CS 5551 First Increment Report

Adam Carter
Class ID: 9, Project Group ID: 1
CS 5591 – Spring 2015

I. Introduction

This is the summary report of the first iteration of work performed on the Terrapin Collection Manager system. This system proposes to implement a framework for organizing and managing a user's Board Game collection. The full background of this project can be found in the Project Proposal Document [1] submitted previously.

For this first iteration, I focused my work on the foundational services, primarily the Database and REST Service tiers. These were the areas I am most familiar with, and are necessary stepping-stones for system functionality. Many of the details behind the architecture of this system were discussed previously in my Project Plan [2], and for the sake of brevity will not be repeated here except to discuss changes or additions.

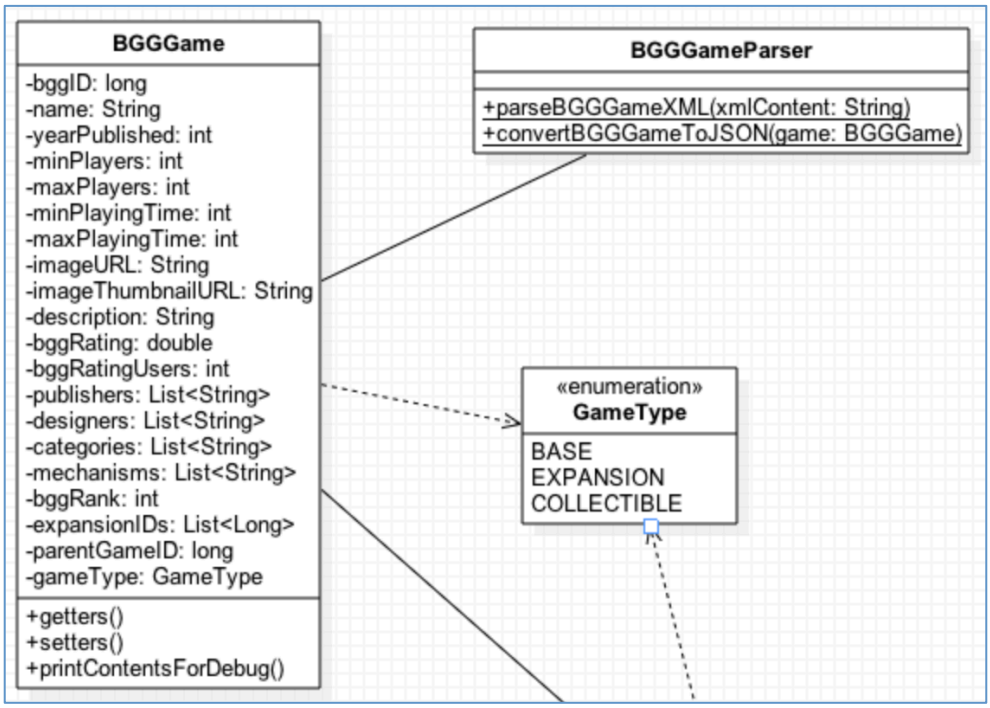
The database tier is being implemented using the NoSQL database system MongoDB [3]. Database connectivity is supported by the Mongo Java Driver library [4]. The REST Services tier is being implemented in Java using the Spring REST framework [5]. All Java projects are building using Apache Maven [6]. Testing has been implemented using JUnit. REST testing is additionally supported by the Spring REST Assured and MVC Mock testing Frameworks [7]. All JUnit tests can be run automatically through the Maven Surefire Plugin [8], and additional test reporting such as Code Coverage reports are implemented using the Atlassian Clover plugin [9]. GitHub [10] is being used as the source code repository. No front-end products were produced as part of this iteration.

II. Design Overview

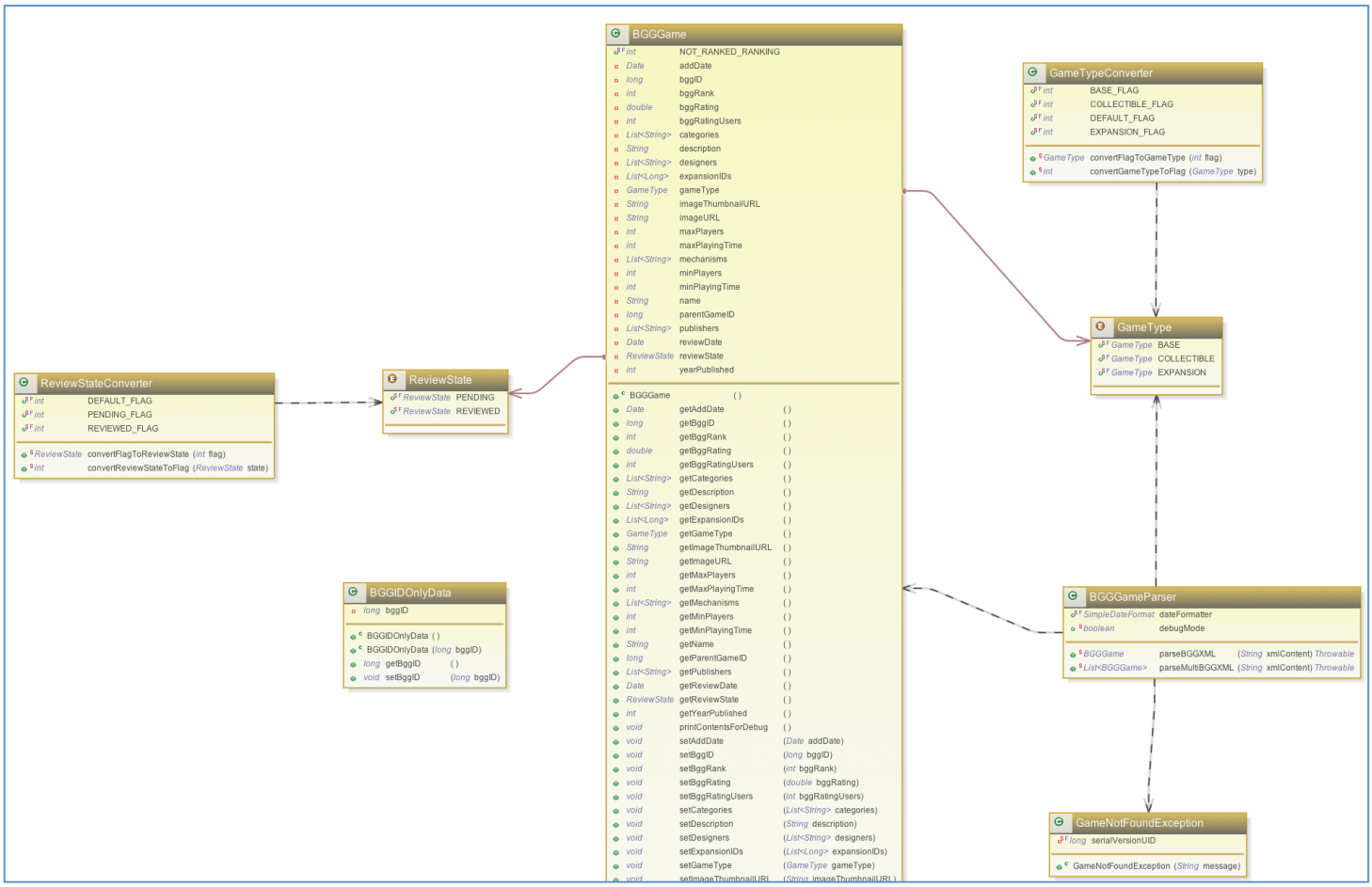
Since many of the aspect of the initial design were laid out in my project plan, I will instead talk about some of the key concepts that have changed.

One of the key issues I encountered from a design perspective is the review process for building my index of content. I needed a way to distinguish between games that had been newly uploaded, those that had been approved, and those that had been rejected as either broken entries or (in the case of the Price Data objects) were not actually board games. In order to accomplish this, I set out to develop a two-stage process with new metadata columns added to the appropriate data models.

Presented below is a contrast of the original design, and how the model had to be expanded to accommodate the new approach.



