot delete any old documents. It can only update existing documents, and it also prevents insertion of duplicate documents. This modularization scheme makes it very easy to safely add new documents to the database by ensuring that old documents cannot be deleted and that inserting new documents will not cause duplicate entries. Additionally, it would be easy to add an entirely new collection to the database if needed, because all information for the collection could be added with just one file. One disadvantage that this model has is that the publisher and genre collections use some information from the board game collection to populate their documents. When the board game collection is updated using the script in the board game collection master file, it specifically will not update the publisher or genre collections. Therefore, the publisher and genre collection scripts will have to be run separately to get the updated information from the board game collection. However, this disadvantage is not worth mitigating because it's preferable to run two additional scripts than risk modifying a collection you didn't intend to (i.e. running the script to update the board game collection and it automatically updates the genres and publishers collection when you only wanted to update the board game collection).

DESIGN PATTERN

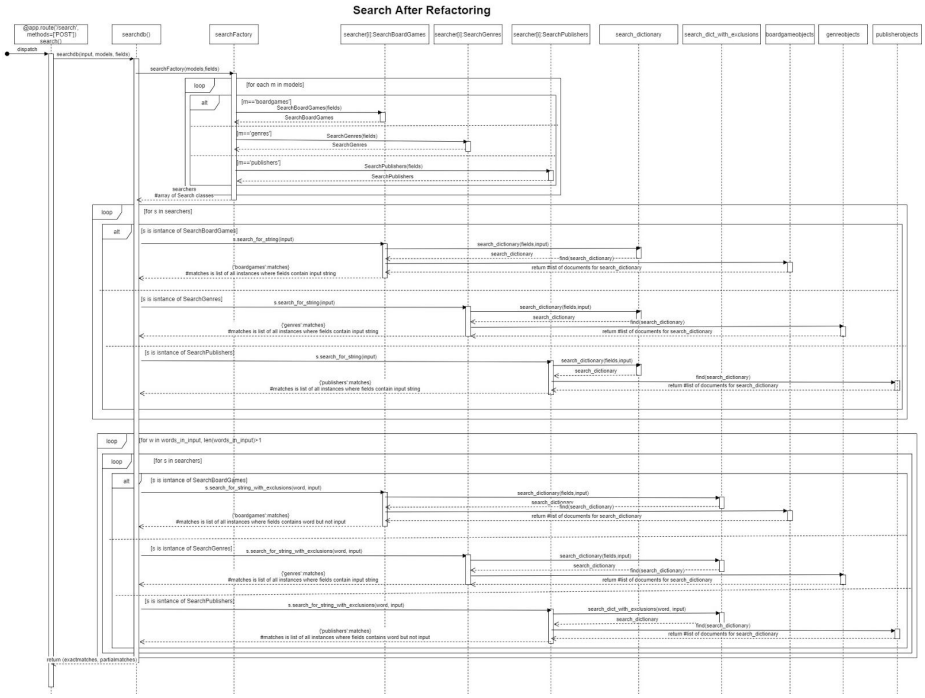
Factory Method Pattern

The refactored python code for searchdb follows the Factory Method Pattern. Rather than having the client creating the searchdb have to know the exact class name and how many possible search types and detailed inner workings of the model they want to search, the searchFactory function allows searchdb to create the searcher based on the model name.

Advantages: The code is more readable and less reliant on conditional statements. Future iterations of the project can add, modify, or remove models and fields without affecting the core search structure. Changing how searching is executed on the database is also modular. If we wanted to adjust the partial matches to not exclude exact matches, we could simply change one line. Despite using more classes and methods than before, compared to the approach we initially used for searchdb this takes fewer lines of code and is easier to debug as testing can become more modular as well.

Disadvantages: Looking at the UML, the factory method pattern appears more complicated. Initially, searchdb was only one function and not nearly this complicated with class declaration.

