Introduction

Product Family: Music Players

Team Members and their Products :

a. Krupa Merupu – VLC Media Player
b. Maithili Parikh– Songbird
c. Praveen Kumar Medapati – Banshee

The product family being investigated are Music Players. They come in many shapes, sizes and flavours and contain infinite combinations of features, suitability of each being entirely subjective to a User. Therefore, to create a reference architecture for this product family, we begin comparing and contrasting the various features that the above Music Players share, along with the components that allow each to successfully operate. This means, comparing the previously drawn up Module View, Allocation View and Components and Connector View to draft a reference architecture which can be applied to all.

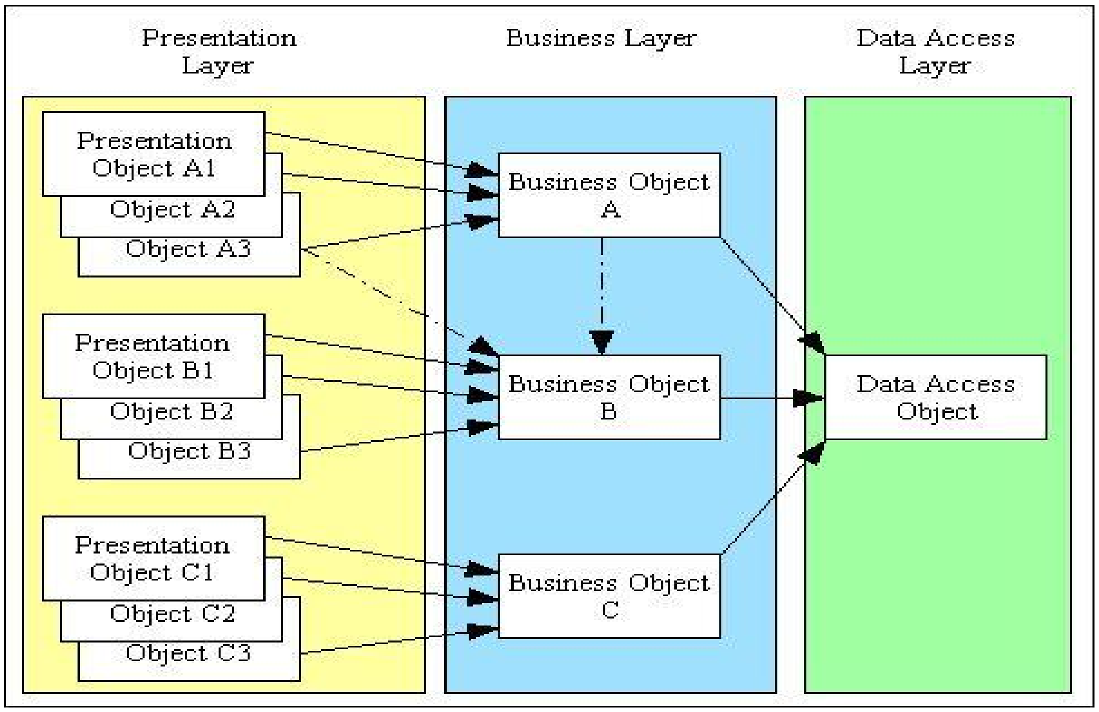
In this way, a reference architecture encompasses an abstract terminology i.e. it is applicable to all members of the family, but is not specific enough or restrictive enough to only apply to certain member. For example, and architecture developed for this family must contain a module, component or layer which allows for external communication as all 3 products being investigated make use of it. This common feature/functionality is called an invariant i.e. one that is true for all members of the product family. Conversely, each mechanism used to connect to the Internet, or the purpose of such a connection is different for each of the product. This difference is referred to as a variant.

Thus, a reference architecture comprises of the invariants, which must hold true across all members of the product family, and the variants, differences that may exist between each member of the product family. A relevant example of this is how VLC, Songbird and Banshee handle loading the tracks. Both Songbird and Banshee are a query based player which uses metadata to load tracks for playing. VLC, the track can be directly loaded by using queries compatible with the database language. Therefore, the retrieve feature is an invariant, while the mechanism used to satisfy the feature is variant.

Product Family and Reference Architecture

The Music Player architecture has many members, and are constantly evolving based on the popular technology and services required by Users. Despite this, analysis has proven that there is one distinct architecture type which can be used as a reference when designing, or analyzing any such player. The 3-Tier architecture, consisting of the Client Tier, Internal Tier and Back-end Tier is the best option when it comes to a reference architecture. The Architecture, given below, was used as a reference. The reference architecture Is built with the intention of being able to easily understand and explain any version of Music Players, and thus enable easy analysis.

Music Players, in this context are Open – Source, generally written in C++ or some similar Object Oriented Programming language. However, they may or may not be operable on certain platforms. For example, Most Windows and Linux operating systems support C++ programs. However, Windows platforms might make installation more difficult. Lastly, Mac OS does not support C++, thus the availability will depend on whether there exists a compatible source code.



As seen in the above diagram there are 3 main tiers or layers of a 3-tier architecture which are as follows:

Presentation tier:

This is the topmost level of the application. The presentation tier displays information related to how the user interacts with the system. It communicates with other tiers by which it puts out the results to the client tier and all other tiers in the network. (In simple terms it is a layer which users can access directly such as an operating systems GUI)

Application tier (business logic,logic tier, or middle tier):

The logical tier is pulled out from the presentation tier and, as its own layer, it controls an application's functionality by performing detailed processing.

Data tier:

The data tier includes the data persistence mechanisms (database servers, file shares, etc.) and the data access layer that encapsulates the persistence mechanisms and exposes the data.

Main Features

- Play

The fundamental feature associated with Music Players, this feature should allow a User to access and play an audio file.

- Search

This feature is present in a majority of the Music players available, however it may not be an existing feature in all. This feature is responsible for retrieving the audio files as per User requirements e.g. a Track name entered by the User should yield a track ready to be played. Often, this feature is coupled with a catalog feature. A catalog feature allows the software to populate a local "library", to allow for easy Search retrieval.

