Reverse Engineering ESE Project

ESE1

Overall design, extensibility and fitting to user story/requirements

User story / Requirements

Importing files
Works fine.

Creating empty playlists
Works fine.

Managing playlists
No separation of playlists and the library.

Playing songs
Playing songs works generally fine (with the play and stop buttons), but some files can't be played and throw exceptions in the console.

Persistence
Works fine.

Playing playlists
Works fine.

Shuffle tracks in playlists
The tracks are well shuffled and can be listened to, but the tracks can't be unshuffled anymore.

Rearranging tracks in playlists
Rearranging tracks one by one works fine, but rarranging several tracks at a time behaves strangely if one wants to put them at the bottom (or the top) of the list.

Pause
Works, but there is some annoying delay between the click on the pause button an the actual silence of the music player.

What is the overview of the system?

The main class which contains all the GUI and model classes is the MusicPlayer (inheriting from JFrame). Among all the GUI attributes as Buttons, Labels etc. it contains a Library and a TrackPlayer. The Library is a large class which encapsulates a multitude of functions as managing (add, remove, edit and so on) albums and tracks, navigating through a playlist and many more (more than 40 methods and 600 lines of code).
The TrackPlayer on the other hand is the device for playing tracks from a TrackTableModel (like Library or Playlist). It can be paused or be told to switch to the next track.

OO design?
There is a high coupling in the Library class, since it manages persistence and track handling all in one (some delegated functionality to Playlist though). Playlist and Album do share much functionality, since they are tracks in principle, but they don't inherit from a common superclass or use common classes.
TrackTableModel inherits from AbstractTableModel which is a GUI-Component. This means that Playlist is in fact a GUI component too and is not present in the model (MVC).

Designed understandable, pragmatic?
Relatively understandable, but messy with the handling of the tracks and their belonging to playlists and albums.

How extensible is the design?
Libraries are saved with the Xstream tool and the methods are situated in the Library class itself. So export and import have to be done in the same way, which would be of some effort. Other extensions however

would be nastier to do, because of the tightness of model and user interface.

How easy is it to plug in a different view (for example a Web Interface) into the application?
As the TrackTableModel is inherited from a swing-component, a whole new class has to be written and the functionality of the subclasses of TrackTableModel have to be copy-pasted. The TrackPlayer can be reused quite as it is.

How easy is it to plug in a different persistency strategy (for example a Web services) into the application?
The Persistence is handled with XStream's .xml-files, so there would be two method adaptions in the Library-class (saveData and loadData), but in a more object oriented way, these methods should invocate a persistence class (in which the webservice/database changes could be easily modified locally)

BuildProcess
Does the project build?
yes.

Tests and error handling

Tests
52 Tests, all green

They do not test with real mp3-files, but only with a wrapper. So they can't be sure that it works with real mp3-files too. Especially the load-funciton isn't tested. We found two commented tests in LibraryTest. They are both about removing something from the library. This case isn't covered with other tests.
The tests cover almost all model-classes, except the Seriazable classes of Library, Selection, Playlist and Track and a LibraryFileHandler and a TrackTabelModel. As the TrackTabelModel is an abstract class, it is ok to not test it. The LibraryFileHandler instead should be tested as he has public methods. Not all public methods of the tested model classes are tested.

Error Handling
There are only auto-generated catch-blocks, which do nothing or print out only the stracktrace.
The Library, the Album and the Track Classes check a pre- and postcondition (invariant). If the invariant-check fails, the whole program fails???
We tested to add a non-existing file: the programm adds it without complaining. But if you like to hear what this nothing.mp3 consists, nothing happens, not even an info is written in the console, that this file doesn't exists.

CVSStats

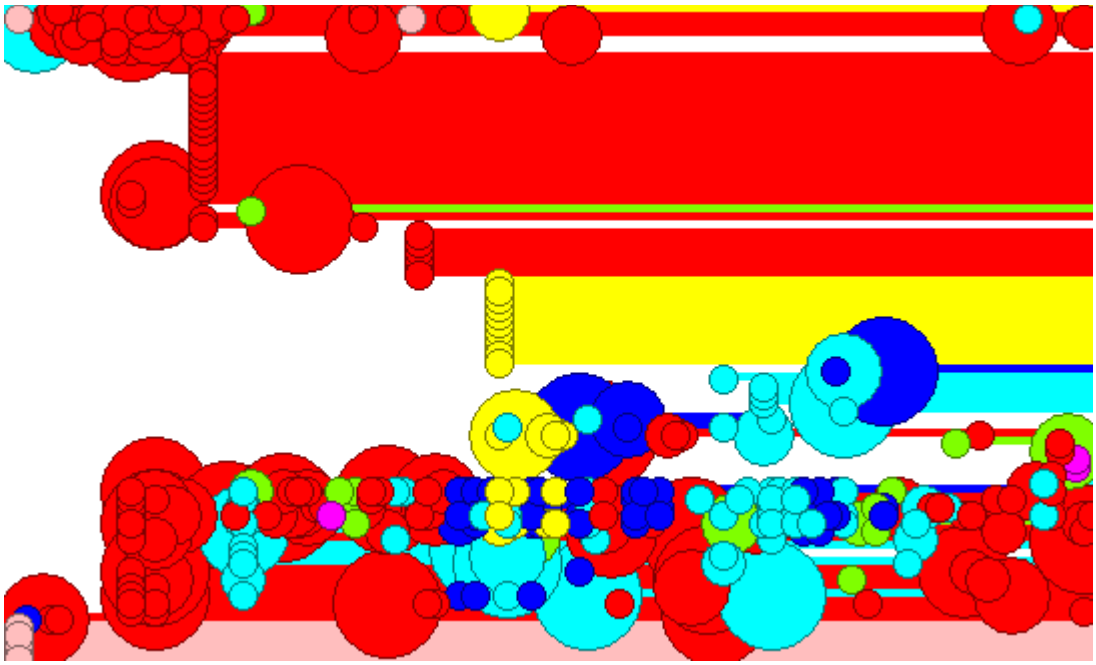

Abbildung 1: simon (red), schuler (cyan), oesch (blue), john (chatrtreuse aka green), michaelstoeckli (mangenta), mbinz (yellow), ese-staff (pink)

We can see immediately that the mangenta author (Michael Stöckli) doesn't participate. He has only three commits, two of them affect the UML-Diagramm.
On the other hand, the red author (Simon Schwab) is really active. He added a lot of stable data (png) and implemented/changed a lot of things. The yellow author (Manuel Binz) added a lot of stable data too (in his case, the stable data are jar-files). Besides this, Manuel Binz wasn't really active. Perhaps we can identify a bug-fixing cvs metric.
But Simon Schwab is not the only active author in this team: the cyan author (Simon Schuler) has worked closely with red on the Java-files.
The green (Sundar Klingenberg) and the blue (Jan Oesch) author had more or less the same role: they both gave some small inputs (green even fewer than blue), but almost don't own any file.
Further, we see that this team worked over the whole period, not only a sprint in the last two days.

Documentation and coding style

Documentation

Besides the doxygen, there is a small UML-Diagram in the cvs repository which correctly reflects a part of the project.

