Reverse engineering ESE projects

Severin Schoepke, James Matheka, Milan Nikolic

1. System Overview

All the teams strive to conform to the MVC model. We therefore take as a reference template the following Figure 1 from Wikipedia[1]. From this we make a rough sketch of our understanding of each team's model design.
We also note exeptional enities we encountered in the process.

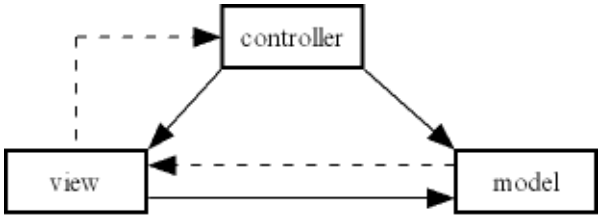
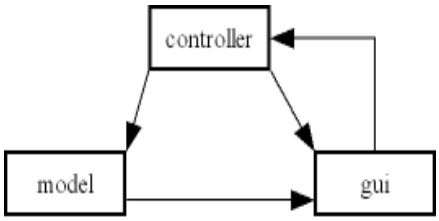


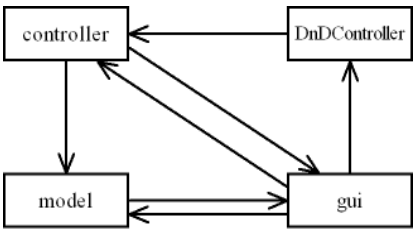
Figure 1

ESE 1



- clear separation of model, view and controller (package-wise)
- 2 God classes: `FrameListener` and `Library`
- 1 Brain Method: `FrameListener::actionPerfromed()`
- one big UI class
- model elements extend Swing classes (e.g. TrackTableModel extends AbstractTableModel)
- FeatureEnvy methods: eg `Library::editTracks()`
- No inheritance

ESE2



- clear separation of model, view and controller (package-wise)
- 1 God Class: `Library`
- 2 Brain methods:
  `ListPaneListener::actionPerfromed()`
  `FrameContent::getTablePane()`
- ShotgunSurgery Methods: `Library::getAlbum(index)`
- FeatureEnvy methods:
  `Library::addTrackToAlbum(Track)`
- Not well designed (e.g. `addTrackToPlaylist()` in `Library` and not in `PlayList` - no RDD)
- Use of abstract classes to group similar behaviors

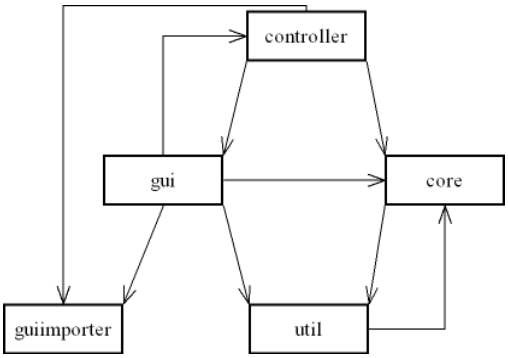
ESE3

- Single classes assigned multiple functions (no separation of tasks). For example, single classes act as controller, view and model (`jTunes`)
- The `jTunes` class constructor acts like a singleton pattern.
- (not very strict) separation of model, view and controller - track playing functionality in gui package (`BasicPlayer`)
- GUI classes change the state of the model directly, e.g. . `AbtstractImporter::importFiles()` calls `Library::add(Library)`
- Obsolete classes e.g. `FileImporter`
- Type of inheritance suggests a lot of code reuse
- 3 God Classes: `MainMenuBar`, `MainFrameController`, and `TrackDisplayPanel`
- 3 Data classes: `SimpleProgressBar`, `Track` and `AbstractImporter`

ESE4

- another case of classes behaving as controllers, views and models at the same time.
- messy structure: no real separation of model, view and controller
- 2 God Classes: `PlayList` and `PlayListItem`
- 5 Data Classes: `EditPlayListDialog`, `EditTrackDialog`, `Loader`, `Saver` and `Track`.
- No inheritance

ESE5



- 1 Data Class: `newTableModel`
- Improper use of hierarchy: `AbstractPlayList` should not use its extending class `UserPlayList`
- Hierarchies also not well defined e.g. implementation of `addObserver()` should be done on `AbstractPlayList`
- Uses the Observer pattern
- GUI module updates the model directly e.g. `addPlayListGUI::actionPerfromed()` calls `Library::add(playlist)`
- Many unused imports

2. Extensibility

Most of the projects are not very extensible: No generic interfaces for different functionalities are provided so that adding a new implementation would be easy.
In the following subsections the five projects are compared regarding the addition of a new view and a new persistence strategy.