UML Diagram (link here for larger view)



Pseudocode:

For each model in input models create a searcher

For each searcher

update exact matches with the results for model (search string)

For each word in the search string

For each searcher

update partial results for word with the results for model (word not search string)

Return exact matches and partial matches

Code Snippet Before and After Factory Method Pattern

```
def searchdb(input, models, fields):
   exactmatches = {}
   partialmatches = {}
   if models['boardgames']:
      searchdict = {}
      if list(fields) == ["all"]:
         searchdict = {"$or": [{"Name": {"$regex": ".*" + input + ".*",
'$options': 'i'}},
                        {"Description": {"$regex": ".*" + input + ".*", '$options':
'i'}},
                        {"Publisher": {"$regex": ".*" + input + ".*", '$options': 'i'}},
                        {"genres": {"$regex": ".*" + input + ".*", '$options': 'i'}}]}
      else:
         fieldsearches = []
         for f in fields:
            fieldsearches.append({f: {"$regex": ".*" + input + ".*", '$options':
'i'}})
         searchdict["$or"] = fieldsearches
      exactmatches['boardgames'] =
list(boardgameobjects.find(searchdict))
   ……...
   partialwords = input.split()
   if len(partialwords) > 1:
      for word in partialwords:
         partialmatches[word] = {}
         if models['boardgames']:
            final = []
            searchdict = {}
            if list(fields) == ["all"]:
               searchdict = {"$or": [
                  {"Name": {"$regex": ".*" + word + ".*", '$options': 'i'}},
                  {"Description": {"$regex": ".*" + word + ".*", '$options': 'i'}},
                  {"Publisher": {"$regex": ".*" + word + ".*", '$options': 'i'}},
                  {"genres": {"$regex": ".*" + word + ".*", '$options': 'i'}}]}
            else:
               fieldsearches = list()
               for f in fields:
                  fieldsearches.append({f: {"$regex": ".*" + word + ".*",
'$options': 'i'}})
               searchdict["$or"] = fieldsearches
            matches = list(boardgameobjects.find(searchdict))
            for m in matches:
               if m not in exactmatches['boardgames']:
                  final.append(m)
            partialmatches[word]['boardgames'] = final
         ……...
   return exactmatches, partialmatches
```

```
def searchdb(input, models, fields):
  exactmatches = {}
  partialmatches = {}
  words_in_input = list(input.split())
  searchers = searchFactory(models, fields)
  for s in searchers:
     exactmatches.update(s.search_for_string(input))
  if len(words_in_input) > 1:
     for w in words_in_input:
        results_for_word = {}
        for s in searchers:
           results_for_word.update(s.search_for_string_with_exclusion(w,
input))
        partialmatches.update({w: results_for_word})
  return exactmatches, partialmatches
class SearchBoardGames:
  def __init__(self, fields):
     if fields == ['all']:
        self.fields = ["Name", "Description", "Publisher", "genres"]
     else:
        board_game_fields = ["Name", "Description", "Publisher", "genres"]
        self.fields = []
        for f in board_game_fields:
           if f in fields:
              self.fields.append(f)

  def search_for_string(self, input):
     return {'boardgames':
list(board_game_objects.find(search_dictionary(self.fields, input)))}

  def search_for_string_with_exclusion(self, word, input):
     return {'boardgames':
list(board_game_objects.find(search_dict_with_exclusions(self.fields, word,
input)))}
……..
def search_dictionary(fields, input):
  search_dictionary = {"$or": []}
  for f in fields:
     search_dictionary["$or"].append({f: {"$regex": ".*" + input + ".*",
'$options': 'i'}})
  return search_dictionary

def search_dict_with_exclusions(fields, word, input):
  search_dictionary = {"$or": []}
  for f in fields:
     search_dictionary["$or"].append({"$and": [{f: {"$regex": ".*" + word +
".*", '$options': 'i'}},
                                {f: {"$not": {"$regex": ".*" + input + ".*",
'$options': 'i'}}}]})
  return search_dictionary

def searchFactory(models, fields):
  possible_searchers = {'boardgames': SearchBoardGames, 'genres':
SearchGenres, 'publishers': SearchPublishers}
  searchers = []
  for m in models:
     searchers.append(possible_searchers[m](fields))
  return searchers
```

REFACTORINGS

Search Refactoring

Before refactoring, the python function searchdb was 120 lines and the javascript for searchresults.html was 260 lines. This was due to repeated declarations of variables, execution of code, and code in different functions. In searchresults.html jQuery there was a set of 6 lines that check which models checkboxes are checked that appeared for multiple other functions. In searchdb the naming convention made it unclear if the variable declared in one conditional statement was the same variable declared in another conditional statement. These issues would make it more difficult for someone to make enhancements such as adding more field searches, models, or result filtering. During refactoring the searchdb function was moved to search.py, which is closer to 75 lines of code (without spacing). The javascript on searchresults.html 180 lines of code.