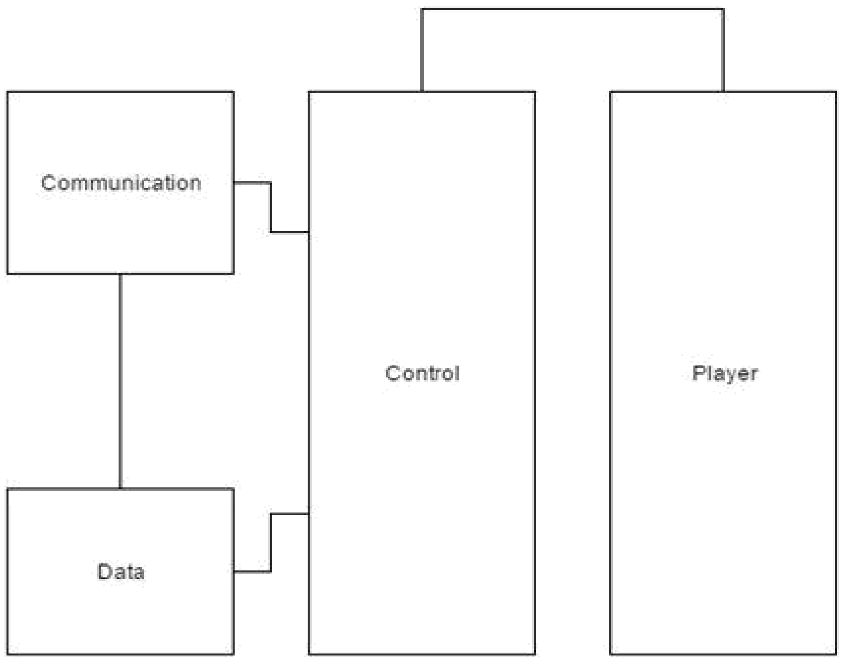
Main Interactions

- Play Music – User chooses which file to play, in which order.

- Create Playlists – User can manipulate the order in which files are played by the music player.

- Search – User should be able to retrieve a particular file from a compiled library.

Module View



Modules

1. Player – module responsible for the actual task of accessing and playing the music file requested by the User. This could either be by retrieval using the meta-data as an indicator, or playing the audio file provided. This operation is dictated by the source code.

2. Control – Manages and dictates what is to be played by the Music player. This module is responsible for handling the interactions between User and Data store, it takes in User Interactions and provides appropriate results. An example is handling search queries.

3. Data – This module is managing access to local and online databases. Its main interaction with the Control module, is to create a result list which satisfy the query provided. It only handles retrieval of data given a particular query.

4. Communication -This module is responsible for the external communication of the software. In such a case, it is used for the Data module to communicate with the online sources. Furthermore, some music players connect to external sources for maintenance purposes and this module can also facilitate that.

Invariants

1. Data – The data module, described by the module view, is responsible for handling a collection of meta-data or other such data, on which searches can be performed locally. This is

invariant, as this mechanism is absolutely necessary to provide the Search feature, and the Search Interaction.

2. Player – Similar to Data, this is a fundamental module whose existence in the view cannot be altered. This module is responsible for the task of accessing and playing the audio file provided. This satisfies the main feature Play.

3. Communication – This is a module which adds on the functionality of communication with external sources. This means, that any data stored online, or any online storage data can be accessed via this module. Thus, its existence is mandatory and the connection between the Data module and this module. Furthermore, even if no external sources are used, an external communication module is necessary for maintenance.

4. Control – This is the central module, which manages and facilitates interactions between the User Interface and the program. As a result, this module handles both the Search and Play features. Search is handled by communicating with the Data module, then the Play module. Thus, this module must exist and facilitate the flow of data and User Input amongst the program. 5. Control to Data – This connection, between the Control module and the Data module is necessary to facilitate User searching for a particular file.

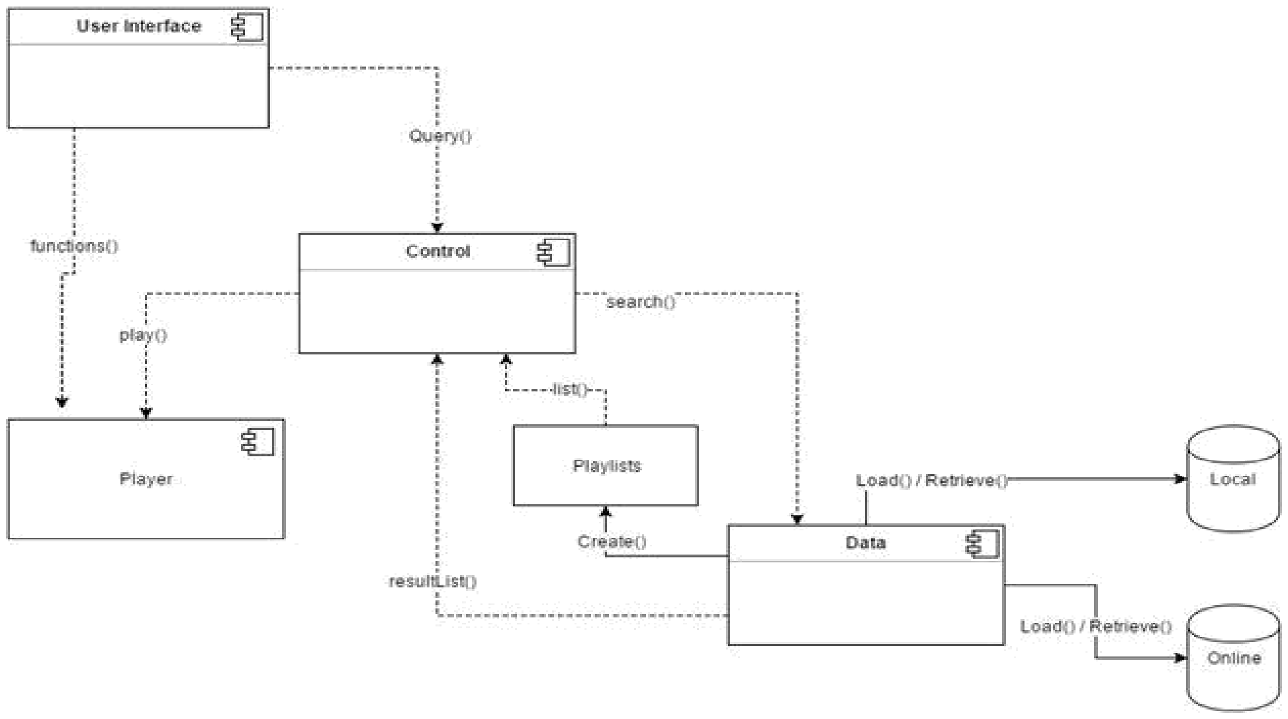
Variants

1. Communication – While this module was identified as an Invariant, the internal structure of the module is a point of variation. This is because, depending on the functionality, there could be many different structures, sub-modules or dependencies which encompass this module. For example, streaming services could require individual sub-modules within the communication modules to provide functionality. Thus, the internal structure of this module is a point of variation.

2. Data – Similar to the Communication module, the internal layout of this particular module could be different depending on where the data is being gathered from and then how it is stored. Some players use meta-data, while some use hash-table and others use something else completely. Thus, depending on the storage method, the module is different across the product family. A similar mechanism is at play in the Player module. This is again because of the procedure used by each player to retrieve and play a particular find. Some players use meta-data during processing and only load the audio files at the last possible opportunity, while others load the audio file during processing. Thus, depending on the way in which the

3. Lastly, the connection between the Data and Communication modules is a point of variation. This is because, if the music player is using online sources, then a connection to the Communication module is required, while if only local sources are used this connection is meaningless. This varies across the products in the family and is not an invariant.