Javadoc: Maybe a third of the classes are documented. The comments are short; they often do not tell the whole story:

```
/**
 * Removes a Track from the library
 *
 * @param track
 *          The Track to be removed
 */
public void removeTrack(Track track) {
      int index = this.trackIndex.indexOf(track);
      this.trackIndex.remove(index);
      track.getAlbum().remove(track);
      //this.createTrackIndex();
      for (Playlist playlist : this.playlists) {
            playlist.removeTrack(track);
      }
      this.fireTableRowsDeleted(index, index);
}
```

This method does not just remove a track from the library, it removes it from two lists, from the album and notifies the listeners of the change.

Patterns: There are no patterns mentioned in the documentation at all.
And it seems that there are no design patterns used in the project, which could have been documented.

Code details

Lint4j: there are 8 problems listed.

Besides lint4j:

```
public void setEnabledShuffleButton(boolean b) {
      if (b) {
            this.shuffleButton.setEnabled(true);
      } else {
            this.shuffleButton.setEnabled(false);
      }
}
```

Write `this.shuffleButton.setEnabled(b)` instead.

```
private LinkedList<Album> albums;

private ArrayList<Playlist> playlists;

private LinkedList<ListDataListener> listListeners;

private ArrayList<Track> trackIndex;
```

Use higher Interface List (or Collection) instead.

```
public void addAlbum (Album album) {
	assert (album != null);
	assert (this.invariant());
	int oldAlbumCount = this.countAlbums();

	this.albums.add(album);

	assert (this.invariant());
	assert (this.countAlbums() - oldAlbumCount == 1); // post-conditions
}
```

Very nice design by contract.

```
public boolean isDataFlavorSupported(DataFlavor flavor) {
	for (DataFlavor f : this.trackFlavors)
		if (f.equals(flavor))
			return true;
	return false;
}
```

Recommended use of braces after if's and for's.

```
public Object getInfo(int index) {
	assert (index >= 0 || index <= 2);

	if (index == 0) {
		return (this.getTrackNumber());
	} else if (index == 1) {
		return (this.getTitle());
	} else if (index == 2) {
		return (this.getArtist());
	} else {
		return null;
	}
}
```

Use switch-case-statement. Add constants holding those dangerous - as they might change in this method and not in the clients - integer values.

ESE2

Overall design, extensibility and fitting to user story/requirements

User story/requirements

Importing files
The import dialog is very slow and the tracks are put into their albums, if there is one specified in the mp3 or in an album named "Kein Album" (which is fancy but the requirements beg to differ).

Creating empty playlists
Works fine

Managing playlists
Works fine except for deleting tracks out of playlist which some times does not work.

Playing songs
Works generally fine (with the play and stop buttons) but some files can't be played and throw exceptions in the console (same as ESE1).

Persistence
Works fine.

Playing playlists
Works fine.

Shuffle tracks in playlists
The tracks are well shuffled and can be listened to.

Rearranging tracks in playlists
Rearranging tracks one by one works fine but rearranging several tracks at a time does not really work.

Pause
Does stop playing but does not resume once paused. One has to press the stopp button to be able to play some other song again.

What is the overview of the system?

The ContentManager is responsible for all sort of functionality of the program, but it does not do all the work. All the functionality is delegated to other classes except for the storing of the selection of Albums and Tracks. For instance there is the Library class which contains a list of Albums and a list of PlayLists. The class manages the adding and removing of tracks, albums and playlists. Furthermore the ContentManager contains a LibrarySaver which saves and loads Librarys.
The ContentManger is also responsible for the update of the GUI. It contains the MainFrame and the FrameContent, which display the content. The ContentManager is in some sort the C of MVC (and not missplaced in the model package). The FrameContent itself inherites from JPanel and holds all GUI-Components.

Design OO-conform?
MVC is more or less well implemented (M = Library; V=MainFrame/FrameContent; C=ContentManger). However classes know each other too much. The Observer Pattern should be used and for instance Library implements Observable, but it is never used.
LibraryElement is superclass of Playlist and Album which share functionality and attributes, i.e. a list of tracks and some accessors (nice).

Designed understandable, pragmatic?
Understandable but messy with the message sending (Observer Pattern once again)

How extensible is the design?
For the import and export only the LibrarySaver has to be enhanced and a few GUI components have to be added. In general the Architecture is quite structured and probably extensible.

How easy is it to plug in a different view (for example a Web Interface) into the application?
The classes know each other too much, but a slight reengineering of the message handling would solve the problem. A direct access to the Library class would work too (since it is encapsulated).

How easy is it to plug in a different persistency strategy (for example a Web services) into the application?
The Library is saved in the LibrarySaver as an xml file, so an easy replacement of the LibrarySaver methods would do it (but an interface for all saver methods would be nice).

BuildProcess
Does the project build?
yes.

Tests and error handling

Tests
17 Tests, all green

They do not test with real mp3-files, not even with a wrapper. Only void Tracks are added, edited and removed from the albums / libraries. This does not correspond to the work the player has to fulfill later.
They test almost all Model Classes except the LibraryElement, the LibraryPlayer and the Player. The LibraryElement is an abstract class, so it is ok to not test it. The methods from the abstract class should be tested within the subclasses, which is not the case.

We found in the testcases assertions like that:

```
assertTrue(track.getNumber() == 2);
assertTrue(a.getRepeat(c).toString().equals(c.toString()));
```

They should be replaced by the method `assertEquals(arg1, arg2)` which is faster.

Error Handling
We found some active error handling: catch blocks which are not only printing the stacktrace, but do really something. On the other hand, we ask ourself, if this is really the right usage of the catch-block. It just seems to be some kind of an if-else statement.

```
private void readFile() {
      DocumentBuilderFactory factory =DocumentBuilderFactory.newInstance();
     try {
            DocumentBuilder builder = factory.newDocumentBuilder();
            this.doc = builder.parse(new File("library.xml"));
     } catch (SAXException sxe) {
            // Error generated during parsing)
            Exception x = sxe;
            if (sxe.getException() != null)
                  x = sxe.getException();
            x.printStackTrace();
     } catch (ParserConfigurationException pce) {
            pce.printStackTrace();
     } catch (IOException ioe) {
            // No file found? make a new one
            this.makeDefaultFile();
            try {
                  DocumentBuilder builder = factory.newDocumentBuilder();
                  this.doc = builder.parse(new File("library.xml"));
            } catch (Exception e) {
            }
     }
}
```

Tested to add a non-existing mp3-file. Nothing happens: the file doesn't appears in the playlist and no error

is thrown. The System has a windows sickness: radical changes can only be applied after you restarted the system. Particularly the removing of a track throws an ArrayIndexOutOfBoundsException and the track is still playable in the playlist. The next time you start the Musicplayer, the track will be removed.