Code Smells for searchdb: long method, duplicate code, shotgun surgery, poor commenting.

Refactoring for searchdb:

Extract method: Consistently repeated lines of code were extracted with searchFactory(), search_dictionary(), and search_dict_with_exclusions().

Change Function Declaration: In search() in main.py, models was a dictionary with keys "boardgames", "genres", "publishers" and values True or False. It is now a list of strings for each model type that searchdb should return matches for. SearchFactory() takes on average less time as on the user end search only searches a single model type or all models (list is either size 1 or size 3). If searching was updated to search two models at a time, only the route for '/search' in main.py would need to be altered.

Replace conditionals with polymorphism: Initially searchdb used conditional statements for each model type. The Factory Method Pattern and converting searching each model to its own class made the code more modular and fewer conditionals are necessary.

Replace loop with pipeline: Originally partial search results were compared with all the exact match results after both queries had been completed on the collection. This takes $O(n^2)$ where n is the length of the collection. The loop removed any partial results where the exact result was also true. Instead of looping over the exact matches for each partial match, search_dict_with_exclusions() inspects that for each field, word is in that field and the input string is not. This significantly cut down on time as it now takes at most $2n$ for each partial match.

Refactoring for search.html javascript:

Extract method: Results filtering is handled with javascript within the html page using lots of functions checking the status of different checkboxes. The common lines were extracted to handlemodelfilters() and handlefieldfilters().

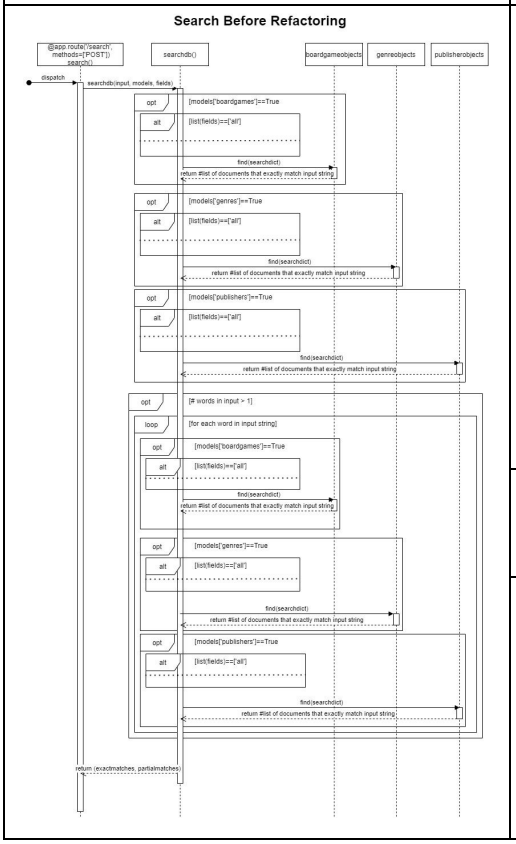
Remove dead code: Some dead code was due to the repeated methods that were extracted and other code was removed as it was a situation that would never happen.

Renaming variables: Initially each model had its own class for the minimum number of players and maximum number of players. After renaming them all to be the same (as the field filters are applied across models that are shown) the conditional check for the model was reduced to only check if it was a board game as the age filter only applies to that model.

Pull Up Field: This isn't perfectly done as a subclass and superclass relationship, however renaming of the classes to be the same (minplayers, maxplayers) for each model allowed the jQuery to hide or show the result without consideration of its model.

Search Sequence Diagram Before:

(before larger view)



Searchresults.html javascript before:

//This code was repeated 4 times line for line

```
if ($('.showgames').prop("checked") == true) {
  $('.gamerow').show();
} else {
  $('.gamerow').hide();
}
if ($('.showgenres').prop("checked") == true) {
  $('.genrerow').show();
} else {
  $('.genrerow').hide();
}
if ($('.showpublishers').prop("checked") == true) {
  $('.publisherrow').show();
} else {
  $('.publisherrow').hide();
}
filteredresults = $('tr:visible');
```

Searchresults.html javascript after:

//code was condensed and simplified to this helper function

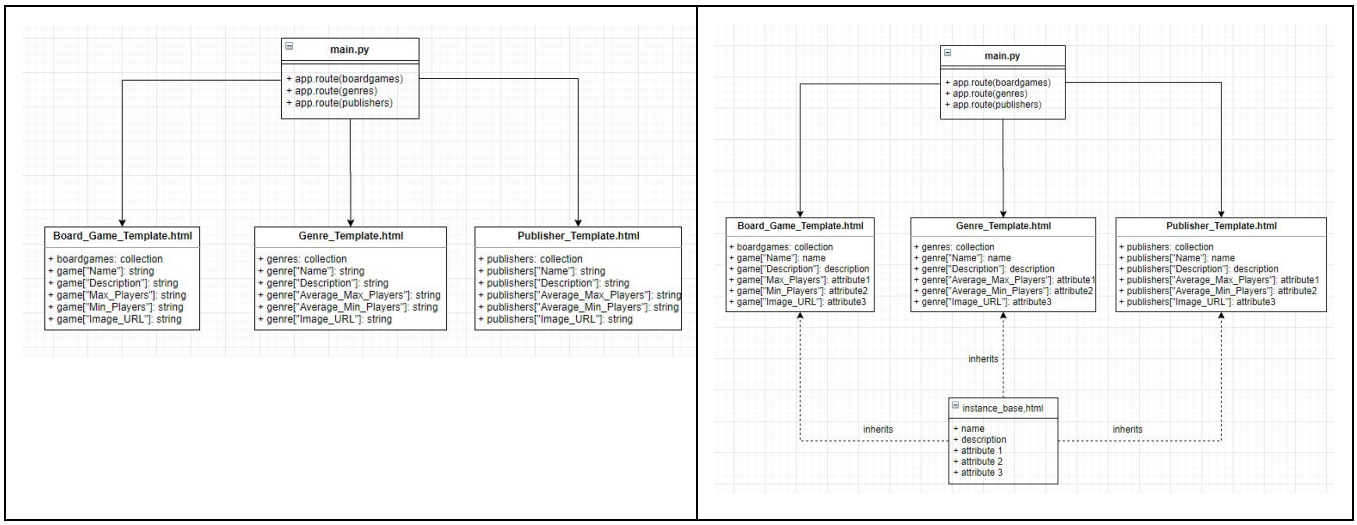
```
function handlemodelfilters(){ //updates filtered results to hold what models we need
  $('tr.gamerow,tr.genrerow,tr.publisherrow').hide();
  if ($('.showgames').prop("checked"))
    $('.gamerow').show();
  if ($('.showgenres').prop("checked"))
    $('.genrerow').show();
  if ($('.showpublishers').prop("checked"))
    $('.publisherrow').show();
  filtered_results = $('tr.gamerow:visible,tr.genrerow:visible,tr.publisherrow:visible');
}
```

Base Template Refactoring

Type of Refactoring and Justification

This type of refactoring was Shotgun Surgery, since any stylistic change to the instance pages before the refactor would need to be updated across all three model types. We made all the instance pages inherit from a base instance page to solve this. We needed this change because we spent a lot of time updating all of the pages during Phase II and Phase III.

UML Before:	**UML After:**



Code Snippet Before:

```
{% block bodywithmargin %}
    <h1 style = "font-size:30px; padding-top:100px; padding-bottom: 25px; padding-left: 30px;">Board Games</h1>
    <div class="container-fluid">
        <div class="container-fluid">
            <div class="row">
                <div class="col-sm-4" style="width: 100%">
                    <img src='{{ game["Image_URL"] }}' style="max-width:100%;width: 500px;object-fit: cover;">
                    <br>
                    <br>
                    <h5>Year Published: {{ game['Year_Published'] }}</h5>

                    <h5>Players: {{ game['Min_Players'] }}-{{ game['Max_Players'] }}</h5>
                    {% if game['Min_Playtime'] != game['Max_Playtime'] %}
                        <h5>Playtime: {{ game['Min_Playtime'] }}-{{ game['Max_Playtime'] }} minutes</h5>
                    {% else %}
                        <h5>Playtime: {{ game['Min_Playtime'] }} minutes</h5>
                    {% endif %}
                         <h5>Year Published: {{ game['Year_Published'] }}</h5>
                    <form action="/publisher" method="POST">
                        <h4>Publisher:
                            <button type="submit" class="btn btn-link" title="entry" name= "publishername" id="publishe
                        </h4>
                    </form>
```

