1. Data Access Objects Pattern. This pattern is used as a barrier between the actual implementation of that type of persistence and the program. The Data Access Object or DAO is an abstract interface that defines the operations that the program can perform on the model. In our project we used it for persistence but that is not required to use the DAO pattern. An example would be, If you had a car object you would create a carDAO that the rest of the program would interact with but behind the carDAO you could have a sqlCarDAOImpl or mongoCarDAOImpl or any type of implementation of the carDAO that you wanted to use and the rest of the program does not car. In our project we used this pattern in the persistance package. for Example, we havt the IUsersDAO class

```
public interface IUsersDAO {

    /**
     * Retrieve the Users from the Database specified at startup
     * @return a List of IUser
     */
    public List<IUser> getUsers();

    /**
     * Save all the Users to the Database specified at startup
     */
    public void saveUser(IUser user);

}
```

   and then we have the FileUsers DAO and the SQLiteUsersDAO. I am only including the FileUSerDAO as an example. So the program can call the PersistanceProvider.getFileUserDAO and never care which one we decided to use.

```
public class FileUserDAO implements IUsersDAO {
	@Override
	public List<IUser> getUsers() {
		File folder = new File("../data/local/users");
		if (!folder.exists()) {
			folder.mkdirs();
		}
		List<IUser> users = loadFilesForFolder(folder);

		return users;
	}
	@Override
	public void saveUser(IUser user) {
		XStream xStream = new XStream(new DomDriver());
		String xml = xStream.toXML(user);
		try {
			File tmp = new
File("../data/local/users/"+user.getId()+".txt");
			if (!tmp.exists()) {
				tmp.createNewFile();
			}
			FileWriter fw = new FileWriter(tmp);
			BufferedWriter bw = new BufferedWriter(fw);
			bw.write(xml);
			bw.close();
		} catch (FileNotFoundException e) {
			//System.out.println("Could not find file");
			e.printStackTrace();
		} catch (IOException e) {
			//System.out.println("Something went wrong");
			e.printStackTrace();
		}
	}


	private List<IUser> loadFilesForFolder(final File folder) {
		XStream xStream = new XStream(new DomDriver());
		List<IUser> users = new ArrayList<>();
		for (final File fileEntry : folder.listFiles()) {
			if (fileEntry.isDirectory()) {
				loadFilesForFolder(fileEntry);
			} else {
```

```
                                        try {
                                                BufferedReader br = new
BufferedReader(new FileReader(fileEntry.getPath()));
                                                StringBuilder sb = new StringBuilder();
                                                String line = br.readLine();

                                                while (line != null) {
                                                        sb.append(line);
                                                        sb.append(System.lineSeparator());
                                                        line = br.readLine();
                                                }
                                                String everything = sb.toString();
                                                User tmp =
(User)xStream.fromXML(everything);
                                                users.add(tmp);
                                        } catch (FileNotFoundException e) {
                                                e.printStackTrace();
                                        } catch (IOException e) {
                                                e.printStackTrace();
                                        }
                                }
                        }
                        return users;
                }
        }
        }
```

2. Abstract Factory pattern: The purpose of this pattern is to allow you to create related classes from a single factory without having to specify their exact class. Or in other words, you don't have to specify exactly what type of object you want to create, just that you want one from a family of classes. For example, in our project we have a Persistence Provider that contains the UserDAO and the GamesDAO. Depending on what form of persitence you are using, You need to construct the correct type. But the program does not care, It just wants a GamesDAO. So you call the PersistanceProvider.getGamesDAO and it will return the correct one to you.

```
                public static IPersistanceProvider getSingleton() {
                        if (singleton == null) {
                                if(PersistanceProvider.daoType != null)
                                {
                  if(daoType.equals("fileJar.jar")) {
                    singleton = PersistanceProvider.loadJar("fileJar.jar",
"server.persistance.file.FilePersistanceProvider");//new
PersistanceProvider(PersistanceProvider.daoType);
                }
                                        else if(daoType.equals("sqliteJar.jar"))
                                        {
                                                singleton =
PersistanceProvider.loadJar("sqliteJar.jar",
"server.persistance.sql.SQLitePersistanceProvider");
                                        }

                                }
                        }
                        return singleton;
                }
```

3. Plugin pattern: This pattern is used when you want to add external classes without having to recompile. This classes are designed to a predetermined interface. With a plugin you can add additional functionality to your program. For example, in our project we used this to allow us to load different types of persistence methods into our program. This means, that later on if we decided to use a mongoDB we could create a jar that contained the appropriate classes that would interface with our existing program and just load the jar when we started the server. Here is the class where we load the jar into the class path and extract the classes that we need.

```
    private static IPersistanceProvider loadJar(String jarfile, String classname) {
        try{
            Path path = Paths.get(jarfile);
            URL url = path.toUri().toURL();
            URL[] classLoaderUrls = new URL[]{url};
            URLClassLoader urlClassLoader = new
URLClassLoader(classLoaderUrls);
            IPersistanceProvider factory =
(IPersistanceProvider)urlClassLoader.loadClass(classname).newInstance();
            urlClassLoader.close();
            return factory;
        }catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
```

Team Report

Curtis: 4
Mitch: 3
Jacob: 3
James : 1
Kevin: 4

I spent about 25 hours on this phase

Everyone did a pretty good job this time around.
Curtis acted as Project Manager for the most part and he did a really good job
Mitch worked on the SQLite DAO’s
Jacob cleaned up the Commands and added the persistence commands in the server
Kevin: I worked on the File System storage
James: Never heard from him.