CVSStats

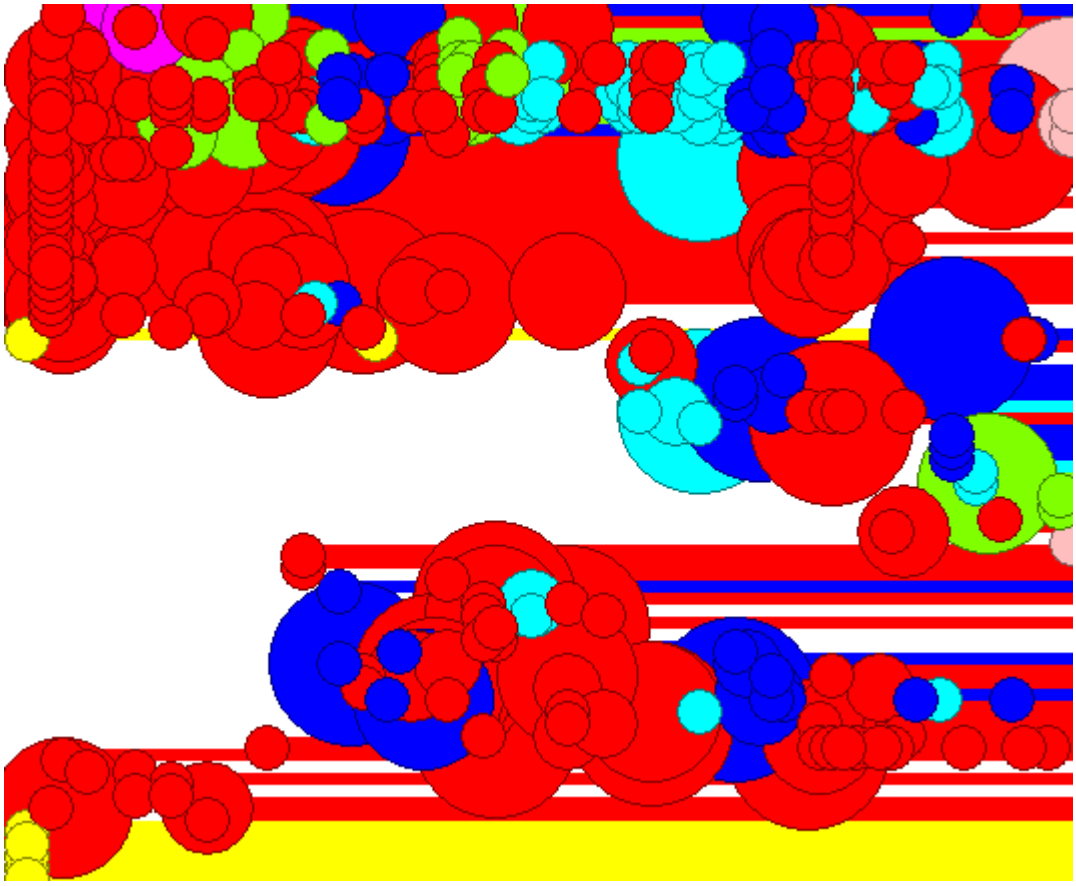

Abbildung 2: zeradun (red), mariusmoser (cyan), mbaumg (blue), silvan (chartreuse aka green), mike (magenta), simon (pink), ese-staff (yellow)

In this cvs-view it's obvious, that the red author (zeradun) is the real owner of this project, not team 3. He initialized the project and added almost all data. The second thing we can see, is that there is only a few stable data: the jars added by ese-staff.
The upper part is about the model and the gui, in the uppermiddle part are necessary files such as the readme and the build.xml. The lowermiddle part attends the drag and drop package.
We have two authors in this team (mangenta (Michael Ineichen) and pink (Simon)) that do not really participate. Michael Ineichen has only one commit at the beginning of the project, Simon only one short before the deadline.
There's not much to say about the other three authors: they were active during the whole time on the three upper parts.

Documentation and coding style

Documentation

Besides the doxygen, there is a small readme file addressed to users; a UML-Diagram is missing.

Javadoc: No link to javadocs on the build server, all the other teams do have such a link.
The reason is obvious: We generated the javadoc by ourselves and noticed that about 3 classes out of 37 contain javadoc, and even these comments are rather poor.
Example:

```
/**
 * Set the Mainframe. For Access
 *
 * @param frameArg0
 */
public void setMainFrame(MainFrame frameArg0) {
	this.frame = frameArg0;
	this.innerFrame = this.frame.getInnerContent();
}
```

We would have figured out also without the comment that the method setMainFrame() sets the mainframe. But what the fragment 'For Access' means is not clear at all.

Patterns: As there is absolutely no documentation (neither online nor as javadoc), we were not able to find documented patterns. But while reading through the code, we found a class extending the Observable class. But there are no methods which override methods of the super class, and also no methods of the super class are called from anywhere of the project. In addition, there is no class implementing the Observer Interface, so the Observer Pattern isn't actually used.

Code details

Lint4j: Lint4j lists zero problems, this is the only group which managed to get this value.

Besides lint4j:

```
	private ArrayList <Album> albumList;
	private ArrayList <PlayList> playLists;
```

ArrayList is bad, the higher interface List (or even Collection) should be used.

```
public void appendToConsole(String s) {
	if (frame == null) {
		// Frame is null
	} else {
		frame.appendOutput(s);
	}
}
```

The first case can be left out and the condition only checked on a frame of being not equal to null.

```
	public final String STANDARD_ALBUM_NAME = "Kein Album";
	public final String DEFAULT_PLAYLIST_NAME = "All Tracks";
```

Why these constants are not declared 'static'?

```
	public Track isTrackLoaded(File file){
		return this.table.get(this.getHash(file.toURI().toString()));
	}
```

An undocumented method should not be named isTrackLoaded() and then not return a Boolean but an Object of some other type.

```
public ArrayList<Album> getSelectedAlbums() {

            return null;
}
```

This method is absolutely useless.

ESE3

Overall design, extensibility and fitting to user story/requirements

User story / Requirements

Importing files
The tracks are sorted out by artist and put int their albums, if they are specified in the file (requirements...). Tracks with no artist/album are not imported.

Creating empty playlists
Works fine

Managing playlists
Works fine

Playing songs
Works fine but all tracks of an artist can't be played.

Persistence
Does not work (database errors).

Playing playlists
Works fine

Shuffle tracks in playlists
Works fine, but one can't begin with the last track in the list.

Rearranging tracks in playlists
One at a time works fine.

Pause
Works fine, but there are many exceptions thrown in the console

What is the overview of the system?

The jTunes class contains all important classes like the GUI class MainFrame, the ImportManager, persistence classes and the the sound facilities. As in ESE2 there are Albums and Playlists which inherit from a common superclass: TrackCollection, which is mainly a list of tracks.
The persistence is very scattered and relies upon a database (which is a little odd, since every user has its own playlists); there is only one central class for persistence: the DatabaseAccessHandler, but it provides only wrappers for the queries and initialisation of database connections. DataSaver, DataLoader and MyProperties use the DatabaseAccessHandler for SQL statements.
A Library is a collection of Albums, as PlaylistLibrary is a collection of Playlists. Both are used in the MainFrame as model.