The original object model from the project plan



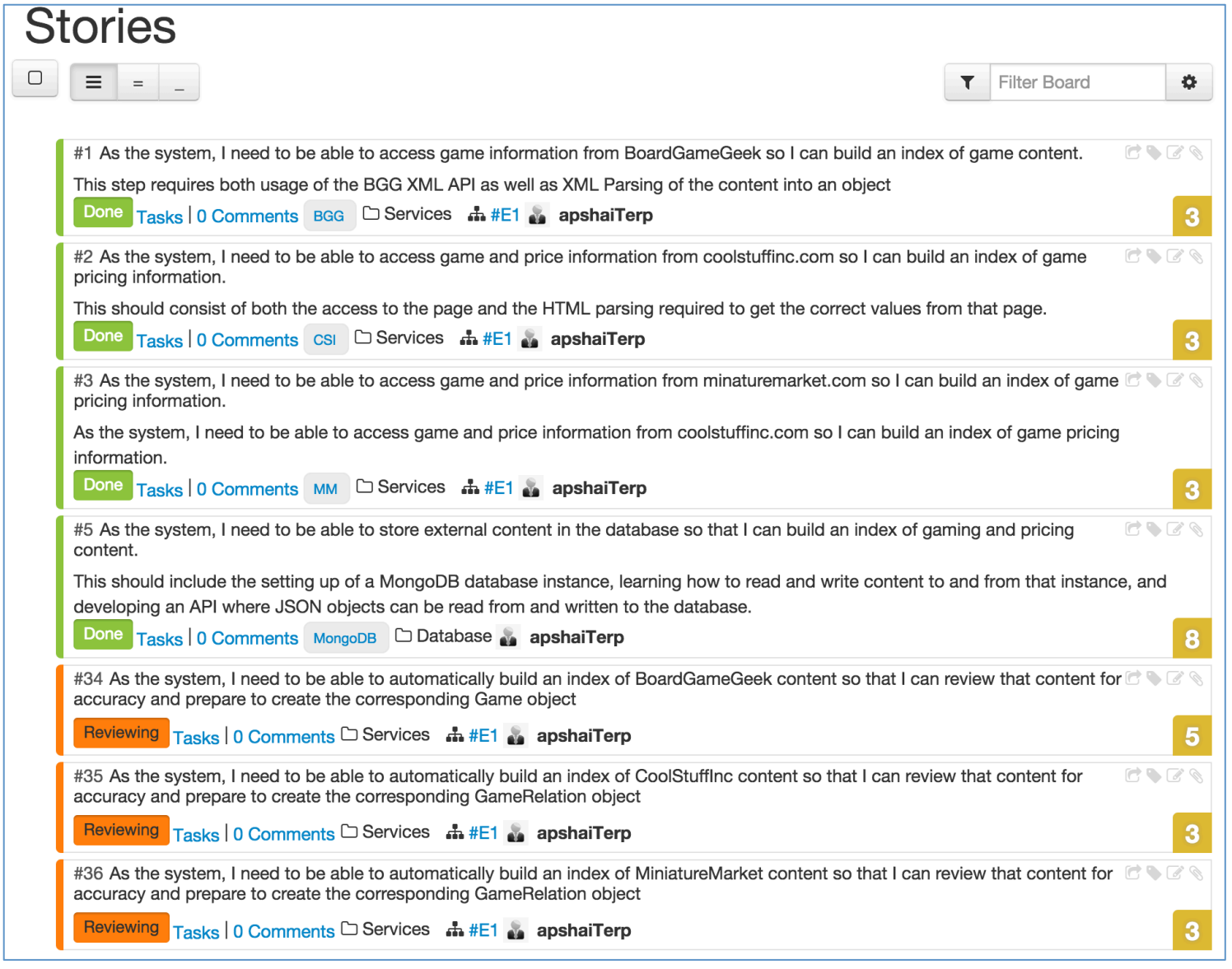
Updated view of Model. Full image at http://tinyurl.com/q2xarto.

To Summarize, three new elements needed to be added: A Date value to indicate when we originally scanned this object, a ReviewState value, which should consist of the following values {PENDING | REVIEWED | REJECTED}, and a second Date to indicate when the review took place.

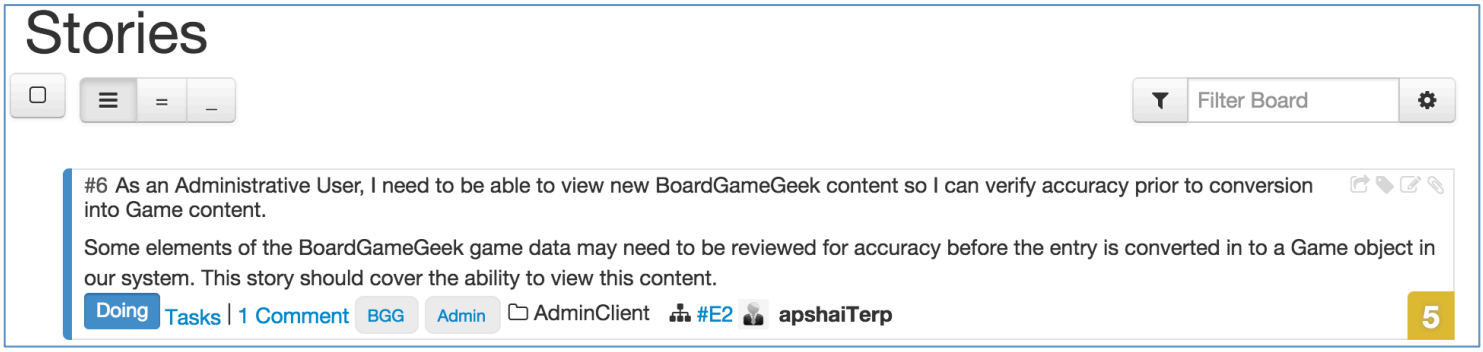
This also led to my creation of a new architectural element I am referring to as an Agent. This process is a Thread-based process that will scan the remote sources for new content and automatically add them in a PENDING state to my database. This concept has been applied to BoardGameGeek content as well as CoolStuffInc and MiniatureMarket price data.

When considering the first iteration, my goal from a design perspective was to create a functioning REST service, implement the required database CRUD support, and work to implement the Agent to pull external content into my new index.

From a story perspective, the following stories were worked on this iteration:



In addition, the following story was slated for design and implementation this iteration, but while begun, the majority of the work is not yet complete, so I moved it already into the second iteration:



These stories focus exclusively on the Database, REST Service, and Agent components of my architecture.

Finally, from a design and project organization point of view, I have broken my project into multiple sub-projects, which is appropriate for their deployment strategy. The highly modularized approach allows for much tighter testing and implementation strategies, and follows best practice principles around component decomposition. I will present a brief summary of the projects that were created and how they satisfy the architectural requirements of the project.

- **ac-games-pojos** – https://github.com/apshaiTerp/ac-games-pojos
  This project contains the basic data objects modeled in the original plan (i.e. BGGGame, CoolStuffIncPriceData, etc), as well as the necessary manipulation methods required to use these objects, specifically our HTML/XML parsing tools (i.e. BGGGameParser). This was made a standalone project so that it could be consumed by all required architectural elements. In practice, all other projects to this point are dependent on the objects and utilities contained by this project.

- **ac-games-db** – https://github.com/apshaiTerp/ac-games-db
  This project contains the Interface definition for all database operations implemented to this point. Because I did not want to pin myself to Mongo completely, I abstracted the interface into this project so that I could potentially create other implementations going forward. There is no realized code in this project, only interfaces that define what operations must be available by any implementing project.

- **ac-games-db-mongo** – https://github.com/apshaiTerp/ac-games-db-mongo
  This is the MongoDB implementation of all required database operations defined by the ac-games-db project. This project uses a Factory Creation pattern to allow for Singleton instantiation of the Database connection required. This project contains all the logic required both to interact with the database as well as convert from Java POJOs to MongoDB

JSON/BSON objects and back. This project relies on the Mongo Java Driver API. This project can be consumed by any other application tier requiring direct access to the database.

- **ac-games-restservice-spring** – https://github.com/apshaiTerp/ac-games-restservice-spring
  This is the REST Service project, implemented using the Spring Framework. This project requires a number of external APIs unique to the Spring Framework and will be deployed as a standalone service, so it requires it's own project. The Spring framework is an annotation based framework, so a sample REST mapping might look like this:

```
@RequestMapping(method = RequestMethod.GET, produces="application/json;charset=UTF-8")
public Object getBGGData(@RequestParam(value="bggid") long bggID,
                         @RequestParam(value="source", defaultValue="bgg") String source,
                         @RequestParam(value="batch", defaultValue="1") int batch) {
```

  The Spring Framework uses Controllers to implement a given REST handle, so each of my service offerings will contain annotations like this:

```
@RestController
@RequestMapping("/external/bggdata")
public class BGGDataController {
```

  This project can be executed as a standalone executable JAR, or a web service WAR file which can be deployed to almost any webservice host, such as Jetty or Tomcat.

- **ac-games-agent** – https://github.com/apshaiTerp/ac-games-agent
  This is the agent component described above. While still early in it's design, the basic concept of this project is to contain a ThreadPool of recurring tasks that will query the external data sources for new content and automatically push that content into the database in a PENDING state for review.

  Going forward, this project may also attempt to automatically move PENDING projects with clear and complete item definitions into a REVIEWED or REJECTED state, though that design work has not been attempted as part of this iteration and will be added as a new story to the backlog.

Finally, while testing is part of design, I will save that discussion for the Testing Overview section, though at this point it is worth mentioning that all projects are designed to used the Maven Surefire plugin to facilitate automated JUnit test execution bound to the appropriate maven build stages, meaning each project will not build a final executable without successfully executing all automated tests. I am also using Atlassian Clover to include that test data as part of the site build for each maven project as well as provide Code Coverage statistics that extend to both source code and test code. This will again be discussed in more detail during the Testing Overview section.