Board_Game_Template.html

Code Snippets After:

```
{% block bodywithmargin %}
    <h1 style = "...">{% block title %}{% endblock %}</h1>
    <div class="container-fluid">
        <div class="container-fluid">
            <div class="row">
                <div class="col-sm-4" style="...">
                    <img src="{{ model['Image_URL'] }}" style="...">
                    <br>
                    <br>
                    <h5>{% block attribute1 %}{% endblock %}</h5>
                    <h5>{% block attribute2 %}{% endblock %}</h5>
                    <h5>{% block attribute3 %}{% endblock %}</h5>
                    <h5>{% block attribute4 %}{% endblock %}</h5>
                    <h5>{% block attribute5 %}{% endblock %}</h5>
                    {% block links %}{% endblock %}
                </div>
                <div class="col-sm-8" style="...">
                    <h2>{{ model['Name'] }}</h2>
                    <hr>
                    <h5>Description:</h5>
                    <p>{{ model['Description'].replace("<br />", '').replace("<p>", '').replace("</p>", '').replace("<e
                    <hr>
                    {% block multimedia %}{% endblock %}
                    {% block multimedia2 %}{% endblock %}
```

Instance_base.html

```
{% block title %}Board Games{% endblock %}
{% block attribute1 %}Year Published: {{ game['Year_Published'] }}{% endblock %}
{% block attribute2 %}Players: {{ game['Min_Players'] }}-{{ game['Max_Players'] }}{% endblock %}
{% block attribute3 %}
                    {% if game['Min_Playtime'] != game['Max_Playtime'] %}
                        Playtime: {{ game['Min_Playtime'] }}-{{ game['Max_Playtime'] }} minutes
                    {% else %}
                        Playtime: {{ game['Min_Playtime'] }} minutes
                    {% endif %}
{% endblock %}
{% block attribute4 %}Year Published: {{ game['Year_Published'] }}{% endblock %}
{% block attribute5 %}{% endblock %}
{% block links %}
    <form action="/publisher" method="POST">
        <h4>Publisher:
            <button type="submit" class="btn btn-link" title="entry" name= "publishername" id="publishername" value='{{game["Publis
        </h4>
    </form>
    <form action="/genre" method="POST">
        <h4>Genres:
            {% for genre in game["genres"] %}
                <button type="submit" class="btn btn-link" title="entry" name= "genrename" id="genrename" value='{{ genre }}'>{{ ge
            {% endfor %}
        </h4>
    </form>
{% endblock %}
{% block multimedia %}
    {% if game['Videos']|length != 0 %}
    <h5>Videos:</h5>
    <br>
    <div class="row" style="...">
        {% for i in range(0, 4) %}
            {% if i < game["Videos"]|length %}
```