Components and Connectors View



1. User Interface – Responsible for allowing a mechanism for the User to interact with the Music Player. Various functions involved, and influence operations of the Player via the functions() connector. This Component also allows the User to inform the Control component of the user search results using Query() connector.

2. Control – Handles Queries from User and results. Queries are provided via the Query() connector, then search() procedural call connector is used perform search via Data component, and the resulting data received via the resultList() connector, then passed to the Player component via the play().

3. Data – Communicates with the Local and Online storage via Load() and Retrieve() to create a library of music files. This library is then searched to satisfy queries from the Control component.

4. Player – Receives items to be played, using the play() connector, then accesses the file and plays it as directed by the source code. Can be influenced by the functions() connector with operations such as Play, Pause, Next etc.

Invariants

1. Functions() – This is a necessary connector to facilitate communication between the User and the Player. With this connector, the user can use the UI to manipulate the operations of the Player component. Thus, it is an invariant.

2. Query() – Similar to the Functions() connector, this is needed by the software to facilitate searches by the User. This means, the connector takes in User search parameters, then passes

them to the control unit. This communication mechanism between the Control component and UI is invariant.

3. Search() and ResultList() – These two connectors, are cyclic and perform the Search function. This means, they must exist so as to provide the Search feature to the Music player. These can also be represented as a single procedural call, but essentially satisfying the same requirements.

4. Control – This is the central component which coordinates the various other components that make up the architecture of the software. As a result, some mechanism exists in every member of the product family that handles this. It could be as simple as changing the directory of the Data store, to something more complicated such as codecs or plugins. Thus, it is an invariant.

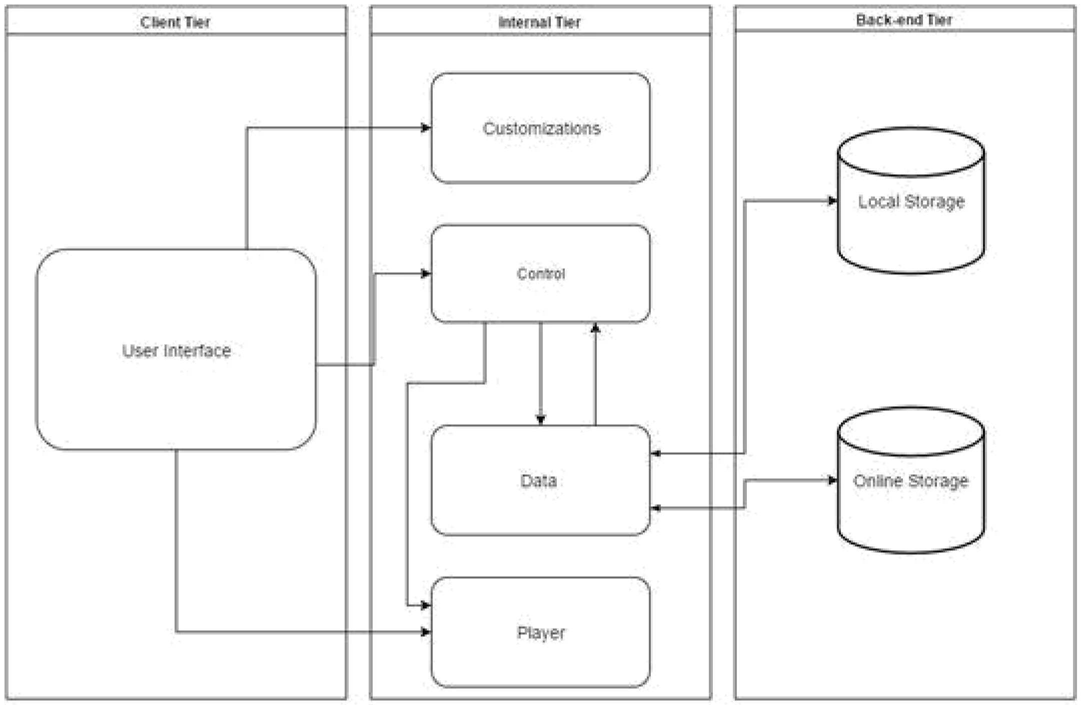
5. User Interface – The UI is an invariant. It is necessary to provide interactions between Users and the software. As a result, every single music player has some Interface to facilitate this interaction.

Variants

1. User Interface – Sub – attributes within this Component are points of variation. Each music player offers up a range of features on the User Interface which a User can access. For example, tomahawk offers a Charts attribute, while Clementine and VLC do not. In this way, the User Interface component, while being a invariant, contains many possible points of variance. 2. Data – This component is responsible for cataloging, coordinating and managing audio files via a “library” system. This system happens in all Music players. However, the process used to compile this library can differ between music players. Furthermore, depending on the sources (online or local), the operations and connections of this component could be different.

3. Load()/Retrieve() – These connectors are a key variation point. This is because, some Data systems simply retrieve files from a pre compiled database. Also, some systems are not responsible for the loading of data into these “libraries”. Thus, these connectors may or may not exist depending on functionality provided by the software.

Allocation View



Client Tier: This tier is responsible for the main Interactions between the User and the Software. As such, it contains the User Interface only. Sub – modules exist within the User Interface which interact with specific modules in the Internal Tier.

a. Allows an Interface to modify the Customization module.

b. Allows the User to interact with the Data module in the Internal Tier.

c. Allows the User to directly Interact with the Player module. This means, the User can directly intervene in the operations of the Player module in the Internal Tier. One such example is when a User uses the User Interface to skip to the next track on the list.

Internal Tier: This tier represents the internal workings of the Software. This Tier and its sub – modules are responsible for the operation of the software in the background i.e. this means that the operations that occur in this tier are not visible to the User, and cannot directly be modified by the User.

a. Customizations – This allows the User to customize their “experience” with the software. In other words, this sub-module allows changes to the Internal Tier of the software to satisfy User preferences. For example, to modify the location from which Tracks are loaded. It may also alter operations of the Internal Tier e.g. Add more Codecs.

b. Data – This module has multiple responsibilities. Firstly, it retrieves data from the local or online storage module. Secondly, it may be responsible for cataloging this data store in an efficient way. In some cases, this module may also be responsible for storing data to the local data store.

c. Player – This module is the core of all Music Player software. Essentially, it is responsible for playing which ever track is fed to it. This may involve either retrieving the actual track, once given the meta-data then playing it, or playing the track is provided. The operation of this module can be controlled via the Controls sub-module in the User Interface.

d. Control – This module is responsible for the main operations of the software. For example, data retrieval from the Local Storage, based on a User Query. This is the module which is responsible for deciding the operational parameters of the music player.

Back-end Tier – In a 3-tier Architecture layout as above, the Back-end Tier represents the Data Stores required by the Software to function successfully. The components are:

a. Local Storage – This is a locally stored compilation of the Music files. Locally may refer to on the same hard drive, or a fixed network or a data store that does depend on external connectivity. Depending on functionality of the software being reviewed, this local storage could be made up of either the actual Tracks (audio files) or a “database” of the meta-data associated with the audio files.

b. Online Storage – This type of storage is becoming more popular in recent times, and this functionality is being incorporated into more and more music players. This Online/External storage is one that requires some form of authentication from User before files are open to access. This could be an online streaming service such as Spotify, which allow Music players to access and store meta-data as per User requirements, or Cloud Drives such as OneDrive, which store the actual music files saved by User.

