***Beets***

Post Mortem

Team dynamic went very well. After delegation of roles and assignments we trusted each other to complete their tasks and it worked out. There were little to no conflicts and everybody was understanding. When one person’s load was too great then sharing of load would occur which was very helpful considering time constraints. Unfortunately, with a group of this size time synchronization was nearly impossible since each team member’s class schedule and obligations were different so in-person meetings had to take place during late nights or at very short periods.

**Technologies**

The FreeBugBase quality assurance tool that we decided to use ended up working very well for the team. First of all, the tool was free which was great for an academic environment. But the overall functionality of the software was great for tracking the defects found during our testing execution. The tool didn’t require much overhead when configuring our project account, so there wasn’t much time lost, which was one of the groups initial concerns when deciding on a tool to use. The FreeBugBase user interface was also very intuitive, so there wasn’t much of a learning curve for the group, and team members were able to quickly record defects found. Also the report functionality was nice, as we were able to sort our found bugs by severity and priority.

Our team set up Beets in a variety of different environments and tested it using a variety of software. Since most of our testing was done independently, this didn’t cause many problems. Due to active development, however, it was at times apparent that we were working with different versions of Beets. For example, we installed it on our VM in December and continued testing that version through the beginning of February without updating it. This made it easier to test within that environment, though it might have been nice to get experience testing an evolving product. Some of the other software we used such as nose and coverage worked well and were very useful; as for mutation testing, there were only a handful of tools available and none of them worked properly, forcing us to abandon that strategy.

**Testing**

Compatibility testing was successful in identifying the operating systems that were able to support Beets. A big challenge when performing compatibility testing was making sure python was installed correctly and that any changes made to the operating systems environment were configured correctly. Those changes were made simpler because the developer clearly explained the steps in the installation documentation. Overall compatibility resulted in a beneficial assessment of both the installation process and what operating systems were able to support Beets .

Acceptance was initially difficult because of the lack of requirements documentation and formal specifications. Contacting the developer, Adrian Sampson, was very beneficial as he helped point us in the right direction as well as assisted us with exploring the documentation and architecture of Beets. More time would have granted more Acceptance tests testing the system in different states allowing us to explore more corner cases.

We initially planned on doing both coverage and mutation testing on the existing unit tests and again with new unit tests we wrote included. Mutation testing failed due to insufficient tools - MutPy only worked with Python 3 (this project uses version 2.7), and PyMuTester was very weak and it silently crashed when run. On the other hand, coverage testing was very successful. Coverage.py produced useful HTML output that allowed us to sort files by the number of missing lines and percent coverage, enabling us to easily identify what to target with our unit tests (specifically commands.py).

Despite unfamiliarity with Python, unit testing proceeded very well. Unit testing was very intuitive, and the code we were testing was commented well enough to give us an idea of its purpose and expected output. We would certainly have increased coverage by more than 1% if we had started developing unit tests earlier and devoted more time to it - we may also have discovered more bugs.

Code inspections were the last testing approach we used. Only four of the members actually contributed to the discussion. All of those four members looked at the code that was going to be reviewed and looked for problems before the day of the review. When we got together to do the reviews, we were able to get started right away. We went through each of the classes that were selected, section by section. We had brief discussions every time someone found something that was potentially a bug. The discussion was focused on whether it was a bug and the severity of the bug. The goal of the code inspections was not to find solutions to the bug, so the team members avoided straying away from just finding bugs. Overall, this testing approach was the most effective method, in terms of finding bugs, as 15 bugs were found.

The time allotted to test the Beets project was insufficient to do robust types of tests. One test that was prime for the project is Mock testing and Database testing. Since the application interacts with an external application in order pull metadata for album content, Mock testing would have been optimal on the modular level and application level for acceptance testing. With more time this would have been accomplished. The Beets application comes packed with SQL lite and is used to store the meta-data pulled from the external library. Testing the database would have also been optimal.