III. Implementation Overview

To discuss implementation, I will also approach this in terms of the various sub-projects, but at a high level, I have implemented full support for three of my required services:

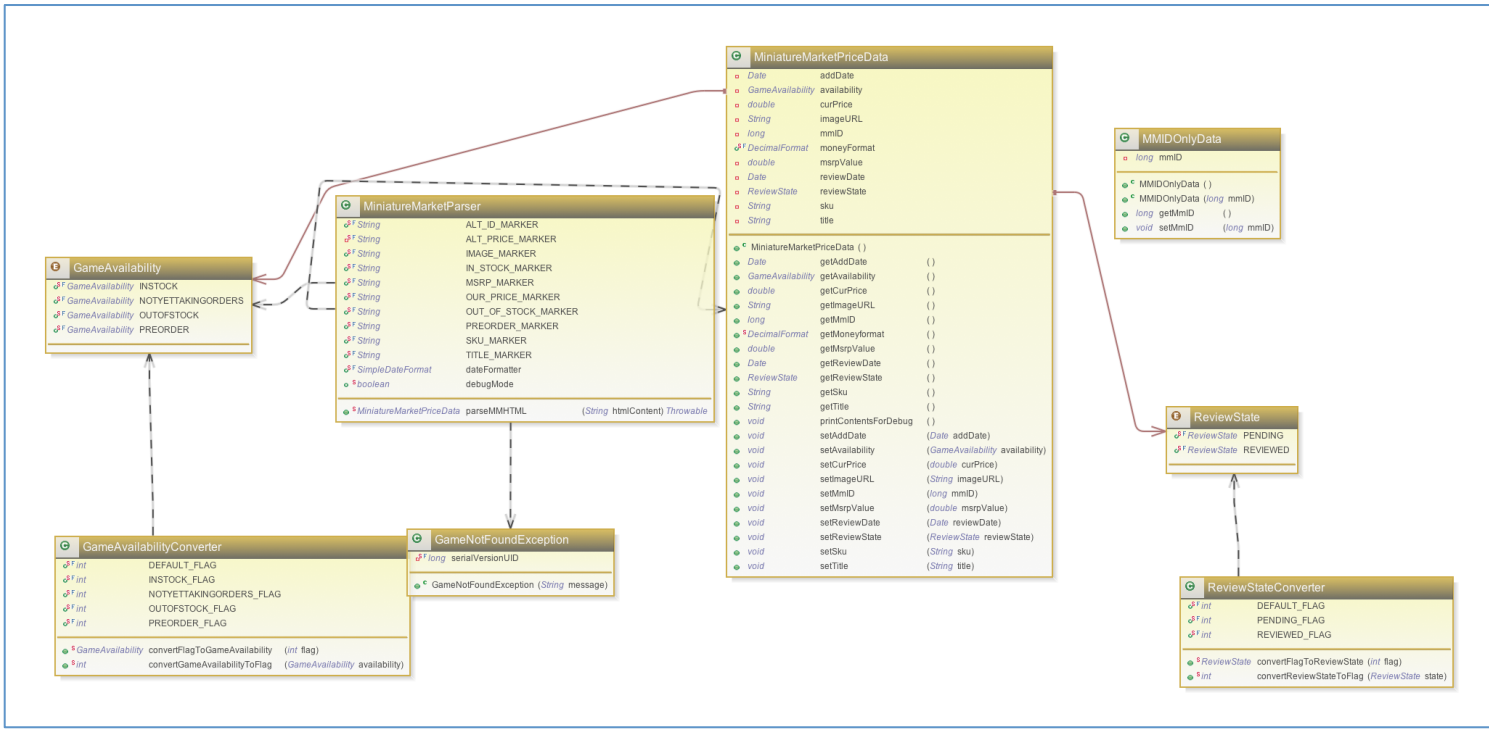
```
/external/bggdata
/external/csidata
/external/mmdata
```

These services have been fully implements for all CRUD operations, meaning they now support GET, PUT, POST, and DELETE operations. They are fully connected to the database tier, so users can flex between reading from the external source or our stored database content.

What follows will be a discussion of the work done in each project this iteration, along with some modeling generated to present the content.

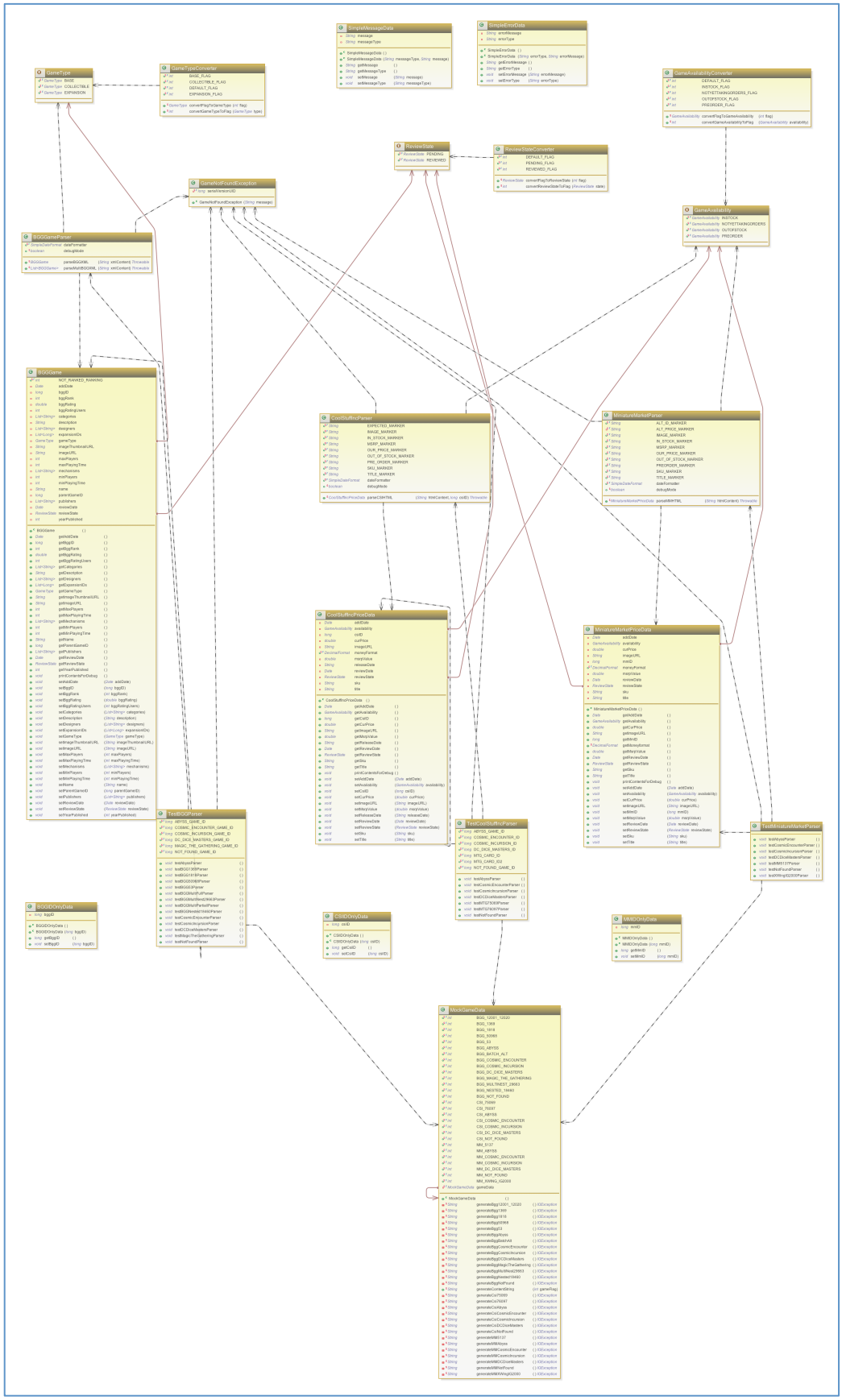
- **ac-games-pojos**

  We will start with a class diagram for one of our feature sets, MinatureMarket data:



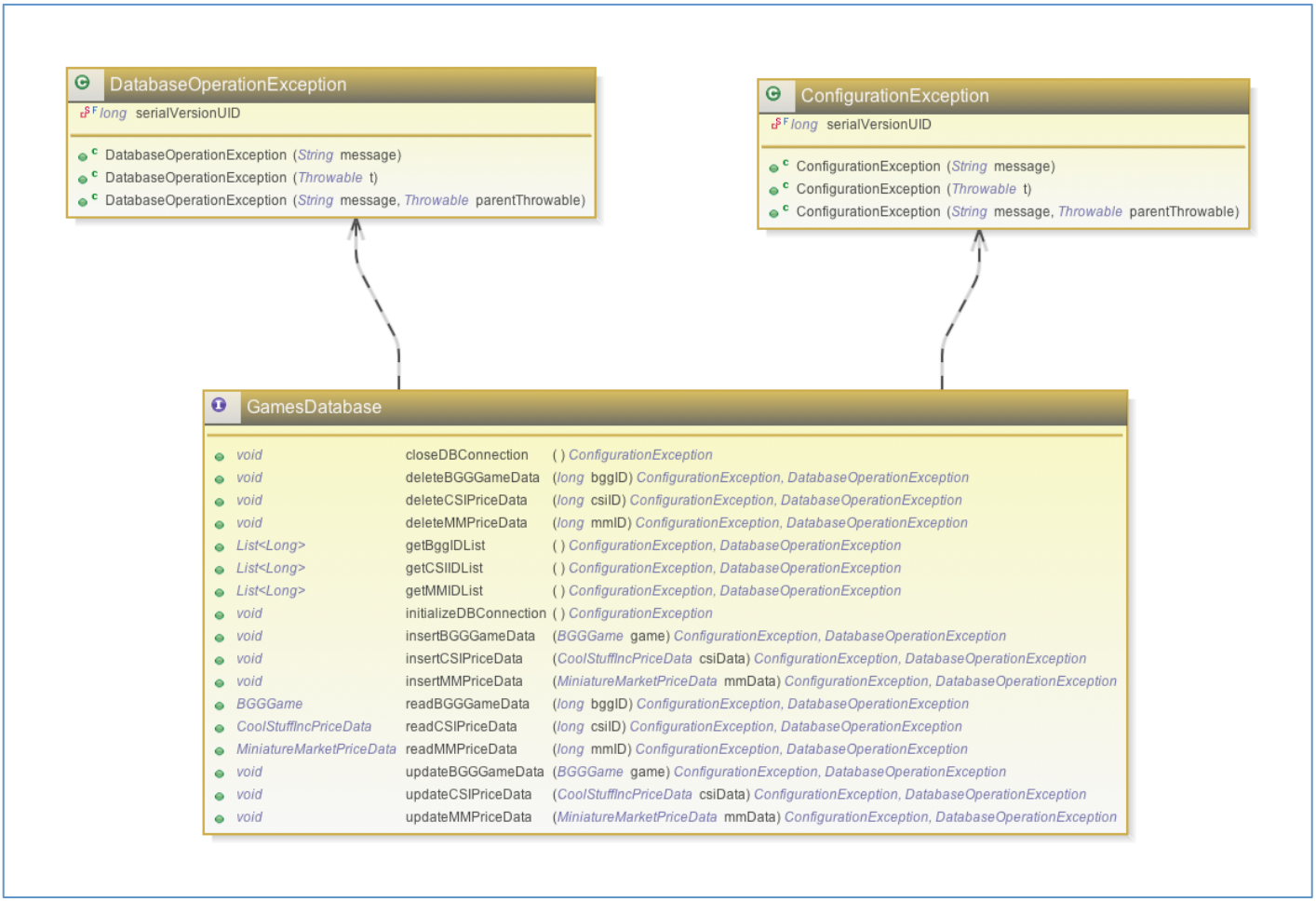
This class diagram and all others for this project can be found at:
https://github.com/apshaiTerp/ac-games-pojos/tree/master/src/models

By way of discussion, in addition to the expected Base data object and Parser object anticipated in the original design, we introduced a few new elements to the model. There is the review state, with it's associated Enum. Also, each Enum requires a converter object that can convert the Enum into an integer flag and back due to limitations of the BSON conversion process MongoDB uses. This class currently provides full support for the three basic objects, BGGGame, CoolStuffIncPriceData, and MiniatureMarketPrice data. The full class diagram is provided below:


SimpleMessageData
SimpleErrorData
GameType
GameTypeConverter
GameAvailabilityConverter
ReviewState
ReviewStateConverter
GameNotFoundException
GameAvailability
BGGGameParser
BGGGame
CoolStuffIncParser
MiniatureMarketParser
CoolStuffIncPriceData
MiniatureMarketPriceData
TestBGGParser
TestCoolStuffIncParser
TestMiniatureMarketParser
BGGIDOnlyData
CSIDOnlyData
MMIDOnlyData
MockGameData

It can be found at the link provided above.

- **ac-games-db**
  This class is meant to be a simple interface. A class diagram reflecting the currently implemented elements is presented here:



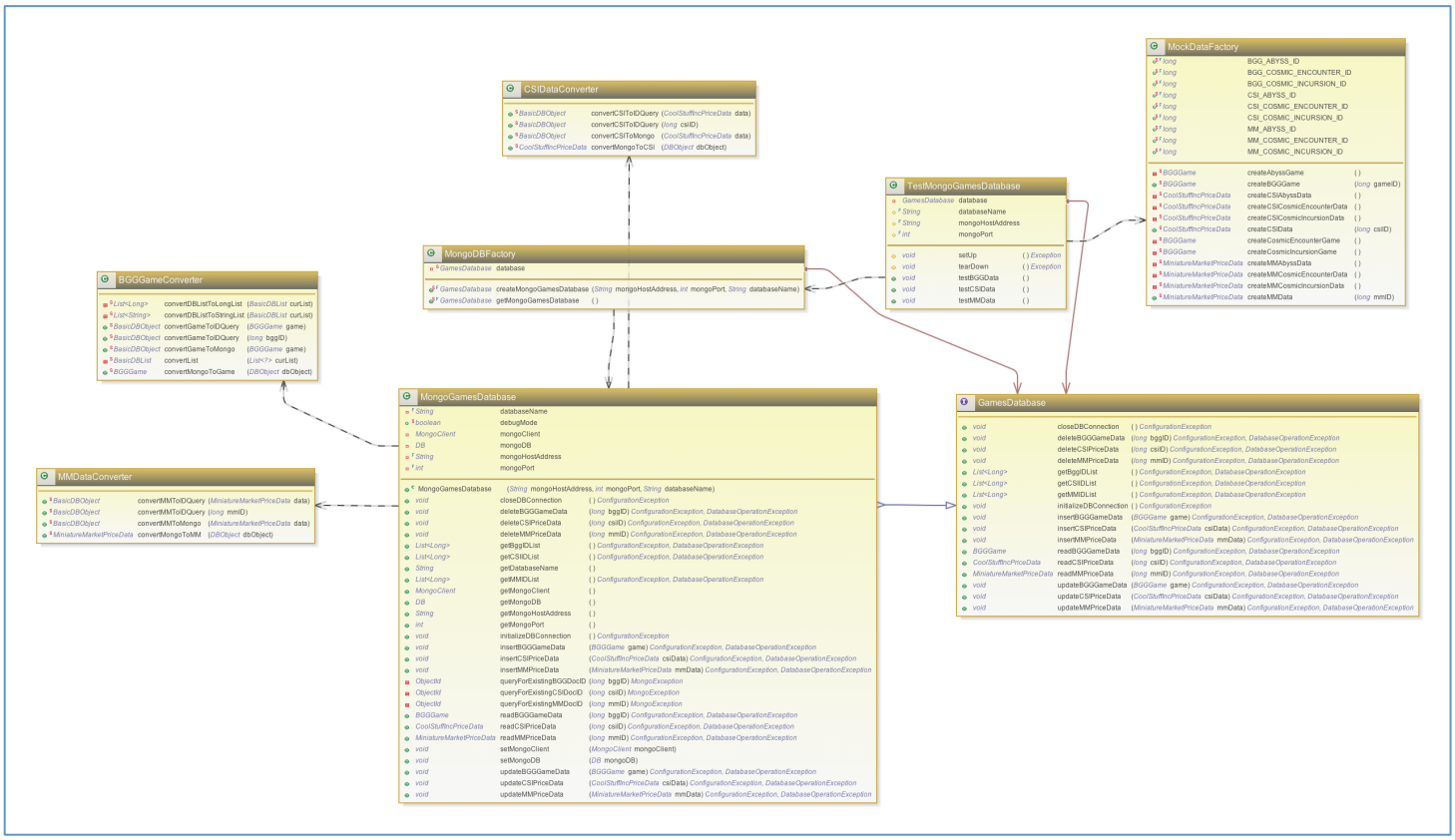
  As you can see, it's a simple interface with defined Exceptions that defines what a database implemention should be able to do.

  This diagram can be found in the project information stored here:
  https://github.com/apshaiTerp/ac-games-db/blob/master/src/model/GamesDatabase.png

- **ac-games-db-mongo**
  This project contains the MongoDB implementation of our Database CRUD operations for our three external datasets. The Database connection is itself protected using a Factory Creation pattern implemented in the MongoDBFactory.java object. The GamesDatabase Interface is implemented by MongoGamesDatabase. Each data object also implements its own DataConverter object that handles the conversion from POJO to BSON, which is not quite a straightforward conversion like JSON. This process requires the manual conversion

of all Enum fields into static values, since even though Enums are serializable by default in Java, that conversion is not supported by the BSON conversion protocols.