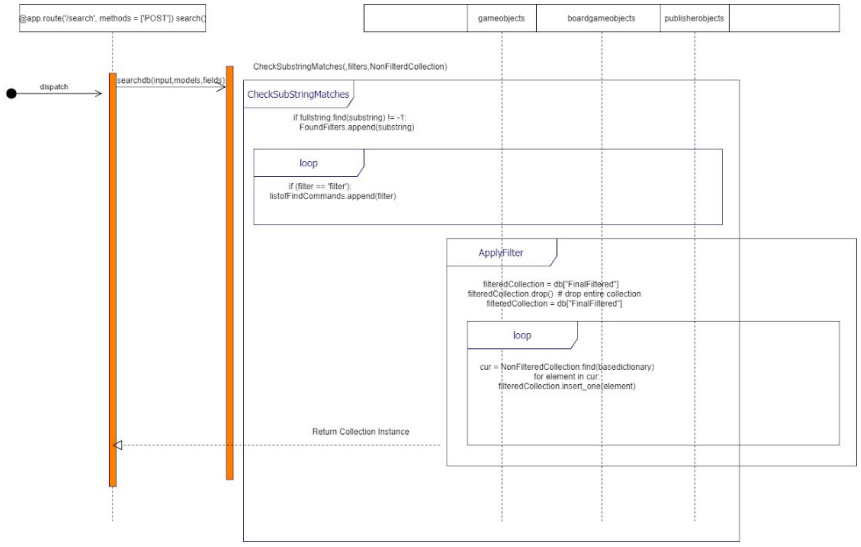
New Board_Game_Template.html

Filter Refactoring

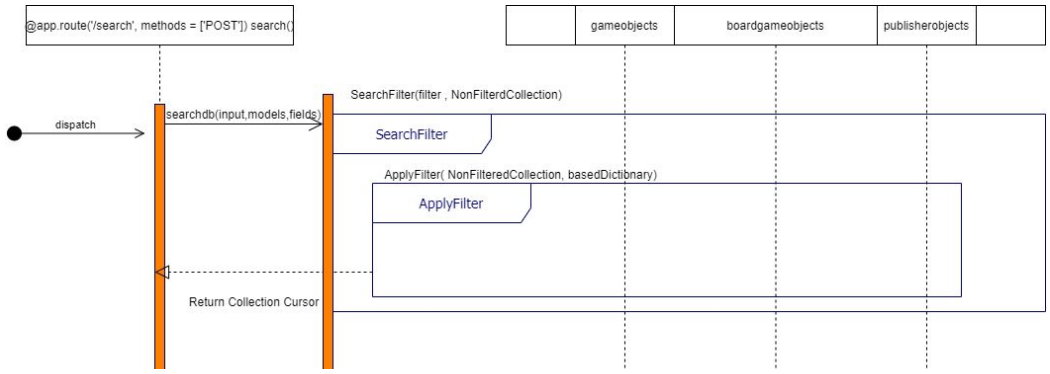
Type of Refactoring and Justification

Our filtering code was refactored to be modular and allow for future developments with an emphasis on modularity. This refactor can be classified as Speculative Generality with a focus on the removal of parameters. In Phase III, each filter had its own dictionary that was tied to the string variable of the filter selected. Many filters shared similarities or shared ranges of values for similar filters. An example of this is our players filter, which had filters for Players: 2 Players3: Players:4 and Players: 5+. In Phase III, we had 4 dictionaries and if statements to match each possible player filter selection. This is a large duplication of code and it becomes tedious to add more filters related to player count. In Phase IV, we used the split function to parametrize each string being passed to our function. This minimized code overhead from 4 if statements and 4 dictionary objects, to 1 if statement and no declaration of dictionary objects. With this refactoring, if we change or add the string that is passed to our main.py when the button is clicked, no additions to our filtering functions must be added. Furthermore, if an entirely new filter type is added, we only need to implement one new if statement to partition the string. Another change made to filters was the code to display them as options in the filter dropdown menu on the list pages. Before, every single filter option was hard-coded. After refactoring, the html page iterates through a passed-in string array and generates the filter selections in a for loop. This means that adding new filters is possible without having to touch html code. The functions were also moved to a filter.py file, so that future development and features for filtering can be developed independently of main.py. We believe that it held enough functionality that a new file was necessary. We believe this will also provide more clarity for new developers learning the code base.

UML Before:



UML After:



Code Snippets Before:	

```
for filter in FoundFilters:
    ############GAME CALLS TO DATABASE####################


    if (filter == 'Players: 2'):
        listofFindCommands.append(dict_Players_2)
    if (filter == 'Players: 3'):
        listofFindCommands.append(dict_Players_3)
    if (filter == 'Players: 4'):
        listofFindCommands.append(dict_Players_4)
    if (filter == 'Players: 5 +'):
        listofFindCommands.append(dict_Players_5)

basedictionary = {"$and": listofFindCommands}
return ApplyFoundFilters(FoundFilters, NonFilteredCollection,
                         basedictionary)  # This function return
```

```
def CheckSubstringMatches(filters, NonFilteredCollection):

    fullstring = filters
    # print("This is the Fullstring : " + fullstring)
    FoundFilters = list()

    dict_Players_2 = {"$and": [{"Min_Players": {"$lte": 2}}, {"Max_Players": {"$g
    dict_Players_3 = {"$and": [{"Min_Players": {"$lte": 3}}, {"Max_Players": {"$g
    dict_Players_4 = {"$and": [{"Min_Players": {"$lte": 4}}, {"Max_Players": {"$g
    dict_Players_5 = {"$and": [{"Min_Players": {"$lte": 5}}, {"Max_Players": {"$g


    for specificFilter in Allfilters:
        substring = specificFilter
        if fullstring.find(substring) != -1:
            # print("Found: " + substring)
            FoundFilters.append(substring)

        else:
            # print(substring + ": Not found")
```

```python
def ApplyFoundFilters(FoundFliters, NonFilteredCollection, basedictionary):
    # NonFilteredCollection is boardgame collections. This is used as a supe
    filteredCollection = db["FinalFiltered"]
    filteredCollection.drop()  # drop entire collection
    filteredCollection = db["FinalFiltered"]

    if len(FoundFliters) == 0:
        # if no filters then just leave.
        return NonFilteredCollection
    # .find({"$and": [filters]})
    # print("Hit before filter loop")
    cur = NonFilteredCollection.find(basedictionary)
```