2.1 Adding a new view:

All of the teams tried to apply the MVC pattern (model, view and controller separation), so it should be easy to add a new view since the model and the view should not be coupled.

ESE 1:
The UI is implemented in a single, big class which is the whole 'application' in the same time (includes the main() method). This design makes it difficult to add a new view: Adding a new view would be the same as wirting a new application which uses some model classes.

ESE 2:
The UI is implemented in different classes. The most important class seems to be the ContentManager: It contains a FrameContent field that holds all UI elements. To implement another view one could swap this FrameContent, but it would not be a really nice solution, since no interfaces are defined that different FrameContents must implement. So a reengineering effort would be required.
Another problem is that the view and the model are not clearly separated: The LibraryPlayer class (model) for example has a field of the type PlayerPanel (view).

ESE 3:
The UI is separated into different classes and there's an 'application' class (jTunes) that holds the view as well as the model. Since there's no interface defined for the view class, swapping the view would require changes in the jTunes class as well...

ESE 4:
The design is similar to the one of team 1: There's one UI class that is the main application in the same time. So adding another view would require a reimplementation of the whole application.
ESE 5:
The UI is implemented in a couple of classes, the biggest part is in the class NewGui. There seems to be missing encapsulation, since all parts of the GUI (buttons etc.) are created in the main() method and passed to the constructor of the NewGui class. As there is no interface for a view defined, one would have to refactor the code before one could add a new user interface.

In conclusion it can be said thatr none of the programs was developed with the addition of new user interfaces in mind. All of them would require a reengineering effort before a new UI could be nicely integrated into the existing applications.

2.2 Adding a new persistence strategy:

All of the teams implemented some sort of Loader class. but none of them bothered to define a generic Loader interface, so implementing a new persistenc strategy and using them in the projects is not very straightforward.

ESE 1:
The LibraryFileHandler class is responsible for loading and saving the library. It is called in the methods load/saveData() in the Library class. Adding another persistence strategy would require some effort since no interface for the Loader used by the Library is defined...

ESE 2:
The ContentManager holds a LibrarySaver instance which handles all loading and storing operations. Again, there's no generic interface for a Loader defined, so adding a new persistence strategy would require some reengineering.

ESE 3:
The Initializer class holds instances of the DataLoader and -Saver classes. As with the projects above, no generic Loader/Saver interface is defined so adding a new storage mechanism would require a reengineering effort.

ESE 4:
The Library class holds instances of the Loader and Saver classes. As with the projects above, no generic Loader/Saver interface is defined so adding a new storage mechanism would require a reengineering effort.

ESE 5:
The XMLUtils class holds methods for loading and saving a Library instance to a XML file. So the basic design is not like that of the other teams: They all have some sort of loader and saver classes. Nonetheless, implementing a new persistence strategy is not straightforward: One would need to integrate the new persistence strategy into the application after having implemented it which would require some additional effort.

3. Unit Tests

Coverage: The unit tests should at least test all the user stories requirements and at optimum cover all functionalities introduced into the respective system
Criteria: White Box Testing: should tests all the possible pathways in a functionality. This includes not only correct inputs, but incorrect inputs, so that error handlers can be verified as well.

ESE1: 52 runs, 0 errors, 0 failures
- covers about 50% of the model part of the system
- not all user requirements are covered
- no verification of incorrect input
- no verification of error handling

ESE2: 17 runs, 0 errors, 0 failures
- very little tests covering the model part of the system
- requirements are not tested at all
- empty tests (LibrarySaverTest)
- no verification of incorrect input.
- no verification of error handling

ESE3: 80 runs, 0 errors, 0 failures
- covers the model part of the system
- covers some verification of incorrect input
- handles some verification of error handling, but should not print the exception stack e.g. `AlbumTest::testAddTrackAppend()`.

ESE4: 26 runs, 0 erros, 1 failure
- very little testing covering the model part of the system
- not very useful testing since user requirements not fully tested
- failures not handled, system remains buggy
- no verification of incorrect input.
- no verification of error handling

ESE5: 17 runs, 4 errors, 0 failures
- very little testing covering the model part of the system
- tests do not cover all that they supposedly test
- tries to test verification of incorrect input. Assertion and exception handling not done well and therefore doesn't make sense e.g. use of *assertTrue*(**false**) in `PlayListTest::testGetTrack()`
- exceptions are caught and the stack trace just printed; the fact that the code under test failed to execute is reported by JUnit as a success, hence code remains buggy.
- errors occur since exceptions are not dealt with correctly

Conclusion:
- no team does comprehensive testing. Not all of the user requirements are even tested.
- all teams except ESE4 use (or try to use) the design by contract pattern to handle errors. Errors generated are not handled well or the pattern is used without making any sense. Examples are:

```
public void add(Track track) {
  assert (track != null);
  assert (this.invariant());
...
}
public void importFinished(File[] files) {
   ...
   if (files.length > 0) {
      ...
   } else {
      // TODO: Implement propper error handling
      assert (false) : "No playable files in selected folder.";
               }
   ...
}
```

No handling of errors if they occur

Does not make sense

- all teams do not override default exception handling to aid in pin-pointing where errors occur or how they deal with them.
- ESE5 could test verification of invalid inputs in a more sensible way. A suggestion:

Original

```
/**
 * The playlist should throw an exeption (IndexOutOfBounds)
 * if one tries to get a track that doesn't exist.
 */
public void testGetTrack() {
    try {
        this.aPlayList.getTrack(0);
        assertTrue(false);
    }
    catch (IndexOutOfBoundsException e) {
        assertTrue(true);
    }
    catch (Exception e) {
        assertTrue(false);
    }
}
```

Better

```
/**
 * The playlist should throw an exeption (IndexOutOfBounds)
 * if one tries to get a track that doesn't exist.
 */
public void testGetTrack() {
    try {
        this.aPlayList.getTrack(0);
        fail("Should never get here");
    }
    catch (IndexOutOfBoundsException e) {
    }
    catch (Exception e) {
    }
}
```

4. Design Pattern

In this section the use of design patterns in the implementations of the ESE projects as well as their documentation are analyzed.
All of the teams try to use the Model View Controller pattern as the general design for their implementations. However, most of them fail to really separate the model, the view and the controller subsystems. Furthermore, none of the teams documented their design and the use and implementation of the MVC pattern.
In addition to the MVC pattern, team 3 and team 5 use the Observer pattern to handle notification of objects about changes in entities that they are interested in. But no team documented (or explained) their use and implementation of this pattern.
Patterns other than the ones mentioned above were not used. We looked for common patterns like Singleton and Composite but nothing was found. We strongly suggest that the teams consider using the Singleton pattern for manager-like classes that have only one instance in the whole system. In addition, the use of the Composite pattern may be considered for the composition of tracks into albums and playlists.

5. Documentation

In the following section the documentation is analyzed. We looked at their usefulness, whether it fits the code or not and at the correctness of the documentation (javadoc).
In general it is to be said that documentation is a weak point of all the projects. None of the teams documented the whole code (every class and every method should have javadoc comment), some of them did almost comment nothing (team 2 and 3).
When there is documentation, it often is not very useful or it is wrong:

ESE 1:
The documentation is incomplete, but good in some parts (e.g. Track or classes). There are some wrong comments (e.g. in the ... class). //TODO: find the wrong comment again...

ESE 2:
There is almost no documentation and, in addition, there are unuseful comments like "// Do something with the file" (in the ContentManager class).

ESE 3:
There is almost no documentation or comments in the code.

ESE 4:
The documentation is incomplete, but good in some areas. In addition, there are errors in the javadoc

code (a lot of methods contain e.g. @author tags in their comments which are just ignored by the javadoc parser).

ESE 5:
The documentation is rather complete, only some parameters and return values are not documented. In addition, the team makes good use of eclipse's support for // TODO: comments.

6. Code Duplication

In this section we tested for any code duplication in the projects. For the duplication to be of any value for our analysis, we limited ourselves to look for duplications that make the system hard to understand and expensive to extend.

Criteria: We identify duplicated clones as having the following properties[2]:
- Minimum length of duplicated code set to 7. This makes our duplicated clones large enough to be significant.
- Minimum exact chunks of code is set to 2. Avoid unnecessary duplication crumbs.
- Minimum line bias in duplicated code is set to 2.

The results are depicted in the table below:

Team	NOF	LOC	NOD (NOL)	Exact	Mod.	Comp.	Ins.	Del.	NODL (%)
Team EINS	26	2,252	25(246)	5	18	1	1	0	11
Pinky and Brains	36	3,148	25(507)	14	6	1	3	1	16
Dini Mer Corp.	91	9,902	259(1,693)	88	55	29	54	33	17
Backstreet Boys	25	3185	58(640)	13	33	4	5	3	20
BackBone Systems	41	2413	32(412)	6	20	1	0	5	17

```
Key:    NOF: Number of Files tested
        LOC: Lines of Code tested (without comments)
        NOD (NOL): Number of Duplicate Chains (Number of duplicated Lines)
        Exact: Exact duplicated clones (copy-paste)
        Modified: copy-paste duplication with a maximum of 2 lines of modifications
        Ins.: Duplications with insertions (minimum of two lines of insertions)
        Del.: duplications with deleted sections across them
        Comp.: duplications with combination of deletion, insertions and modifications
        NODL(%): Percentage of duplicated code in reference to the total code
```

- Most of the code duplication happened in GUI and controller classes, or across them.
- The duplication also shows the absence of clear distinction between the components of applications.
- Exact duplicates were found in the same class and also across different classes. e.g. in ESE5 SpectrumAnalyzerDisplayPanel, start line: 727, end line: 744 and VisualManager, start line: 230, end line: 249, have exactly the same 15 lines of code.

7. Code Smells

In this section we present various code smells that we found in the different projects. The list is by no means complete and we didn't analyze whether one team does exceptionally good or bad.

Bad comments:
"//Do something with the file" (Doesn't contain any information)

Bad naming:
LibrarySaver (it also contains code for loading)
MyBasicPlayer
C++ attribute naming style (m_ prefix, in ESE3's BasicPlayer class)

Playlist LibraryPlayList; (attribute names should begin with no capital letter)
Powermanja and turboProp() ??
private JSplitPane jSplitPane = null;

Not implemented catch blocks:
// TODO Auto-generated catch block

Bad 'packaging':
view and model classes in the 'root' package (ESE4)

Non- portable code:
new File(homePath + "\musicLibrary.xml"); (Does not work on UNIX)

Obfuscated code:

```
Player player = new Player();
CorePlayList corePlayList = new CorePlayList();
Library library = new Library(corePlayList);
// TODO why create this list?
UserPlayList userPlayList = new UserPlayList(corePlayList);
PlayButtonController playBtnController = new PlayButtonController(library, player);
StopButtonController stopButtonController = new StopButtonController(player);
ImportController importController = new ImportController(library);
//ImportButtonController importBtnController = new      ImportButtonController(importController);
ForwardButtonController forwardButtonController = new ForwardButtonController(player);
PreviousButtonController previousButtonController = new PreviousButtonController(player);
ShuffleButtonController shuffleButtonController = new  ShuffleButtonController(userPlayList,
library);
PlayerStatusPanel playerStatusPanel = new PlayerStatusPanel(player);
NumberOfTracksPanel numberOfTracksPanel = new NumberOfTracksPanel(corePlayList);
NewGui gui = new NewGui(playBtnController,
                        stopButtonController,
                        importController,
                        forwardButtonController,
                        previousButtonController,
                        shuffleButtonController,
                        library,
                        playerStatusPanel,
                        numberOfTracksPanel);
```

Bad use of language features:
class Initializer implements GUIPersistenceKeys (GUIPersistenceKeys contains only public static constants)
setActionCommand(this.SHOW_ABOUT); (SHOW_ABOUT is static)$

Shotgun surgery methods, Feature envy methods and God classes mentioned earlier in section 1.

8. Estimating a User Story

In this section, we are going to estimate the following user story:

"The user wants to share his favorite playlist with all his friends that have access to his shared hard-drive. He exports the playlist in the M3U playlist format so that others can import the playlist in their favorite music player. Some of his friends are using 3rd party software, others are using an ESE MusicPlayer."

The following must be done to implement this story:

Implement a parser and an exporter that converts ESE playlists into M3U playlists and vice versa
Integrate this parser/exporter into the existing apps

We split our estimation into these two parts.

M3U Importer/Exporter:
All ESE projects have a similar implementation of playlists: All of them provide access to all Track objects in a playlist and all Track objects somehow contain the filename of the corresponding music file (e.g. as a String or encapsulated in a File object).
All projects also provide constructors or setter to create new Track objects and adder to add different tracks to playlists.
This way, a generic Importer/Exporter could be implemented that looks about the same for each project. It would just provide 2 static methods to either import a M3U file into an ESE playlist or to export an ESE playlist to M3U.
We estimate the time needed to implement this Importer/Exporter to be 2 man-days.

Integration into existing projects:
The generic Importer/Exporter described above must be integrated into the existing applications. This Integration consists of adding new control elements to the user interfaces and calling the corresponding Importer/Exporter methods. When importing a playlist, the newly created playlist must be stored/added to the library and when exporting we need to access the currently selected playlist.
We looked through the different projects and came up with the following time estimation:

Team:	Generic stuff:	Integration:	Total:
ESE1	2	1	3
ESE2	2	2	4
ESE3	2	3	5
ESE4	2	3	5
ESE5	2	5	7

(all numbers are man-days)

References:

[1]. http://en.wikipedia.org/wiki/Model-view-controller
[2]. Mihai Balint, Tudor Gîrba and Radu Marinescu, **"How Developers Copy,"** *Proceedings of International Conference on Program Comprehension (ICPC 2006)*, 2006, pp. 56– 65.