Invariants

1. User Interface – The User Interface component is a necessary part of the architectural layout, and thus is predominant in the 3 Tier architecture. As it is the only point of contact between the User and the Software, and allows the User to use the software it is invariant. It plays a part in implementing the Play and Search features.

2. Control – This control module, as in the previous views, is responsible for coordinating the various components of the software to provide functionality. Thus, is an invariant. Plays a role in Search functionality.

3. Data – This is used by the software to facilitate the Search function. As a result, some form of this component exists in all variations of the Product Family.

4. Player – The fundamental component of the Product Family, it is responsible for accessing and playing the audio files. Invariant, as its despite tweaks to its functionality, it performs the basic function of accessing and playing audio files.

5. Architecture – 3 Tier Architecture – Most modern music players, have a basic set of functionality that they provide. Generally, these can be summed up in the 3 Tiers that are displayed above. A clear point of interaction, or Client layer, an internal processing layer and a

data layer. This view is the most accurate way of displaying this, and thus the design itself is invariant.

Variants

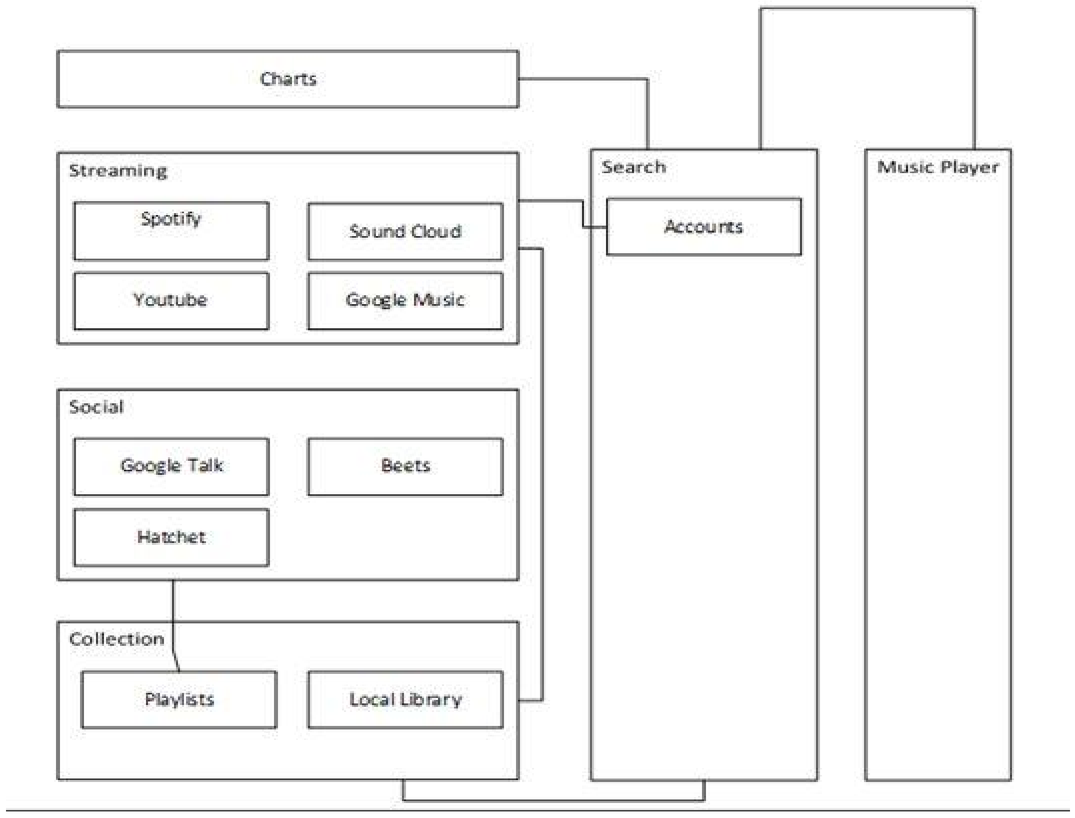
1. User Interface – Different sub-modules are possible, based on functionality thus making this a point of variation.

2. Customizations – This components existence is a variant. Some music players offer customizations, while others do not.

3. The flow of data between the components Data and Control may differ based on functionality. Similarly, the data flow between Data and Player might also optionally exist. This means, that based on functionality, such as at what point the audio file is accessed, there could be more or less connections between Data, Control and Player. Thus, this is a point of variation to satisfy the Play feature.

Product: Tomahawk

Module View

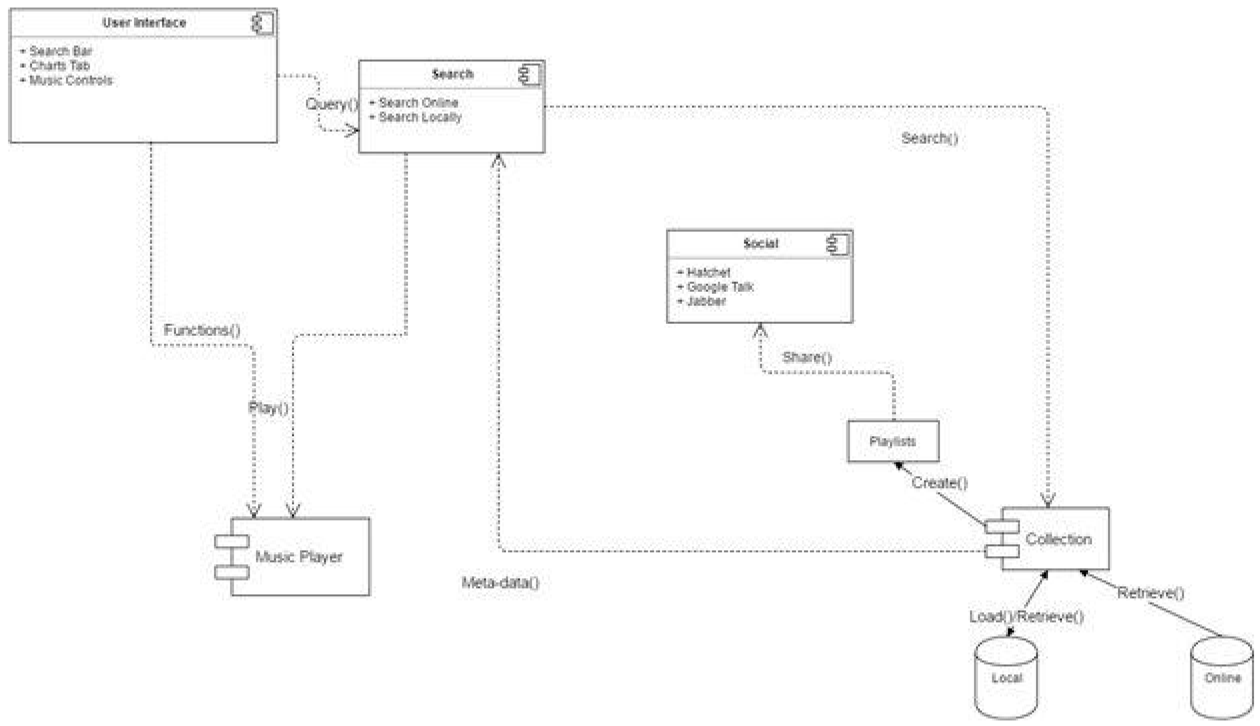


Comparing the module view of Tomahawk to that of the reference architecture, shows that there are indeed many similarities, but also some differences.