OO-Design?
The Observer Pattern is well implemented for the Library class and the PlayListLibrary. But as already told, the persistence is very scattered around. Queries should be hidden behind methods of the DatabaseAccessHandler and not be generated in the calling itselves
The model (Libraries, PlayLists, Albums) are seperated from the GUI and are used by a counterpart in the GUI (for instance PlayListEditor for PlayList)
PlayList and Album inherit from the same super class (TrackCollection), but Library does not. There might be a need for a Composite Pattern.

Designed understandable, pragmatic?
Confusing persistance, but the model is very straight forward.

How extensible is the design?
The import and export might be problematic because of the sql-queries which are thrown in the wind. The other components are quite extensible.

How easy is it to plug in a different view (for example a Web Interface) into the application?
Quite straightforward thanks to the Observer Pattern.

How easy is it to plug in a different persistency strategy (for example a Web services) into the application?
Not that easy, since all classes which use persistence do it on their own with the help of the DatabaseAccessManager (for the connection). The changes would therefore not be local.

BuildProcess
Does the project build?
Yes.

Tests and error handling

Tests
80 Tests: all green

This group has a TestSuite that runs all tests.
The tests are done with a track wrapper, but sometimes with real mp3-files too. Why are not all tests done with these files? The player will have to handle later with real mp3-files.
What we don't like are the void testcases. There are a few of them: the DataLoaderTest seems nice from the outside, but inside is nothing: the setup is forwarded to the superclass and the only testcase is void. Then the FullscreenWindowGUI seems to be tested. Actually, it does nothin too. As the tests should work on all systems, it's not necessary to test the gui, otherwise this test would fail on a commandline only system.
Even if there are 80 testcases, not everything is tested (every public method of a model-class should be tested). For example, the TrackTest has 8 cases, the Track class 15 methods plus 11 constructors.

Error handling
There are a lot of auto-generated catch blocks which only print out the stacktrace or are „silently ignored".
But some of the catch blocks work with the caught exception: print out some additional information or throw a new exception. The additional information avoids debugging: how are the variables set at the time an exception is thrown? We think, to thow a new exception if one occurs is not the best solution. This exception has to be catched in the next method that accesses this method. Like that, the exceptions are in the whole program.

Here we tested too, how the MusicPlayer handles non-existing files. He `finds` one music file and asks me, wheter we want to add it or not. If we agree, nothing happens (neither an exception nor the file appears in the playlist). Removing single tracks functions in this player without problems, but a playlist throws an ArrayIndexOutOfBoundException while removing.

CVSStats

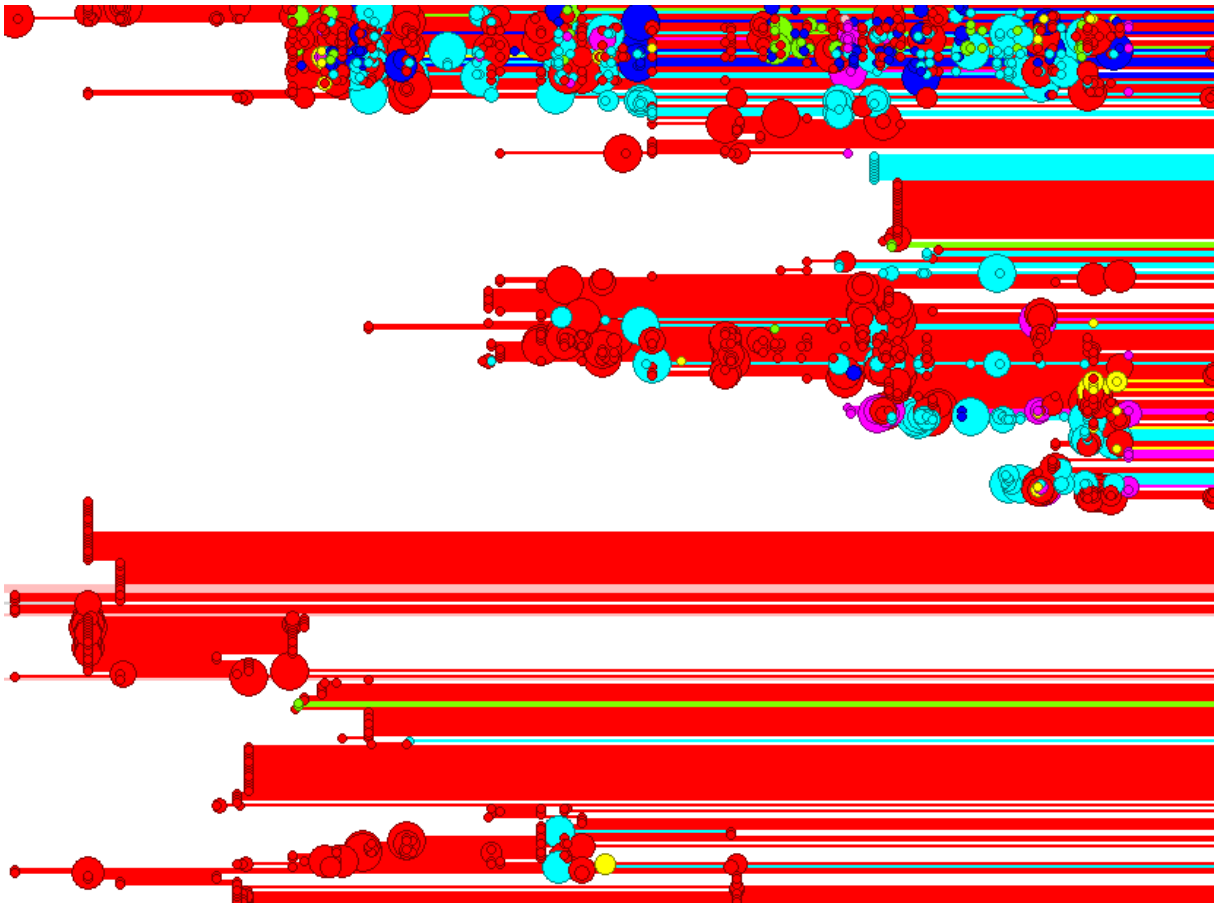

Abbildung 3: cami (red), pkofmel (cyan), man (blue), rusmus (chartreuse aka green), daniel (yellow), hispanico (mangenta), ese-staff (pink)

The team 3 can be lucky to have the red author (Camillo Bruni), because he owns almost the whole project. He added the stable data: jars and images.
In this team we see some files, which were deleted or moved to another directory (in cvs terminology: deleted and created somewhere else).
The cyan (Patrick Kofmel) and the blue (man) author interacted a lot in the upper part (model and gui). The cyan author has a goodish overview over the project, he was in almomst all non-stable parts active. The green (rusmus) and the yellow (Daniel Langone) gave some inputs, but only selective.
About the mangenta author (hispanico) is not much to say: he joined the project later and gave some inputs.