Presented here is the diagram of the current makeup of this project:

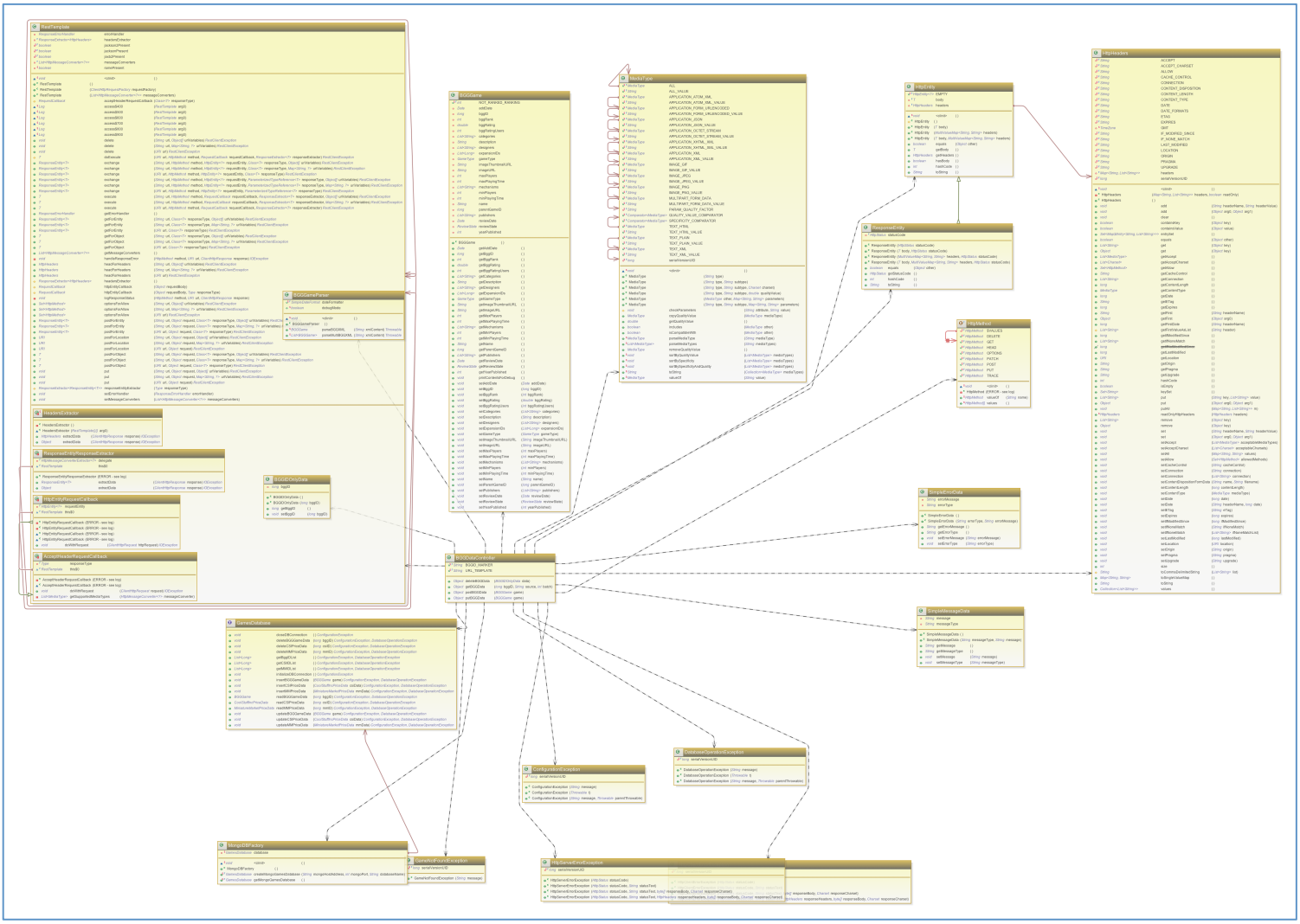


This diagram as well as breakdowns by feature set can be found here:
https://github.com/apshaiTerp/ac-games-db-mongo/tree/master/src/model

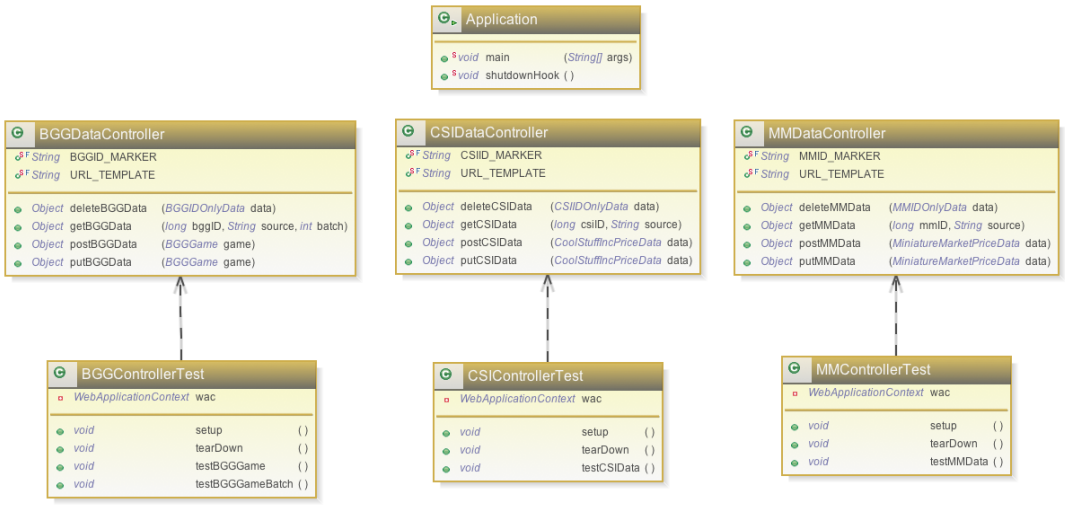
- **ac-games-restservice-spring**
  This project is our REST service implementation using Spring. GET, PUT, POST, and DELETE operations are supported for the three services discussed above. The basic concept behind the services are fairly straight forward, but this project relies heavily on the dependent projects, most particularly the database implementation and the external data source parsing to function correctly.

  In practice, it has been the parsing of external data that has been the most difficult to manage from a server stability perspective. Inconsistent data sets, incomplete data, and external server availability issues have helped me to implement rather robust error handling of potential fail points to prevent the server from crashing when encountering incomplete data.

  Thanks to the Spring Framework, it was quite easy to get a simple skeleton up and running quickly. Presented below is a class model of the core model and it's dependencies:



Above: BGGGameController and Dependencies, Below: Basic Controller and Test structure

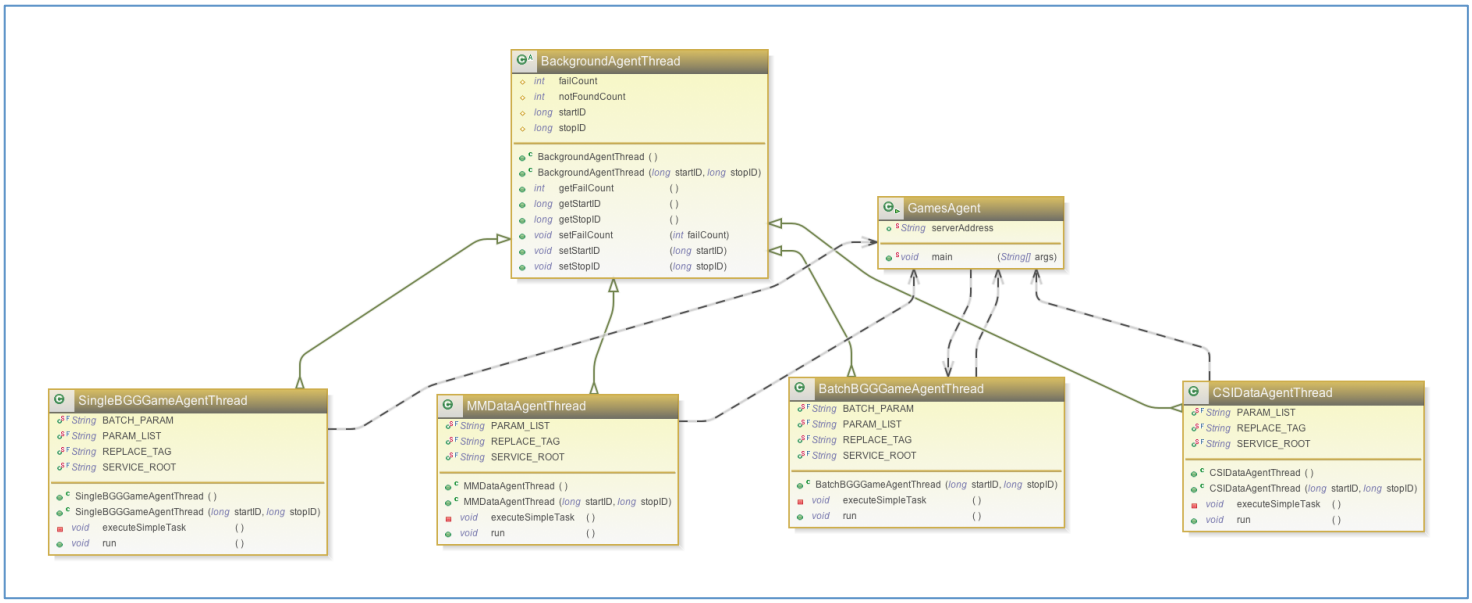


These diagrams for each feature set can be found here:
https://github.com/apshaiTerp/ac-games-restservice-spring/tree/master/src/model

- **ac-games-agent**
  This project is still in the crudest state of all the components implemented during this iteration. While Threads have been built that can read through external data sources, developing functional code here has been where most of the external data source issues have been discovered, including a nifty feature where the BoardGameGeek XML API would lock me out of the system for 60 seconds if I submitted requests too quickly. This forced me to change my approach to open up my external/bgggame GET request to allow for requesting games in batches, which both allows me to process data faster, as well as prevents the timeout issue.

  This project is still incomplete, but does have limited, manually configured Threads that are capable of reading external data through my REST service, then submitting that data in a PENDING review state again through my REST service.

  This is a brief class diagram of the implemented elements so far.



All of these projects are built using Apache Maven, meaning successful builds result in jar files that can be imported to other projects or run standalone as needed. For example, the ac-games-restservice-spring project builds an executable jar that launches my REST service in localhost with no other configuration needed.