Create and drop DB and cycle through all elements that match filter

```html
<div class="dropdown" style="margin-left:60px;">
  <!-- Basic dropdown -->
  <button class="btn btn-primary dropdown-toggle " type="button" data-toggle="dropdown"
          aria-haspopup="true" aria-expanded="false" id = "FiltersForGames" >Filters for Games</button>
  <div class="dropdown-menu" aria-labelledby="dropdownMenuButton">

    <a class="dropdown-item" href="#">
      <!-- Default unchecked -->
      <div class="custom-control custom-checkbox">
        <input type="checkbox" class="custom-control-input" id="checkbox1"   onclick="loadNewFilter('1_Hou
        <label class="custom-control-label" for="checkbox1">1 Hour or More</label>
      </div>
      </a>

    <a class="dropdown-item" href="#">
      <div class="custom-control custom-checkbox">
        <input type="checkbox" class="custom-control-input" id="checkbox2"  onclick="loadNewFilter('1_Hou
        <label class="custom-control-label" for="checkbox2">1 Hour or Less </label>
      </div>
    </a>
    <a class="dropdown-item" href="#">
      <div class="custom-control custom-checkbox">
        <input type="checkbox" class="custom-control-input" id="checkbox3"  onclick="loadNewFilter('30_Mi
        <label class="custom-control-label" for="checkbox3">30 Minutes or Less</label>
      </div>
    </a>
```

Hard-coded filter selections

Code Snippets After:

```python
def SelectFilter(filter, NonFilteredCollection):
    if filter == "nofilters":
        return NonFilteredCollection

    listofFindCommands = []

    if (filter.split('_')[0]) == 'Players:':
        numberOfPlayers = int((filter.split('_')[1]).strip('+'))  # Necessary for 5 + case
        listofFindCommands.append({"$and": [{"Min_Players": {"$lte": numberOfPlayers}},
                                            {"Max_Players": {"$gte": numberOfPlayers}}]})
```

```python
            if (filter.split('_')[5]) == 'More':
                return ApplyFoundFilters(NonFilteredCollection, {"Average_Playtime": {"$gte": int(filter.spli
            else:
                return ApplyFoundFilters(NonFilteredCollection, {"Average_Playtime": {"$lte": int(filter.spli
    return ApplyFoundFilters(NonFilteredCollection, null)  # This function returns filtered collection


def ApplyFoundFilters(NonFilteredCollection, basedictionary):
    return NonFilteredCollection.find(basedictionary)# This collection should be totally filtered
```

No more creating DB and Dropping DB, just return cursor

```html
<div class="column" style="justify-content: left; padding: 0% 1%">
    <div class="dropdown">
        <button class="btn btn-primary dropdown-toggle" type="button" data-toggle="dropdown" aria-haspopup="true" aria
            {{ filter_title }}</button>
        <div class="dropdown-menu" aria-labelledby="dropdownMenuButton">
            {% for filter in filters_list %}
            <a class="dropdown-item ">
                <div class="radio">
                    <label>
                        {% if filters == filter %}
                    <input checked type="radio" name="filter-radio" onclick="loadNewFilter('{{ filter }}')">
                        {% else%}
                    <input type="radio" name="filter-radio" onclick="loadNewFilter('{{ filter }}')">
                        {% endif %}
                    {{ filter.replace("_", " ") }}</label>
                </div>
            </a>
            {% endfor %}
            <a class="dropdown-item" href="#">
                <div class="custom-control custom-checkbox">
                    <button class="btn btn-link" title="list2" onclick="window.location.href = link('{{page_route}}',
                </div>
            </a>
```

Filter selections generated in a for-loop