Documentation and coding style

Documentation

There is a user documentation in the cvs repository of the team; as it is addressed to users of the product and not to developers, the value for us is rather small. And there's of course the doxygen

Javadoc: Less than a third of the classes contain javadoc, and in many of the documented ones only a half of all public methods do have comments.
Sometimes, these comments are detailed, sometimes not:

```
/**
 * Returns a Track of the TrackCollection according to
 * <code>playOrder</code> and updates the <code>playPosition</code>
 * accordingly. Is used by <code>getNextTrack()</code> and
 * <code>getPreviousTrack()</code> and functions independant of
 * shuffleMode.
 *
 * @param orderPos
 *            Position of Track to play in the <code>PlayOrder</code>.
 * @return track
 */
private Track getTrackInPlayOrder(int orderPos) {
      this.playPosition = this.playOrder[orderPos];
      return this.getTrack(this.playPosition);
}

/**
 * create new tables in the db
 *
 * @throws SQLException
 */
private void tryCreateDB() throws SQLException, FileNotFoundException,
            IOException {
      StringTokenizer tokens = this.getCreateTableTokenizer();
      if (!this.dbDirectory.exists()) {
            this.dbDirectory.mkdirs();
      }
      this.conn = DriverManager.getConnection("jdbc:hsqldb:" + this.dbName, //
filenames
                  "sa", // username
                  ""); // pswd
      Statement st = conn.createStatement();
      while (tokens.hasMoreTokens()) {
            st.executeQuery(tokens.nextToken().trim());
      }

}
```

Patterns: No patterns are documented, but we encountered some `this.setChanged()` and `this.notifyObservers()`, which suggests an Observer Pattern. As some classes implement Observer and some classes extend Observable, and `this.addObserver(this)` was found, too, it seems that this Pattern is used actually. The only drawback is that the `setChanged()` method is not always called before a `notifyObservers()`, which results in the latter method doing nothing.

```
public void remove(String name) {
      for (int i = 0; i < this.albumList.size(); i++) {
            if (this.albumList.get(i).getName().equals(name)) {
                  this.remove(this.albumList.get(i));
            }
      }
      this.notifyObservers();
```

```
}
```

Code details
Lint4j: Lint4j lists 159 problems; this is the highest value found for all the teams. That this project contains most classes of all projects may not have contributed positively to this fact.

Besides Lint4j:

```
public interface MainFrameActionCommands {
public final static String CREATE_PLAYLIST = "CREATE_PLAYLIST";
public final static String EDIT_PLAYLIST = "EDIT_PLAYLIST";
public final static String REMOVE_PLAYLIST = "REMOVE_PLAYLIST";
[…]
public static final String MOVE_UP = "MOVE_UP";
public static final String MOVE_DOWN = "MOVE_DOWN";
[…]
}
```

To use this constants, a class implements this Interface and then accesses its new own constants. In Java 1.5, there was newly introduced the possibility of a static import of a class, which should avoid exactly such constructions as shown above. In addition, the order of the modifiers 'final' and 'static' is not consequent.

```
public class LibraryDatabaseAdabter extends DatabaseClassAdapter {

	public Object createNewObject(DatabaseAccessHandler dbch) {
		// TODO Auto-generated method stub
		return null;
	}

	public void deleteObject(Object o, DatabaseAccessHandler dbch) {
		// TODO Auto-generated method stub

	}

	public Object loadObject(int id, DatabaseAccessHandler dbch) {
		// TODO Auto-generated method stub
		return null;
	}

	public void storeObject(Object o, DatabaseAccessHandler dbch) {
		// TODO Auto-generated method stub

	}

}
```

Just like this class all other classes in the package ese06.team3.model.classhandlers are just empty.

```
private void updatePlayerMenuEnableState() {
      int i = this.mainFrame.getTabbedPane().getSelectedIndex();
      // State 1:
      // jTunes has no artists, no albums, no tracks and is in tarackview
      // or playlistView
      if (i == 0) {
            this.playItem.setEnabled(false);
            this.nextTrackItem.setEnabled(false);
            this.previousTrackItem.setEnabled(false);
            this.stopItem.setEnabled(false);
      }
      // State 2:
      // jTunes is in tarackview and has artists but no albums, no tracks
      if (i == 0
                  && (this.mainFrame.getMainFrameAlbumViewPanel()
                  .getArtistTableModel().getRowCount() > 0)) {
            this.playItem.setEnabled(false);
            this.nextTrackItem.setEnabled(false);
            this.previousTrackItem.setEnabled(false);
            this.stopItem.setEnabled(false);
      }
[…]
```

These blocks recur for altogether six states, and the same procedure is repeated in the same class in another method just below. Here, the team should absolutely consider the State Pattern or another Object-Oriented approach.

ESE4

Overall design, extensibility and fitting to user story/requirements

User story / Requirements

Importing files
Loose files (which are not in a folder) cannot be imported. If files are once imported, no other can be imported (if there were subdirectories).

Creating empty playlists
Works fine

Managing playlists
Works fine

Playing songs
Works fine

Persistence
The persistence does not work, if import did not work. After faulty import the application can not be started (null pointer exception).

Playing playlists
Tracks can only be played one per one and not a whole Playlist.

Shuffle tracks in playlists
As playing a whole playlist does not work this could not be testet.

Rearranging tracks in playlists
Is apparently not implemented.

Pause
Works fine

What is the overview of the system?

The class jTunes contains all the GUI, initializes a JFrame and uses a Library class for all model operations.- The Library contains all the tracks and a list of Playlists which in its stead contains a list of tracks. The former manages persistence with two classes: the very similar Saver and Loader.

Design OO-konform?
jTunes knows Library and vice versa which points out a characteristic to use the Observer Pattern. There is no general class implemented which manages lists of tracks. This may lead to duplicated code in the Library and PlayList. Last but not least the Importer class is GUI and model in one.

Designed understandable, pragmatic?
Interweaved flow of messages due to cross referencing of Library and jTunes. Library is a semigodclass which make difficult a fast understanding.

How extensible is the design?
Import and export should be added to Saver and Loader but better would be a refactoring. Because of the relations of library and jtunes, GUI-related extensions may be complicated.

How easy is it to plug in a different view (for example a Web Interface) into the application?
Very difficult, because of cross referencing again.

How easy is it to plug in a different persistency strategy (for example a Web services) into the application?
Adding a new persistence layer should be quite easy, since there are two classes which are encapsulated (Saver and Loader). It would be easier to unify these two classes to one persistence class though (to make persistence changes all locally).

BuildProcess
Does the project build?
yes (if no faulty import has been performed).

Tests and error handling

Tests
26 Tests, 1 Failure

This group has a TestSuite that runs all tests. Unfortunately, one test fails and throws an AssertionFailedError. The two grey highlighted assertions fail (the second fails if the first is commented out). But it's strange, that all tests on the buildserver run without any failure or error (and neither the PlayList nor the PlayListTest have changed since the firsrelease). In our eyes, the test has an error in reasoning: first the playlist untitled is removed and should not be there anymore. One line later, whitout creating any new playlist, the untitled playlist reappears immediately. Unfortunately there is no comment to explain what they will test exactly.

```
public void testDelete(){
      save_ = new Saver(null);
      this.playlist_ = new Playlist("untitled");
      save_.savePlaylist(this.playlist_);
      assertTrue((new File(PATH + "untitled.xml")).exists());
      this.playlist_.delete("untitled");
      assertTrue(!(new File(PATH + "untitled.xml")).exists());
      save_.savePlaylist(this.playlist_);
      assertTrue((new File(PATH + "untitled.xml")).exists());
      this.playlist_.setName("otherName");
      assertTrue(!(new File(PATH + "untitled.xml")).exists());
}
```

The test coverage is low, they test only three model classes (which are in the default packeage team4). On the other hand they test more than 26 cases: sometimes the add and remove function is tested in one testcase. It would be better to test them seperately because if the adding fails (or the first assertion fails), the rest won't be executed anymore.