In terms of invariants, the Search module above, is similar to the Control module indicated in the reference model view. This is because, the Search module performs the same functions as the Control module namely, it facilitates the transfer of data and instructions to keep the program operating as desired by the User. This can be seen in the connections between the Social module, Collection module which performs similar functionality to the Data module in the reference architecture, and the Streaming module which behaves similarly to the Communication module. Furthermore, the similarity also extends to the Player module. These similarities then, satisfy the invariants.

Next we compare the variation points. First, the sub-modules present in the Communication module (Streaming module in this case) are different, and can be organized differently per product in the family. This means, that where there is a variation point, variation is possible and often occurs across members of the product family. Similarly, the Data module (Collection) consist of various sub-modules which provide the sources necessary for successful playing of a file. Moreover, there exists a relationship between the Data module and the Communication module allowing for an external source to be incorporated into the program. These variations, occur within the variation points and so still relate to the reference module view. However, the Charts module is neither an invariant nor a variant.

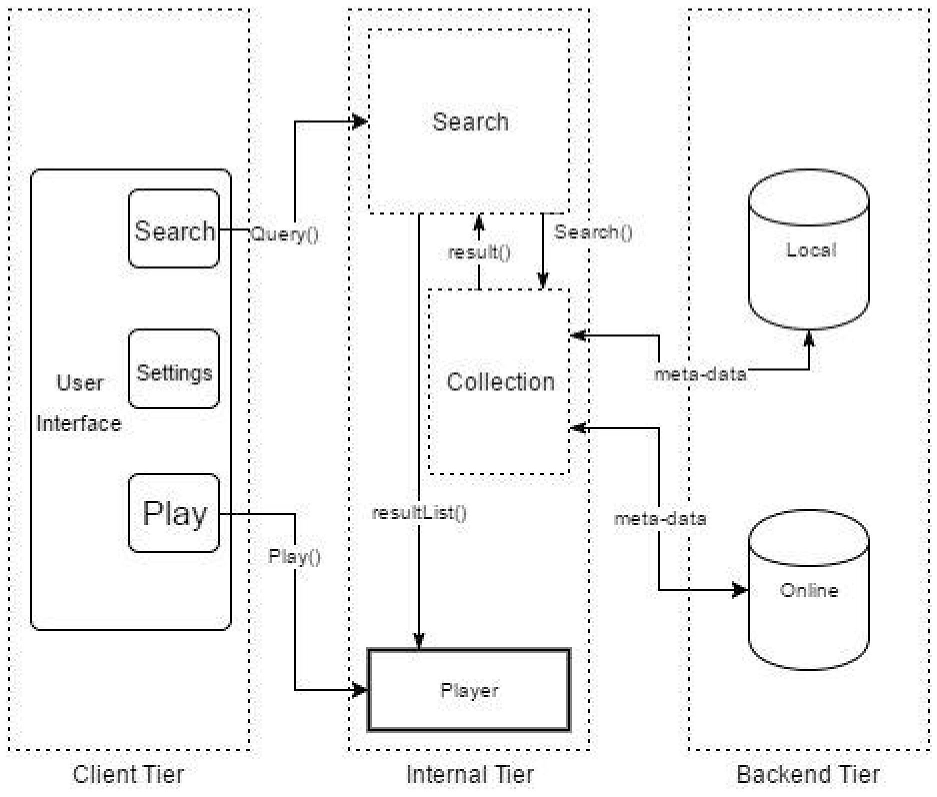
Components and Connectors View



This view primarily represents the data flow in the scenario when a User selects a song from the library and plays it. The data flows from the UI, to the Search component which retrieves data from the Collections component before passing it to the Music player to play. Since this feature is fundamental in the Music Player product family, the reference Component and Connector diagram follows a similar path. This makes comparing the two extremely easy. Firstly, the invariants; functions(), query(), search(), resultList() are connectors which exist in both the reference view and this view specific for Tomahawk. While a lot of the connectors share names, resultList() is called meta-data() in Tomahawk. Then, we can also compare the Control component and the UI component. Apart from the attributes, they are the same as that of the reference architecture.

Subsequently, we can discuss the variation points. The attributes within the UI component are different. They definitely vary across the various players. In addition, the Collection component (Data) in this case has the same Online and Local connections as the reference version, but it may not always be the case. Lastly, the Load()/Retrieve() functions are different because of the type of Online mechanism offered by Tomahawk. Since Tomahawk can only retrieve online meta-data, only Retrieve() exists between Online database and the Collections module. However, this may be different if the Online source was a cloud storage system.

Allocation View



The allocation view maps the functional objects that comprise the software, onto physical components on which the software is run. Thus, this 3 tier architecture above, describes the Client Tier which contains the UI component, this being an invariant. Further, it describes the internal operations of the software in the Internal Tier. In this tier, the Collection module and Search module, perform the same functions as the invariants Data and Control. Similarly, the

Player module is also an invariant. Lastly, the fact that the structure is 3 tier, comprising of the Client tier, Internal Tier and the Backend Tier is also an invariant.

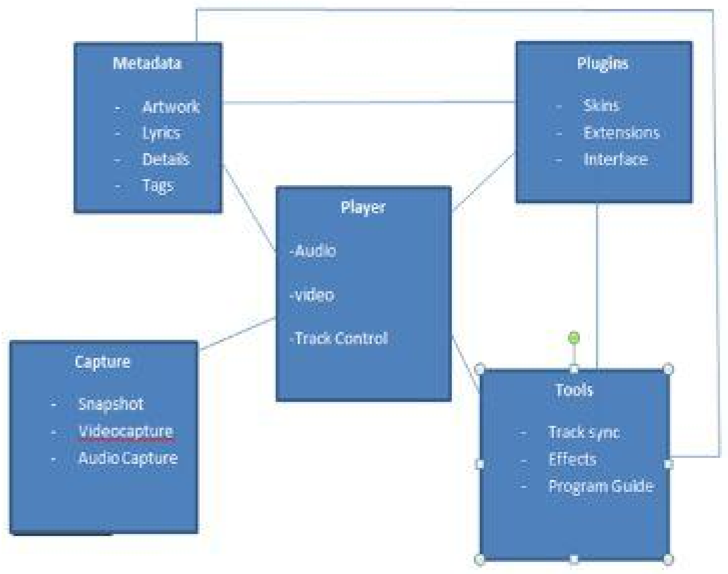
However, within the variation points, variations do occur. As such, the sub components within the UI module are different for Tomahawk, and are different across members of the product family depending on the level of customization or functionality offered. In addition, the Customization component is not present in the Internal Tier. This is missing here because Tomahawk does not offer that functionality and actively limits the amount of customizations possible. Lastly, the point of variation that exists between the Collection module and the Search module follows that described by the reference architecture. However, this also changes from product to product.