I was also able to install a local MongoDB instance that runs in localhost.

IV. Testing Overview

I mentioned previously that all projects (with the exception of ac-games-db, which contains no actual executable code) implement the Maven Surefire Plugin and Atlassian Clover plugins to facilitate automated JUnit testing and Code Coverage reporting. This is a valuable tool, since tests can be run frequently to ensure that any new changes remain passive and do not otherwise impact features that are working correctly.

One major reason I chose to use separate, highly modularized projects was because of Unit Testing. With so many different platforms, being able to segment Unit tests to be able to test individual slices of my architecture without at this point testing the whole is very valuable in early development.

Before discussing individual testing considerations, it's worth mentioning that I am only situationally using TDD techniques. I find this to be most valuable when I am encountering an error or defective data. In these circumstances, it is easy to write the test that tests for correct behavior, then fix the code to achieve test completion. In general, however, I tend to write a module or method then write the test for that module or method.

Since automated testing has only been implemented for 3 of the eligible 4 projects, I will only present data from those projects, though all such data is visible through the maven sites for each project.

Because this code is not yet ready for full deployment yet, I have uploaded the final end-states, including test evidence and maven site, into the following location on GitHub:
https://github.com/apshaiTerp/cs-5551-documentation/tree/master/maven%20sites

I will present screen caps from both the surefire report data and the clover coverage report for each of the three projects below, which are visible as well through the maven site listed above.

- **ac-games-pojos**

  One major tool to facilitate testing for this project was to save off expected output from our external sites, either actual HTML pages or XML content, so that I would ensure I was testing against 'real' data. Even best efforts were incomplete, so as I would encounter pages that broke my parsing algorithm, I would save off that data, write a new Unit Test, then work until I figured out why I wasn't parsing correctly and fix it.

  One issue with true code coverage statistics in this case is that I did not write simple test cases to test the Enum coverters, though they are being tested in a more contextual fashion in other projects. My goal is never 100% code completion, and for the sake of time, these sacrifices are an acceptable tradeoff, since I know those objects are being tested, if not in a simple Unit Test here.

  Presented below are the screen caps for the Surefire report the Clover reports respectively.

Surefire Report

Summary

[Summary] [Package List] [Test Cases]

Tests	Errors	Failures	Skipped	Success Rate	Time
28	0	0	0	100%	0.545

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

Package List

[Summary] [Package List] [Test Cases]

Package	Tests	Errors	Failures	Skipped	Success Rate	Time
com.ac.games.data.mock	28	0	0	0	100%	0.545

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

com.ac.games.data.mock

	Class	Tests	Errors	Failures	Skipped	Success Rate	Time
	TestBGGParser	14	0	0	0	100%	0.454
	TestCoolStuffIncParser	7	0	0	0	100%	0.063
	TestMiniatureMarketParser	7	0	0	0	100%	0.028

POJO library for Collection Management project 0.1.0-SNAPSHOT Clover Coverage Report

Dashboard
Coverage Reports
Coverage (Aggregate)
Test Code (Aggregate)
Test Results

Application Packages

com.ac.games.data *(65.9%)*
com.ac.games.data.parser *(89.3%)*
com.ac.games.exception *(100%)*
com.ac.games.rest.message *(0%)*

Classes **Tests** **Results**

Class	Coverage
BGGGame	*(85%)*
BGGGameParser	*(92.3%)*
BGGIDOnlyData	*(0%)*
CSIIDOnlyData	*(0%)*
CoolStuffIncParser	*(89.2%)*
CoolStuffIncPriceData	*(80.2%)*
GameAvailabilityConverter	*(0%)*
GameNotFoundException	*(100%)*
GameTypeConverter	*(0%)*
MMIDOnlyData	*(0%)*
MiniatureMarketParser	*(82%)*
MiniatureMarketPriceData	*(74.3%)*
ReviewStateConverter	*(0%)*
SimpleErrorData	*(0%)*
SimpleMessageData	*(0%)*

Clover Coverage Report - POJO library for Collection Management project 0.1.0-SNAPSHOT
Coverage timestamp: Wed Feb 25 2015 15:43:15 CST
App Test Results Clouds
Overview Package File
FRAMES NO FRAMES SHOW HELP

Statistics for project Clover database Wed Feb 25 2015 15:43:13 CST:

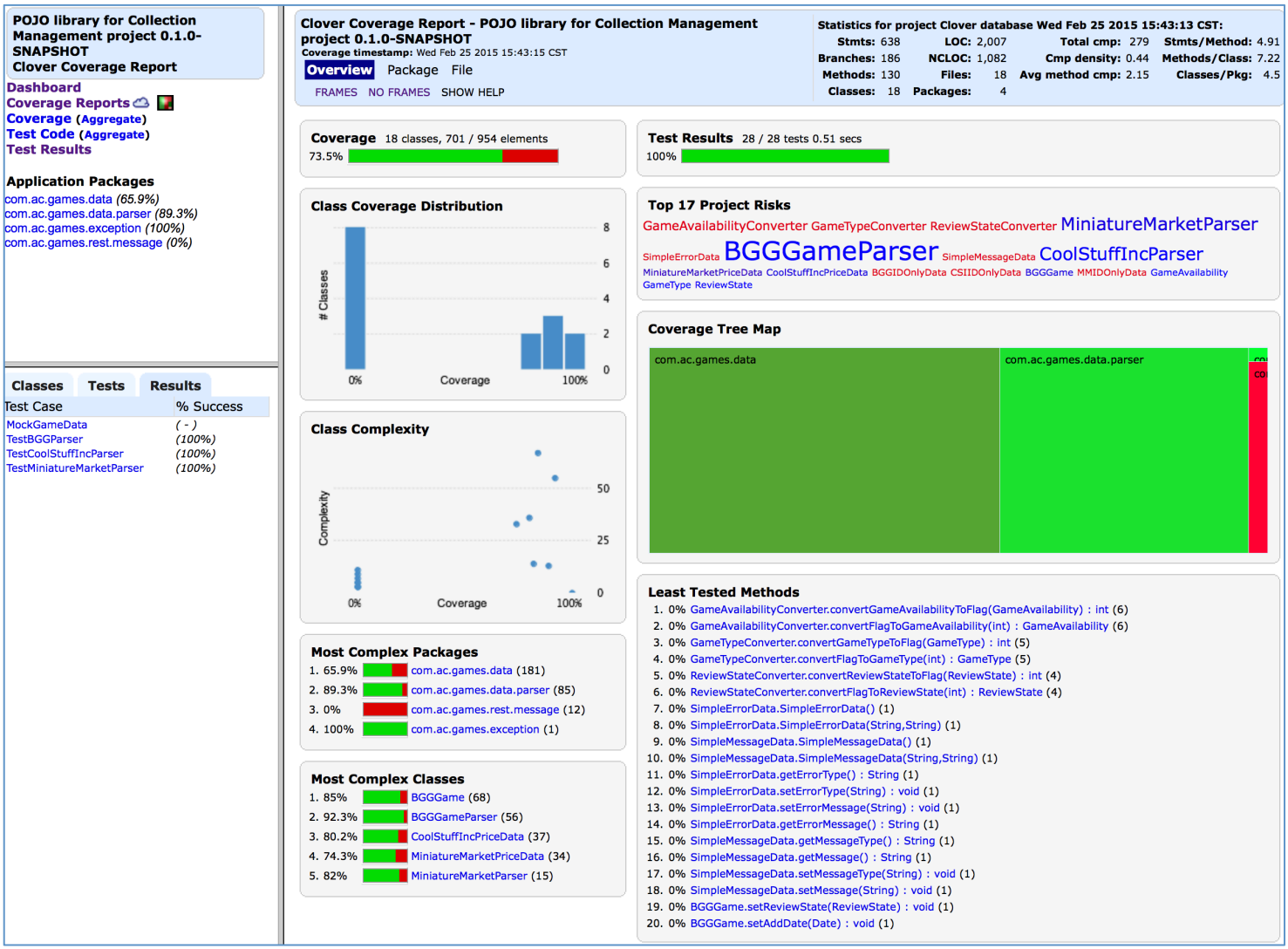
Stmts:	638	**LOC:**	2,007	**Total cmp:**	279	**Stmts/Method:**	4.91
Branches:	186	**NCLOC:**	1,082	**Cmp density:**	0.44	**Methods/Class:**	7.22
Methods:	130	**Files:**	18	**Avg method cmp:**	2.15	**Classes/Pkg:**	4.5
Classes:	18	**Packages:**	4				

	Packages	% Filtered	Average Method Complexity	Uncovered Elements	TOTAL Coverage
Clover database Wed Feb 25 2015 15:43:13 CST	4	0%	2.15	253	73.5%

Package	Files	% Filtered	Average Method Complexity	Uncovered Elements	TOTAL Coverage
com.ac.games.rest.message	2	0%	1	28	0%
com.ac.games.data	12	0%	1.6	184	65.9%
com.ac.games.data.parser	3	0%	21.25	41	89.3%
com.ac.games.exception	1	0%	1	0	100%

Report generated by Clover Code Coverage v3.3.0
Wed Feb 25 2015 15:43:36 CST.

Clover: Commercial License registered to Cerner Corporation.