Some tests are done with fictive tracks, some with real mp3-files. Some even with fictive mp3-files. This test shows, that you can add even non-existing files! And that's of course not what is expected.

```
public void testHasTrack() {
      Track track1 = new Track("Lalala", new File("C://lalala.mp3"));
      Track track2 = new Track("Schubiduu", new File("C://schubiduu.mp3"));

      lib_.addTrack(track1);
      lib_.addTrack(track2);
      ...
}
```

Error Handling

There are some try-catch blocks in the code, but any of the catch-blocks does something. Not even printing out the stacktrace. Sometimes there would be some errorhandling, unfortunately it's commented out, so I can't count this as serious errorhandling.
For example:

```
catch (ClassNotFoundException e) {
      // TODO Alert Box
      // System.err
      // .println("Couldn't find class for specified look and feel:"
      // + lookAndFeel);
```

```
        // System.err
        // .println("Did you include the L&F library in the class
        // path?");
        // System.err.println("Using the default look and feel.");
} catch (UnsupportedLookAndFeelException e) {
        // TODO Alert Box
        // System.err.println("Can't use the specified look and feel ("
        // + lookAndFeel + ") on this platform.");
        // System.err.println("Using the default look and feel.");
} catch (Exception e) {
        // TODO Alert Box
        // System.err.println("Couldn't get specified look and feel ("
        // + lookAndFeel + "), for some reason.");
         // System.err.println("Using the default look and feel.");
        // e.printStackTrace();
}
```

Besides that, the system is very robust. Adding a non-existing file don't leads the system to throw any error or do anything. The user justs don't know that he did something wrong. Strange behaviour if the track that is actually playing is removed: the track disappears from the list, but you can still hear him.

CVSStats

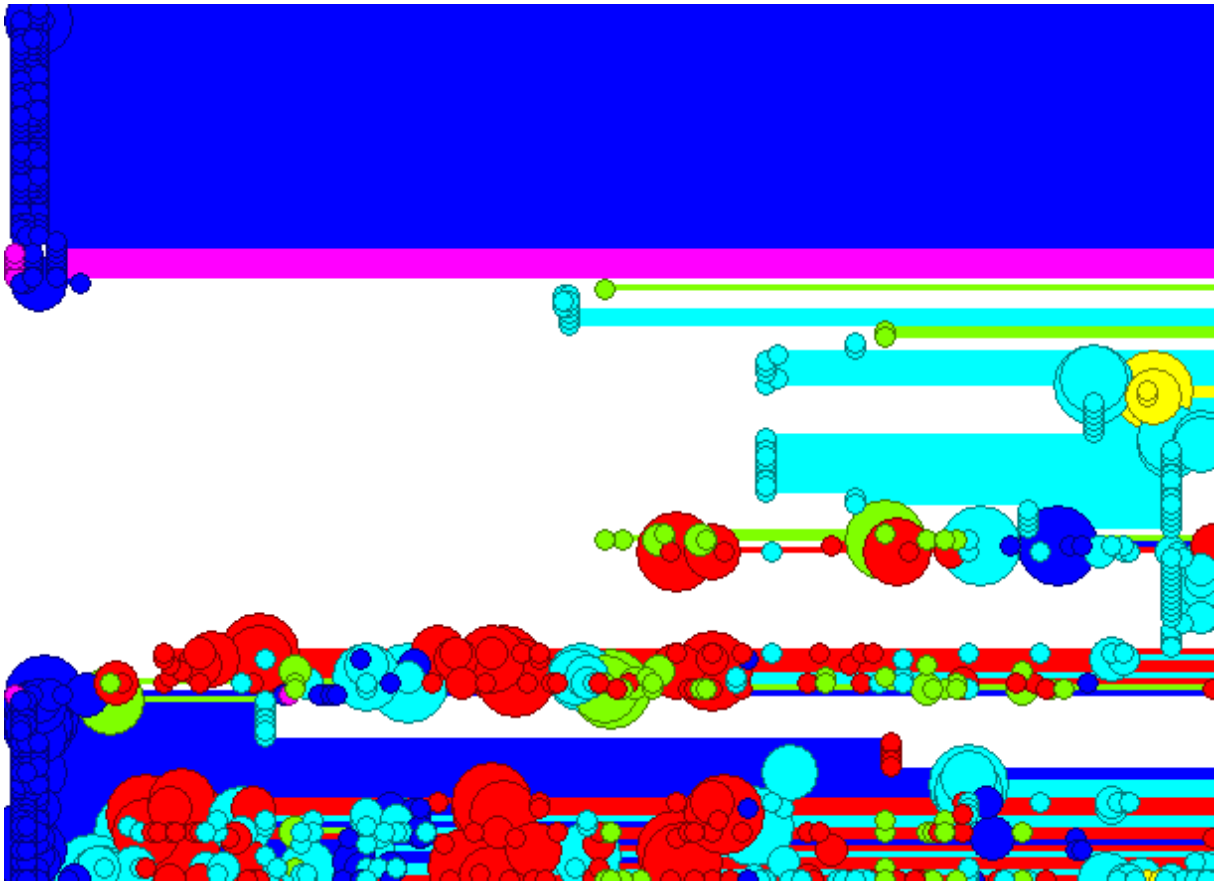

Abbildung 4: jax (blue), androwski (red), redjester (cyan), bendex (chartreuse aka green), daniel (yellow), ese-staff (mangenta)

In this team, the blue author (jax) seems to be the the main author. But the appearances are deceiving: he only initialized the project and added almost all files. A lot of them are stable (jars, taken form other projects (javazoom)) ore deleted later.
As we saw in the former three projects, the most active author was always in red. So the main author is in this team is red (androwski). He and the cyan author (redjester) have implemented the major part with inputs from the green (bendec) and blue (jax) author. The yellow author (daniel) joined the team later, and

commited only during the sprint.

Documentation and coding style

Documentation

There is a user documentation on the wiki page of the team; as it is addressed to users of the product and not to developers, the value for us is rather small. And there's of course the doxygen. But we did not find a UML-Diagram.

Javadoc: Maybe two-thirds of the classes contain javadoc, and in the majority of cases the comments are detailed enough to describe accurately what the method actually does.
Example:

```
/**
 * Removes the track from this playlist by deleting it in both lists. The
 * Tracknumber will be reassigned.
 *
 * @author ESE Team 4 - Backstreet Boys
 * @version 0.2
 * @param track
 *            The track that will be removed.
 * @tested
 *
 */
public void removeTrack(Track track) {
      this.backupTracks.remove(track);
      this.tracks.remove(track);
      this.updateMultipleTrackNumbers();
}
```

Patterns: There are no design patterns mentioned in the javadoc or in the names of classes, and it seems that there were actually no patterns used.

Code details

Lint4j: Lint4j lists 24 problems for this project.

Besides lint4j:

```
public void updateTrackTable() {
      if (this.playlistTable.getSelectedRow() >= 0) {
            String playlistName = (String) this.playlistTable.getValueAt(
                        this.playlistTable.getSelectedRow(), 0);
            Playlist play = this.library.getPlaylist(playlistName);
            // TODO Why does the TableSorter not work?
            TableSorter trackTableSorter = new TableSorter(new
            TrackTableModel(play));
            this.trackTable.setModel(trackTableSorter);
            this.updateInfoLabel();
            this.updateInfoLabel();
      }
}
```

☺ We don’t know…

ESE5

Overall design, extensibility and fitting to user story/requirements

User story / Requirements

Importing files
Works fine

Creating empty playlists
Works fine

Managing playlists
Works fine

Playing songs
The playlist does not stop after the end of the track, but otherwise it complies the requirements.

Persistence
Works fine.

Playing playlists
Works fine.

Shuffle tracks in playlists
Works fine. Although the program generates a new playlist for every shuffling. This is might be not that much userfriendly.

Rearranging tracks in playlists
Works fine

Pause
Works with odd sound cracks when restarting a song.

What is the overview of the system?

The GUI is encapsulated in NewGUI which delegates to Library and PlayList classes. - All Playlists inherit from AbstractPlayList, this is UserPlayList and CorePlayList. The latter consists of all imported tracks. Again there is that Library class, which encapsulates delegates to the core and the user list.
Persistence is guaranteed with XmlUtils which loads and saves Libraries.

Design OO-konform?
- IObservable (there would be a Observable class already implemented)
- Observer Pattern is implemented.
- Encapsulation neglected in certain cases (for instance half of the GUI initialisation in Main class)
- XMLUtil is used as a Singleton, but all the attributes and methods are static.

Designed understandable, pragmatic?
yes

How extensible is the design?
With a refactoring of XmlUtils export and import would be quite easy. With the Observer Pattern implemented, GUI changes are easy too.

How easy is it to plug in a different view (for example a Web Interface) into the application?
As there is no Observer Pattern, there have to be adaptions, but at least the Playlists are separated as a model and can be used as such.

How easy is it to plug in a different persistency strategy (for example a Web services) into the application?
One has to adapt the XMLUtil class where all functionality is locally bound. But the static singleton should be reconsidered.

BuildProcess
Does the project build?
Yes

Tests and error handling

Tests
17 Tests: 4 Errors

Unfortunately, this team has 4 errors in their tests. All four throw a NullPointerException when they want to add a track to the playlist. On the build server, even the whole testclasses PlayListTest and TrackTest fail, because the setUp fails (the build server can't convert the path to the mp3 file). This team is the only team which tested the import: no dummy file seems to be accepted. We will see later how the player handles this case.
Besides this, the testcases cover all model (core) classes except the UserPlayList. In plus they tested wheter a file or directory is accepted or not. All testcases work only with a wrapper.

Error Handling
There are try-catch blocks, but not through the whole project. Some of the set-methods have one, some not. A lot of the catch-blocks print only the stacktrace, some return false, others do nothing.

The add a non-existing file test seems to have no consequences. The import-bar shows 99%. Acctualy, the player don't cares about the non-existing files and searches the whole device for mp3-files until another import interrupts the first one.

CVSStats

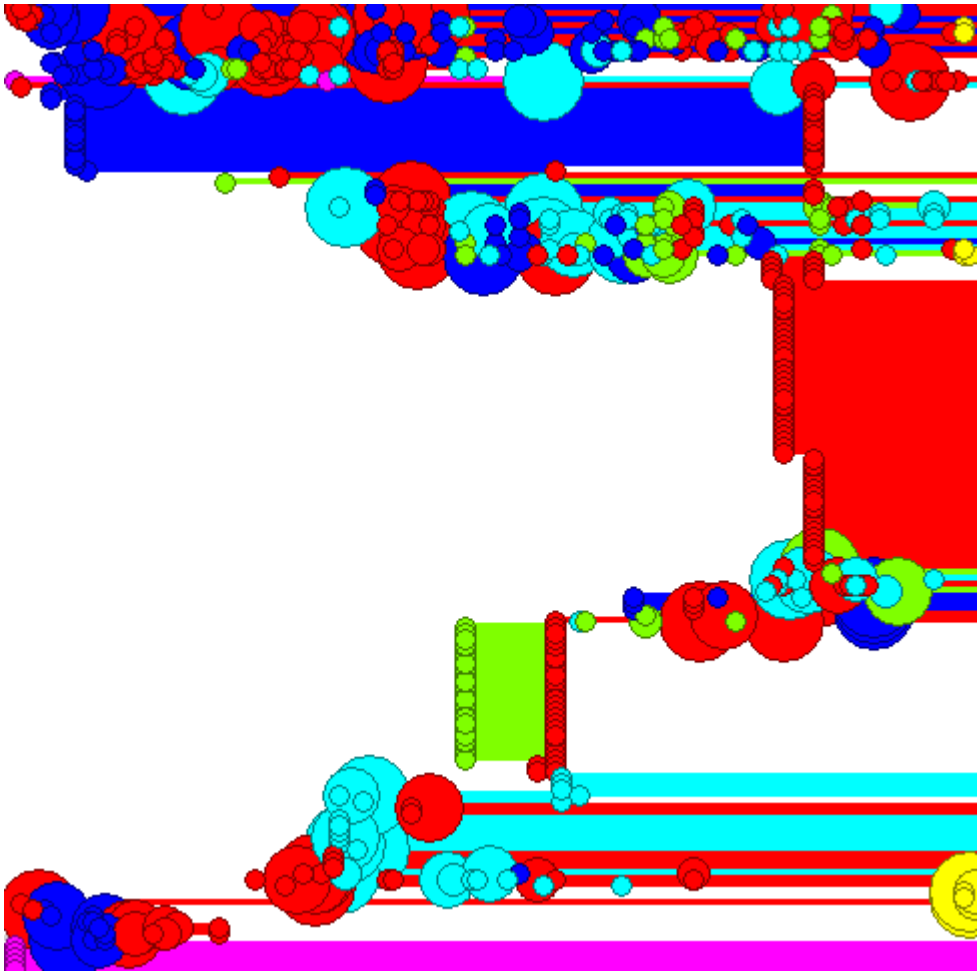

Abbildung 5: manucas (red) ubuergi (cyan), baeschtu (blue), tobiasK (chartreuse aka green), bendec (yellow), ese-staff (mangenta)

At first sight we can't say particularly something about the main author. The blue (Sebastian Bartholomé) and the red (Manuel Cassina) author wrote the core of the project. The cyan author (Ulrich Buergi) took care of the controller. The green author (Tobias Kneubuehler) instead gave only small inputs and added some stable data (gifs). Those were deleted (resp. moved to another package) by the red author. Towards the deadline, the red author tried to give the project a clear structure and separated the stable data (jars, pictures).
The yellow author (bendec) joined the team after the first release.

Documentation and coding style

Documentation

We found 2 Uml-Diagrams and 1 Sequence-Diagram additionally to the doxygen diagrams. As far as we could evaluate, these diagram are up-to-date and reflect the actual behaviour of the project.

Javadoc:
NewGUI.java: Class with most LOC, but absolutely no documentation of public methods.
Player.java: One line per (public) method, no description of class itself.
Track.java: Well documented.
About two-thirds of the classes contain javadoc. Mostly, these comments are detailed and therefore useful.
Example:

```
/**
 * Add a <tt>UserPlayList</tt> to the <tt>Library</tt>. After adding, an
 * event is generated informing any listeners to update their view.
 *
 * @param playList
 */
public void add(AbstractPlayList playList) {
     // TODO error handling if playlist is null
     this.playLists.add(playList);
     this.fireIntervalAdded(this, this.getSize() - 1, this.getSize() - 1);
}
```

The comment tells exactly what the method does.

Patterns:
One Uml-Diagram is actually entitled MVC, and we found classes named IObservable and ObservableChild. It turned out that both patterns are actually utilised in the project (and not just mentioned). An interesting detail is the use of the Observer pattern: Instead of extending the Java class Observable, they created an Interface IObservable, so that they could extend other classes and still implement the Observable Interface.

Code Details

lint4j: There are 46 problems listed.

Besides lint4j:

Some classes have strange names: e.g. NewGui.java (where is the OldGui?), newTableModel.java (there is no oldTableModel, in addition class names should begin with an upper case letter), addPlaylistGUI.java (sounds more like a method, this time the whole word 'GUI' is in capital letters and still, class names should begin with an upper case letter).

```
private ArrayList<AbstractPlayList> playLists;
```

The type 'ArrayList' is discouraged; the higher Interface List (or even Collection) should be used.

```
protected Vector<Track> trackList;
```

The class Vector should not be used anymore; instead there is the Collection Framework provided.

```
private JLabel PlayListLabel = null;
private JLabel jLabelTracks = null;
private JSplitPane jSplitPane = null;
private JList PlayListList = null;
```

```
private JMenuItem popupAdd = null;
```

There are highly inconsequent name choices: the team mixes wildly upper and lower case start letters, and the type prefixes the name, postfixes it or isn't in the name at all, and sometimes, there is a 'j' and sometimes, there is not.

```
/**
 * These constant int represents the status of the player.
 */
public final int STATUS_PLAYING = 1;
public final int STATUS_PAUSE =  -1;
public final int STATUS_STOPPED = 0;
```

Why they are not static?

```
public class Player extends Observable implements BasicPlayerListener{
[…]
/**
 * unused but listed because of implementing Observer.
 */
public void opened(Object arg0, Map arg1) {
	// TODO Auto-generated method stub

}

/**
 * unused but listed because of implementing Observer.
 */
public void progress(int arg0, long arg1, byte[] arg2, Map arg3) {
	// TODO Auto-generated method stub

}

/**
 * unused but listed because of implementing Observer.
 */
public void setController(BasicController arg0) {
	// TODO Auto-generated method stub

}
```

The reason given in the comments is absolutely not true, the methods are necessary because the class implements BasicPlayerListener. And why they are doing that we could not figure out. In addition, a few other classes also implement the Interface BasicPlayerListener, but none of the methods from the Interface are used anywhere in the project.

```
public void pause() {

	try {
		if (this.player.getStatus() == BasicPlayer.PLAYING){
			control.pause();
			this.setChanged();
			this.notifyObservers(this.STATUS_PAUSE);
		}
		else if (this.player.getStatus() == BasicPlayer.PAUSED){
			control.resume();
			this.setChanged();
			this.notifyObservers(this.STATUS_PLAYING);
		}
	} catch (BasicPlayerException e) {
		e.printStackTrace();
	}
}
```

Consider State-Pattern.

```
// TODO what does this method do exactly, and why does it always
//      return false?
public boolean importData(JComponent c, Transferable t) {
     if (canImport(c, t.getTransferDataFlavors())) {
           try {
                 String str = (String) t
                             .getTransferData(DataFlavor.stringFlavor);
                 importString(c, str);
                 return true;
           } catch (UnsupportedFlavorException ufe) {
           } catch (IOException ioe) {
           }
           // TODO implement propper error handling
     }
           return false;
}

// TODO what is this method for?
public boolean canImport(JComponent c, DataFlavor[] flavors) {
     for (int i = 0; i < flavors.length; i++) {
           if (DataFlavor.stringFlavor.equals(flavors[i])) {
                 return true;
           }
     }
     return false;
}
```

Nice TODO's...

```
/**
 * This method is actually not used here
 */
// TODO --> why is it in the TrackTransferHandler if not used in all
//          childern?
protected String importString(JComponent c) {
    return null;
}
```

Disconcerting javadoc (why does it stand here, then?)

```
protected void importString(JComponent c, String str) {
     if (str == null) {
           return;
     }

      // TODO why not load in the constructor?
     loadActiveElements();

     if (this.activePlaylist == this.library.getCoreList()) {
           return; // The coreList already contains everything!
     }

      String[] paths = str.split(TrackTransferHandler.TRACK_SEPARATOR);
     for (String path : paths) {
           this.activePlaylist.add(new Track(path));
     }
}
```

'return;' (done twice) is bad coding style (goto-like behaviour)

```
public void addAt(Track track, int index) {
	//	TODO implement propper error handling
	assert(track != null) : "no track to add"; // pre-condition

	this.trackList.add(index, track);
	// set to given position
	this.activeIndex = index;

	this.fireTableRowsInserted(index, index);
	XmlUtils.savePlaylist(this, XmlUtils.UpdateEvent.ADD);

	assert(this.inRange()) : "index not in range"; // post-condition
}
```

Pre- and post-condition checks: Design by contract; very nice, unfortunately done very rarely in the project.

```
/**
 * method to get all the ID3Tag info as an array
 * index selects the attribute:
 * 0 -> Track number
 * 1 -> Artist
 * 2 -> Title
 * 3 -> Album
 * 4 -> Year
 * 5 -> filename
 * @param index
 * @return String
 */
public String getTagInfo(int index){
	String[] tagInfo = new String[6];
	try {
		if (index <= 5 && index >= 0) {
			tagInfo[0] = Integer.toString(this.mp3Tag.getTrack());
			tagInfo[1] = this.mp3Tag.getArtist();
			tagInfo[2] = this.mp3Tag.getTitle();
			tagInfo[3] = this.mp3Tag.getAlbum();
			tagInfo[4] = this.mp3Tag.getYear();
			tagInfo[5] = this.fileName;
		} else {
			// TODO is this a good idea?
			return null;
		}
	} catch (NoID3TagException e) {
		e.getStackTrace();
	}
	return tagInfo[index];
}
```

Add constants holding those dangerous - as they might change in this method and not in the clients - integer values.