Conformance

When evaluating conformance of a software architecture, we compare the reference architecture and the associated view against that of the product being tested. Thus, Tomahawk can be classified as being Conformant. This means that all features that are implemented in the software are done according to the specification. This is in regards to features such as Search and Play, which are clearly described in the reference architecture. However, this level of conformance also says that this implementation has features that are not dictated by the reference architecture. This is in reference to the Charts module, described in the module view which does not exist in the reference architecture. Similarly, this also applies to the Social module in the module view. Since none of these are implemented in the reference architecture, Tomahawk is classified as a conformant product rather than a fully conformant one.

Product: VLC

Module View



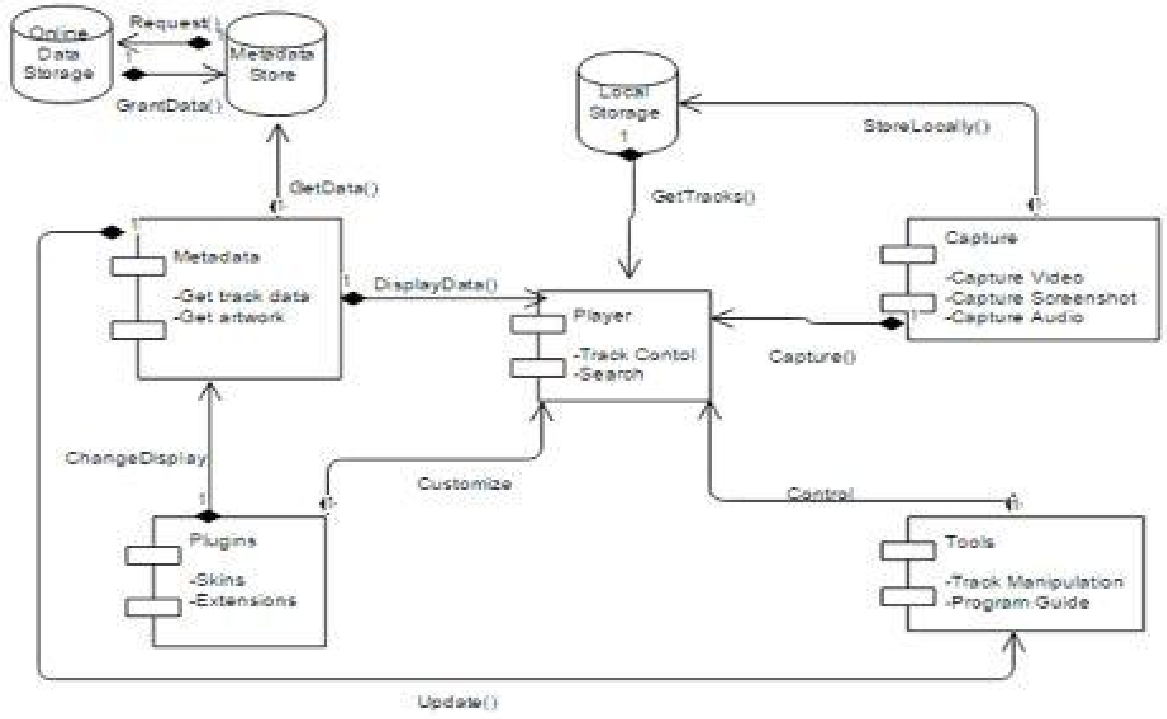
The module view of the VLC music player has a few similarities with the reference architecture provided above, although the naming conventions used for the modules are different. In the reference architecture, the player module is similar to the player module in the VLC media player and performs the function of accessing and playing the music file requested by the User. This could either be by retrieving the metadata,, or playing the audio file provided. Here too, the source code is responsible for actually carrying out this function. As a result, this can be categorized as an invariant.

In the reference architecture, we see another invariant in the control module, which is responsible for managing the media played by the Music player. It also handles the interactions between User and Data store and takes in User Interactions and provides appropriate results. This is very similar to the “Tools” module in VLC media player which has many specific operations such as track synchronization, adding audio and visual effects to the track etc.

Yet another invariant is the data module which manages access to local and online data stores. This is again similar to the metadata module in the VLC music player which may use queries to request data either from an online data store on a local store.

The only real difference which constitutes a variant between the reference architecture and the VLC module view is the absence of any separate module for communication. All the communication is managed by the source code mainly through the use of queries, whereas in the reference architecture the communication is assigned to the communication module to access online data resources.

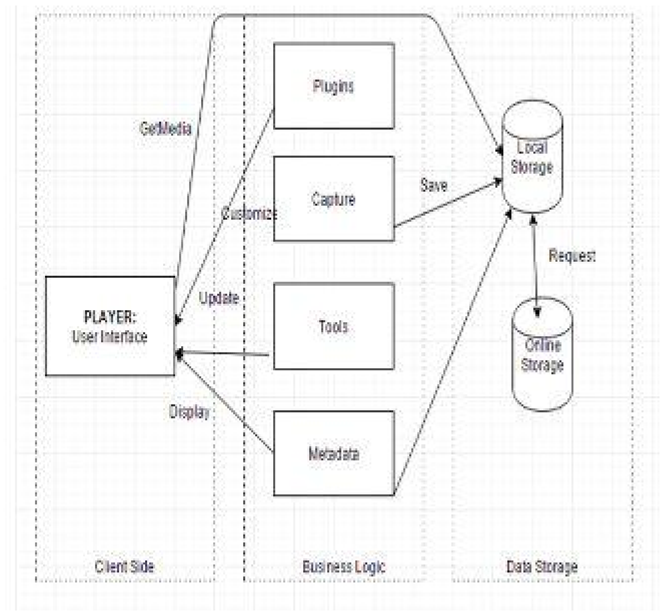
Components and Connectors View



The components and connectors used in the VLC media player architecture bear a striking resemblance to the reference architecture provided previously. However, a variant arises because instead of having separate modules for both UI and player as in the reference architecture, we have both of these components available in VLC as a single, high level module called the player module. Even certain aspects of the control of the media being played is performed by the player module itself and is carried out by the source code which makes up this module. Other tools are available which allow for track manipulation such as effects.

The data aspect of the reference architecture uses queries to send and receive data from a data store, which in turn can access this data from both local and online stores. By using the load() and retrieve() functions (in this case named as request() and grantdata() respectively) data can be transferred back and forth. This is an invariant.