- **ac-games-db-mongo**
  For the Database testing, I wanted to structure a self-contained test that would test all expected database operations for each dataset. Each of these external tests implements the following steps:

  1. Insert Cosmic Encounter
  2. Insert Cosmic Incursion
  3. Read Cosmic Encounter and Verify
  4. Read Cosmic Incursion and Verify
  5. Reinsert Cosmic Encounter
  6. Upsert Abyss
  7. Read Cosmic Encounter and Verify
  8. Read Abyss and Verify
  9. Modify Abyss Data
  10. Update Abyss

11. Read Abyss and Verify
12. Run the IDs select and verify all three games found
13. Delete Cosmic Encounter and Cosmic Incursion
14. Read Nothing for two games, verify Abyss still exists
15. Delete Abyss
16. Test Complete

As such, there are a small number of tests, but those tests before a great deal of work.

Presented below are the screen caps for the Surefire report the Clover reports respectively:

Surefire Report

Summary

[Summary] [Package List] [Test Cases]

Tests	Errors	Failures	Skipped	Success Rate	Time
3	0	0	0	100%	1.282

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

Package List

[Summary] [Package List] [Test Cases]

Package	Tests	Errors	Failures	Skipped	Success Rate	Time
com.ac.games.db.test	3	0	0	0	100%	1.282

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

com.ac.games.db.test

	Class	Tests	Errors	Failures	Skipped	Success Rate	Time
	TestMongoGamesDatabase	3	0	0	0	100%	1.282

MongoDB Database Implementation 0.1.0-SNAPSHOT Clover Coverage Report

Dashboard
Coverage Reports
Coverage (Aggregate)
Test Code (Aggregate)
Test Results

Application Packages
com.ac.games.db *(75%)*
com.ac.games.db.mongo *(72.7%)*

com.ac.games.db.test

Classes | Tests | Results

Test Case	% Success
TestMongoGamesDatabase	*(100%)*

Clover Test Report - MongoDB Database Implementation 0.1.0-SNAPSHOT
Coverage timestamp: Wed Feb 25 2015 15:45:29 CST

Overview | Package | File | Test

FRAMES NO FRAMES SHOW HELP

Package	Tests	Fail	Error	Time	% Pass
com.ac.games.db.test	3	0	0	0.901	100%

Test Classes	Tests	Fail	Error	Time(secs)	% Pass
TestMongoGamesDatabase	3	0	0	0.901	100%

Report generated by Clover Code Coverage v3.3.0
Wed Feb 25 2015 15:45:46 CST.

Clover: Commercial License registered to Cerner Corporation.

MongoDB Database Implementation 0.1.0-SNAPSHOT Clover Coverage Report

Dashboard
Coverage Reports
Coverage (Aggregate)
Test Code (Aggregate)
Test Results

Application Packages
com.ac.games.db *(75%)*
com.ac.games.db.mongo *(72.7%)*

Classes | Tests | Results

Class	Coverage
BGGGameConverter	*(80%)*
CSIDataConverter	*(83%)*
MMDataConverter	*(79.8%)*
MongoDBFactory	*(75%)*
MongoGamesDatabase	*(64.8%)*

Clover Coverage Report - MongoDB Database Implementation 0.1.0-SNAPSHOT
Coverage timestamp: Wed Feb 25 2015 15:45:29 CST

Overview | Package | File

FRAMES NO FRAMES SHOW HELP

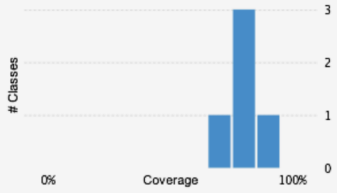
Statistics for project Clover database Wed Feb 25 2015 15:45:26 CST:

Stmts:	541	**LOC:**	1,132	**Total cmp:**	259	**Stmts/Method:**	12.02
Branches:	330	**NCLOC:**	664	**Cmp density:**	0.48	**Methods/Class:**	9
Methods:	45	**Files:**	5	**Avg method cmp:**	5.76	**Classes/Pkg:**	2.5
Classes:	5	**Packages:**	2				

Coverage 5 classes, 666 / 916 elements
72.7%

Test Results 3 / 3 tests 0.9 secs
100%

Class Coverage Distribution



Top 5 Project Risks

MongoGamesDatabase BGGGameConverter MMDataConverter MongoDBFactory CSIDataConverter

Coverage Tree Map

Class Complexity

Most Complex Packages
1. 72.7% com.ac.games.db.mongo (256)
2. 75% com.ac.games.db (3)

Most Complex Classes
1. 64.8% MongoGamesDatabase (141)
2. 80% BGGGameConverter (59)
3. 83% CSIDataConverter (29)
4. 79.8% MMDataConverter (27)
5. 75% MongoDBFactory (3)

Least Tested Methods
1. 0% MongoGamesDatabase.getMongoHostAddress() : String (1)
2. 0% MongoGamesDatabase.getMongoPort() : int (1)
3. 0% MongoGamesDatabase.getDatabaseName() : String (1)
4. 0% MongoGamesDatabase.getMongoClient() : MongoClient (1)
5. 0% MongoGamesDatabase.getMongoDB() : DB (1)
6. 0% MongoDBFactory.getMongoGamesDatabase() : GamesDatabase (1)
7. 41.2% MongoGamesDatabase.initializeDBConnection() : void (5)
8. 55% MongoGamesDatabase.updateBGGGameData(BGGGame) : void (7)
9. 55% MongoGamesDatabase.updateCSIPriceData(CoolStuffIncPriceData) : void (7)
10. 55% MongoGamesDatabase.updateMMPriceData(MiniatureMarketPriceData) : void (7)
11. 55.6% MongoGamesDatabase.deleteBGGGameData(long) : void (7)
12. 55.6% MongoGamesDatabase.deleteCSIPriceData(long) : void (7)
13. 55.6% MongoGamesDatabase.deleteMMPriceData(long) : void (7)
14. 60% MongoGamesDatabase.insertBGGGameData(BGGGame) : void (9)
15. 60% MongoGamesDatabase.insertCSIPriceData(CoolStuffIncPriceData) : void (9)
16. 60% MongoGamesDatabase.insertMMPriceData(MiniatureMarketPriceData) : void (9)
17. 61.3% MongoGamesDatabase.readBGGGameData(long) : BGGGame (10)
18. 61.3% MongoGamesDatabase.readCSIPriceData(long) : CoolStuffIncPriceData (10)
19. 61.3% MongoGamesDatabase.readMMPriceData(long) : MiniatureMarketPriceData (10)
20. 66.7% BGGGameConverter.convertGameToIDQuery(BGGGame) : BasicDBObject (2)

- **ac-games-restservice-spring**

Testing for the Spring framework was made possible using the Rest Assured Mock API along with the Spring MVC test harness. In this setting, it allows you to instantiate a single container (one service endpoint) for Unit Testing as opposed to launching the entire service. This is a great way to facilitate testing individual services in isolation.

For each of my three services, I implement a comprehensive test that attempts to demonstrate all 4 CRUD operations using a single game object. Here are the steps implemented by each test:

1. GET Abyss from BGG XML API through GET service
2. Validate that the Data looks correct
3. GET Abyss object again and store it off
4. POST Abyss to database through service
5. GET Abyss from database and validate
6. Modify Abyss Values
7. PUT Updated Abyss into database through service
8. GET Abyss from database and validate changes were affected
9. DELETE Abyss from database through service
10. GET Abyss and verify the Game Not Found message

Additionally, with the additional parameterization for the BoardGameGeek external API allowing rows to be fetched in batches, a separate test was written to test that this request behaves as expected.

Presented below are the screen caps for the Surefire report the Clover reports respectively:

Surefire Report

Summary

[Summary] [Package List] [Test Cases]

Tests	Errors	Failures	Skipped	Success Rate	Time
4	0	0	0	100%	9.34

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

Package List

[Summary] [Package List] [Test Cases]

Package	Tests	Errors	Failures	Skipped	Success Rate	Time
com.ac.games.rest.test	4	0	0	0	100%	9.34

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

com.ac.games.rest.test

	Class	Tests	Errors	Failures	Skipped	Success Rate	Time
	BGGControllerTest	2	0	0	0	100%	5.1
	CSIControllerTest	1	0	0	0	100%	1.486
	MMControllerTest	1	0	0	0	100%	2.754

ac-games-restservice-spring 0.1.0-SNAPSHOT
Clover Coverage Report

Dashboard
Coverage Reports
Coverage (Aggregate)
Test Code (Aggregate)
Test Results

Application Packages
com.ac.games.rest *(0%)*
com.ac.games.rest.controller *(52.2%)*

com.ac.games.rest.test

Classes	Tests	Results
Test Case		% Success
BGGControllerTest		*(100%)*
CSIControllerTest		*(100%)*
MMControllerTest		*(100%)*

Clover Test Report - ac-games-restservice-spring 0.1.0-SNAPSHOT
Coverage timestamp: Wed Feb 25 2015 15:47:04 CST

Overview **Package** File Test

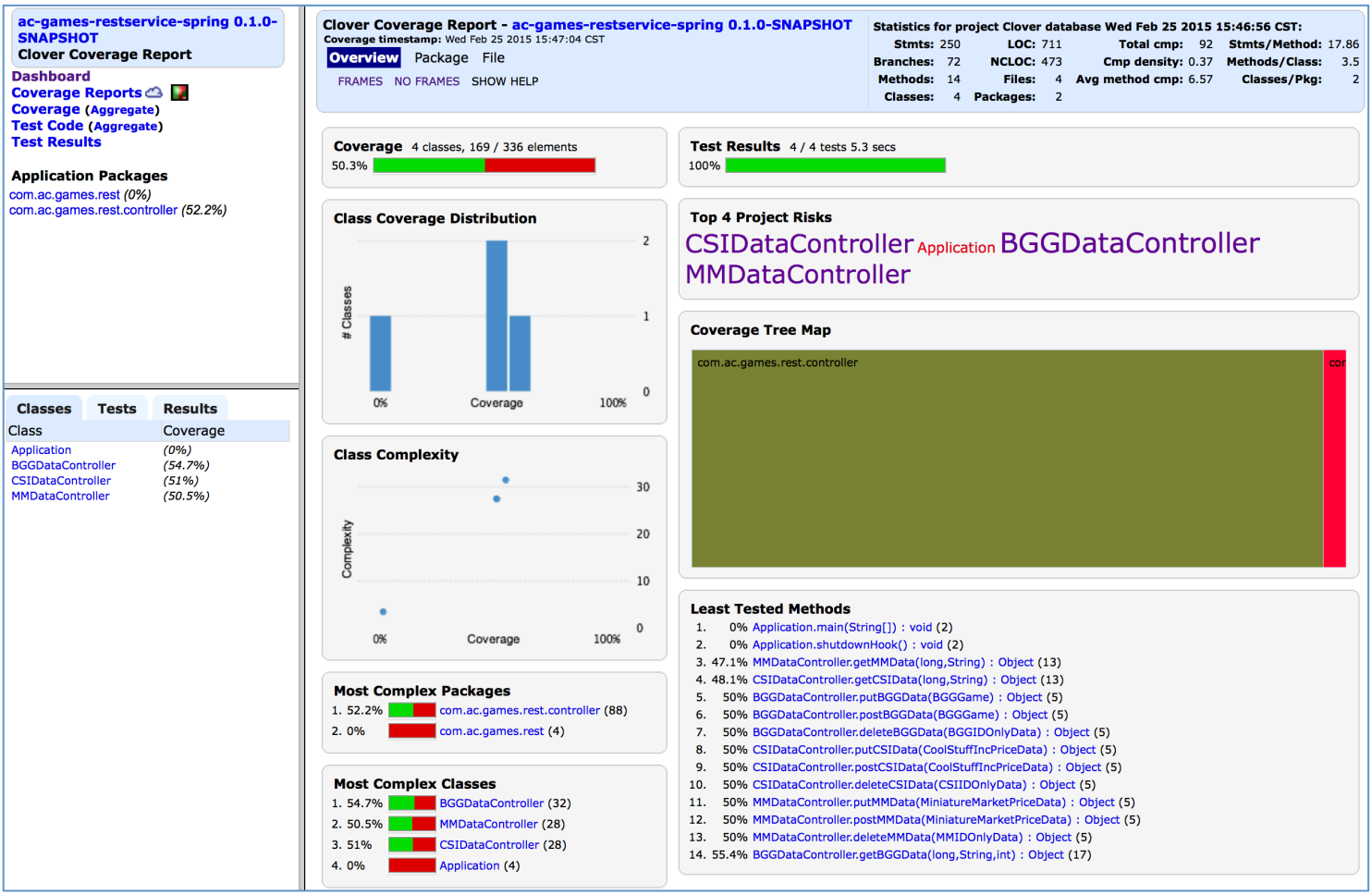
FRAMES NO FRAMES SHOW HELP

Package	Tests	Fail	Error	Time	% Pass
com.ac.games.rest.test	**4**	0	0	5.297	100%

Test Classes	Tests	Fail	Error	Time(secs)	% Pass
BGGControllerTest	**2**	0	0	2.934	100%
CSIControllerTest	**1**	0	0	1.147	100%
MMControllerTest	**1**	0	0	1.216	100%

Report generated by Clover Code Coverage v3.3.0
Wed Feb 25 2015 15:47:43 CST.

Clover: Commercial License registered to Cerner Corporation.



V. Deployment Overview

The ScrumDo link to this iteration can be found here:
https://www.scrumdo.com/projects/project/terrapin-collection-manager/iteration/121581

At this time, there are 3 stories still in Review, which will remain in that state until I can verify that I have addressed all data parsing issues with content parsing.

Because there are no front-end components at this time, the only deployable artifacts are Java jars. This can be accomplished by checkout out each project from GitHub and running the following maven command from the command prompt:

```
mvn clean install site
```

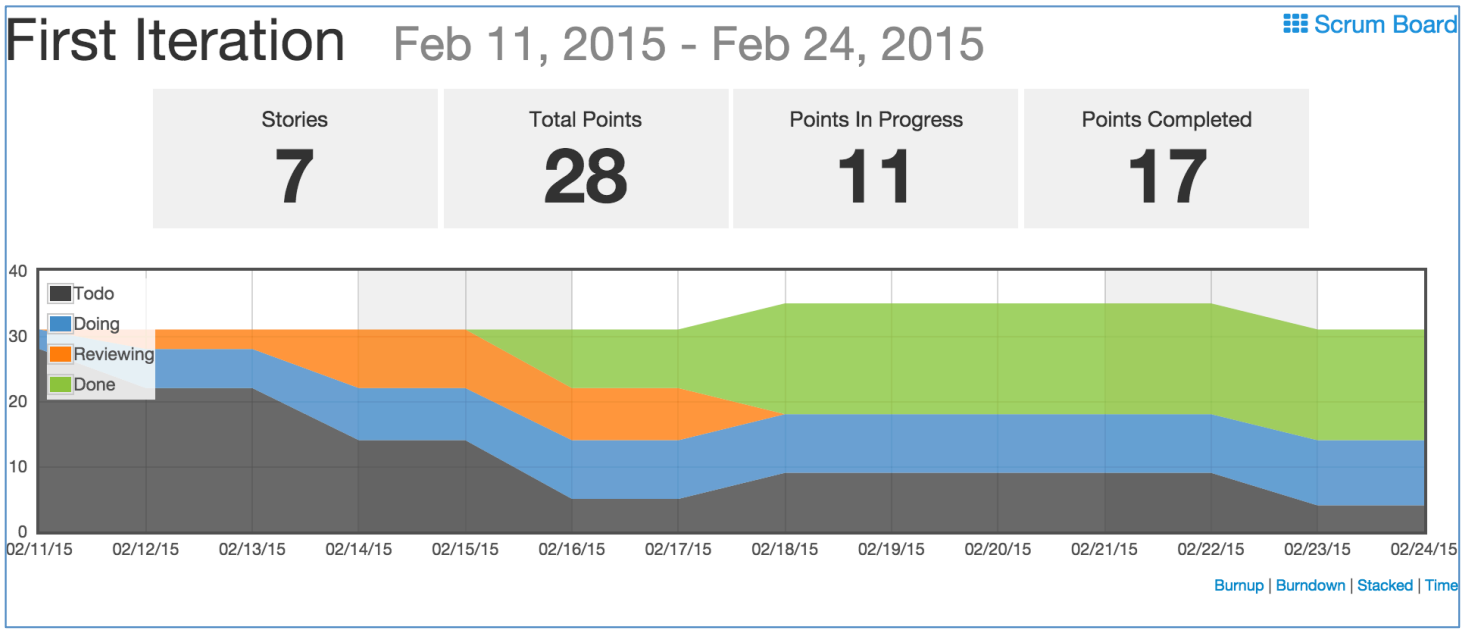
For this project, the following represents the final end-states for each project:

```
ac-games-pojos-0.1.0-SNAPSHOT.jar
ac-games-db-0.1.0-SNAPSHOT.jar
ac-games-db-mongo-0.1.0-SNAPSHOT.jar
ac-games-restservice-spring-0.1.0-SNAPSHOT.jar
ac-games-agent-0.1.0-SNAPSHOT.jar
```

VI. Iteration Summary

Unfortunately, I am no longer able to track hours spent through ScrumDo without paying for an account. But, since I am a team of one, I can honestly report that all effort on this project was my own, and would estimate that I have spent perhaps 40-45 hours of time on this project during this iteration.

Of the ScrumDo views available for tracking progress, this is probably my preferred data view:



This data is not completely reflective of the end state of this iteration, as this view only updates once per day in the free version, but I believe it gives a reasonable accurate picture of story progress over the course of this iteration.

VII. References

[1] – https://github.com/apshaiTerp/cs-5551-documentation/blob/master/CS551-Project-Proposal.pdf
[2] – https://github.com/apshaiTerp/cs-5551-documentation/blob/master/CS%205551%20Project%20Plan.pdf
[3] – http://www.mongodb.org/
[4] – http://docs.mongodb.org/ecosystem/drivers/java/
[5] – http://projects.spring.io/spring-boot/
[6] – http://maven.apache.org/
[7] – https://code.google.com/p/rest-assured/wiki/Usage
[8] – http://maven.apache.org/surefire/maven-surefire-plugin/
[9] – https://www.atlassian.com/software/clover/overview
[10] – https://github.com/apshaiTerp