Allocation View



This view in the VLC player supports the 3-tier architecture (which comprises the client side, business logic and the backend data storage) as in the reference architecture, which satisfies as a variant with respect to the reference architecture. However, as there is a division between the player and user interface/player in the reference architecture whereas both these are present in the same module for VLC, this constitutes a variant.

The other modules present in the middle tier or business logic tier of the reference architecture are also present in the same tier of the VLC architecture under different names. Hence, this aspect too conforms with the reference architecture as an invariant. The plugins, tools and metadata in VLC refer to the customizations, controls and data modules in the reference architecture respectively.

A variance occurs in the data storage tier of the VLC player wherein data is only requested by queries if data requested is not found in the local storage whereas in the reference architecture it can be requested directly by the middle tier.

Conformance

It can be said that by comparing and referring the reference architecture with the various architecture views (allocation view, module view and components and connectors view) of the VLC media player, it is in conformance. This is because most of the features laid out in the reference architecture are present to a high degree even in the VLC media player but this does not mean there is no scope for some degree of variance. The developers of this application have decided to develop VLC media player in such a way that although the major aspects of the player conform to the reference architecture, there are some aspects such as the inclusion of the UI and player as a single module and even the way the data is retrieved from an online store, do not conform to the reference architecture.

Product: Clementine

Module View:

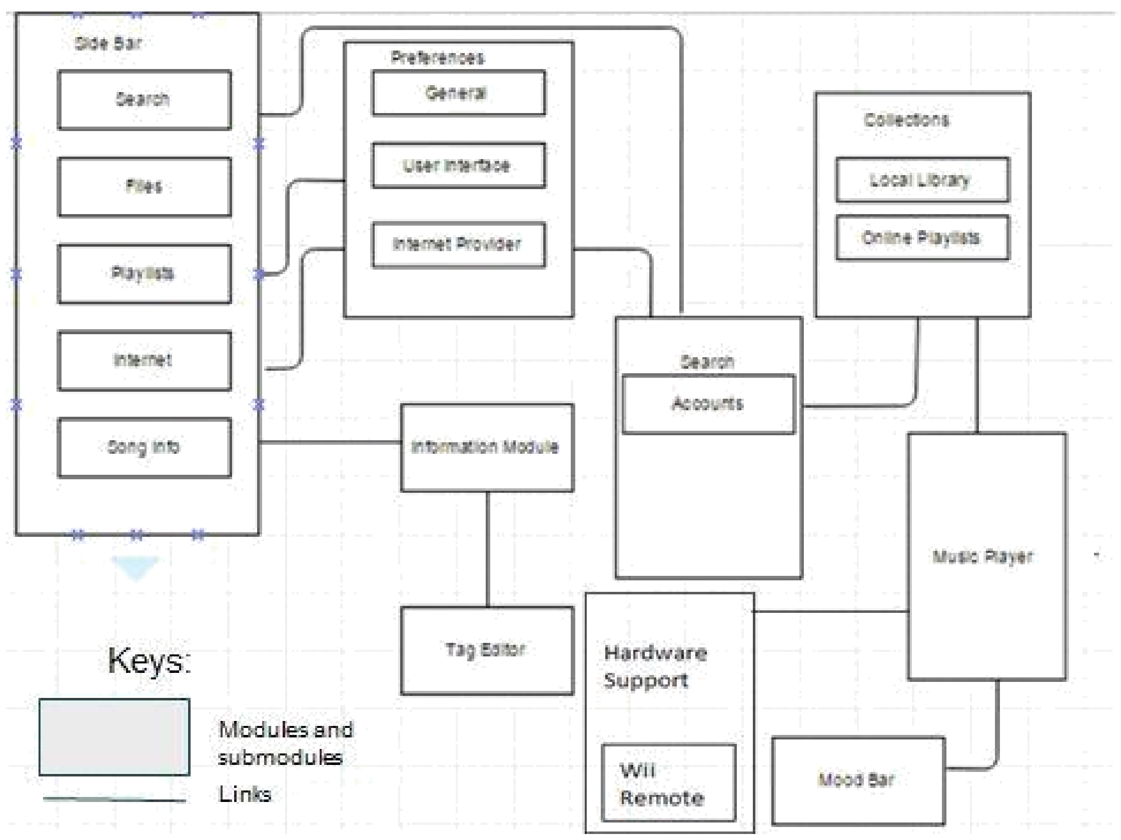


Fig. Module View Diagram for clementine

As we can see in the above diagram clementine consists of various modules which can be compared with the 3-tier architecture reference model which has been explained above.

There are various similarities and dissimilarities of clementine's module when compared to the reference architecture module. As we can see the hardware support module can relate to the control module of the reference architecture while the information and collections module can relate to the the data modules whereas the side-bar module can relate to the communications module of the actual reference architecture. Also the music player extends the similarity of the play module in the reference architecture. This modules does satisfies the invariants.

Now comparing the variants, modules such as side-bar which is similar to the communications module contains different sub modules than the modules used by other products in the family for communication. Also for visualization effects clementine contains a mood bar which is absent in the other products. Clementine also supports external hardware control which cannot be seen in other products of our product family.

Components and Connectors View:

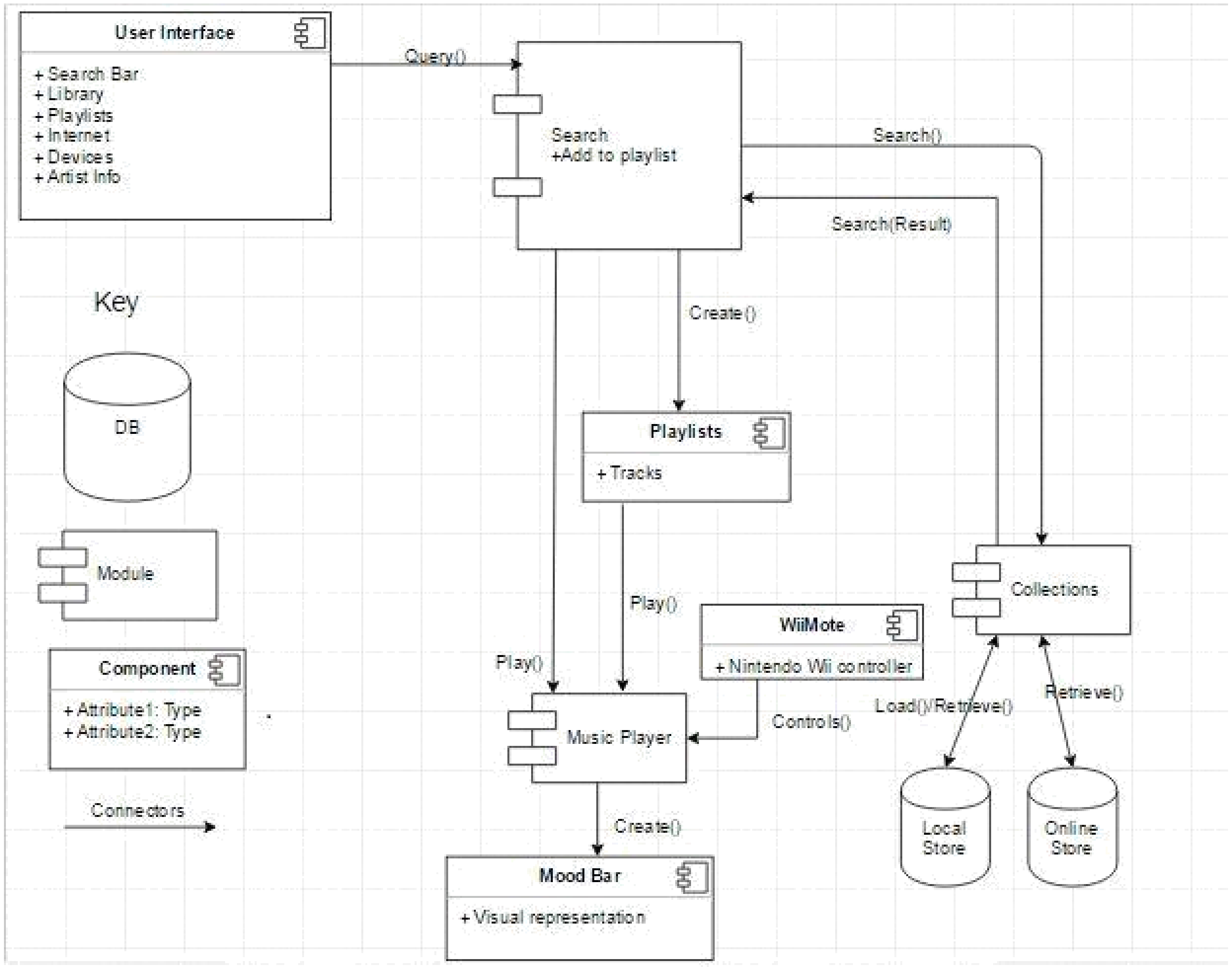

User Interface
+ Search Bar
+ Library
+ Playlists
+ Internet
+ Devices
+ Artist Info
Query()
Search
+Add to playlist
Search()
Search(Result)
Create()
Key
DB
Playlists
+ Tracks
Module
Play()
WiiMote
+ Nintendo Wii controller
Collections
Component
+ Attribute1: Type
+ Attribute2: Type
Play()
Music Player
Controls()
Load()/Retrieve()
Retrieve()
Connectors
Create()
Local Store
Online Store
Mood Bar
+ Visual representation

Fig. C&C View of Clementine Music Player

The above diagram shows how exactly the play request is handled by the clementine music player. As mentioned above the play request is the fundamental feature of our product family and this request is handled in a very similar way as discussed for Tomahawk. The search() request is passed to the collections module which returns search(result) to the search module for it to pass to the music player for playing the file. There are various variants as compared to one above such as the UI component contains various other sub-modules then the ones available in the other products. Also the play request creates a mood bar visualization for each song and this gets stored in the database. Apart from this clementine provides an external hardware support to control the music player.

Allocation View:

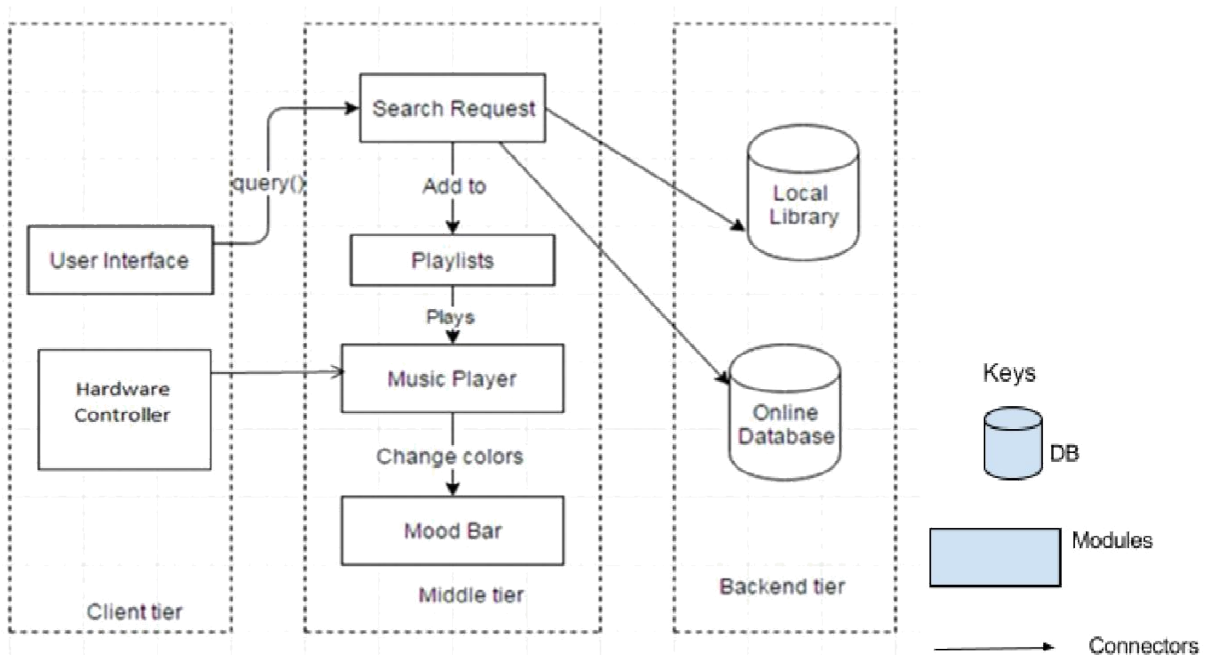


Fig. Allocation View of Clementine

The diagram above shows how the functional objects of clementine are deployed on the physical components on which it can run. As seen above it follows a 3-tier architecture having three tiers as the client tier, the middle tier, the backend tier. The client tier consists of the UI component and the hardware control which is used by the user. The middle tier contains various processing components such as search, player and mood bar which is used to create visualizations. At the end is the backend or data tier which are used by the search component to retrieve the requested data. This kind of architecture can also be termed as invariant, The variation points in clementine as compared to others are the hardware controller component and the mood bar which are not seen in the other two products of our family. On the other hand another variant is the customizations option which is available in VLC but is absent in clementine.

Conformance:

As mentioned earlier while evaluating conformance it depends on how closely the product, which is being tested in our case Clementine, relates to the reference architecture and associated views that are discussed earlier in this document. Clementine follows the reference architecture in terms of various key modules such as search, Player which can also be seen in the reference architecture. Thus, clementine can also be termed as an conformant. The level of conformance really depends on how much the product follows exactly the same modules as in reference architecture. So when we compare the Mood Bar module and also the hardware control module which is absent in the reference view we can say that clementine is conformant with the reference architecture but cannot be termed as fully conformant.

Conclusion: