Advantages of the Current System

The given system is based upon event driven architecture. There is a global application context which contains the following -

- Event buses
- Event listeners
- Services

Whenever an event is created, for example, a new directory is created, then an event corresponding to this event is created. This is then posted to one of the event buses present in AppContext.

The event buses are like queues which contain the event objects to be consumed by the corresponding listeners.

The objects of events are consumed by listeners which keep on listening on various event buses. The listeners use some services which are also present in the AppContext class.

Transition System

There are 4 main subsystems - User Management, Library Management, Last.fm Integration, Administrator Features.

User Management

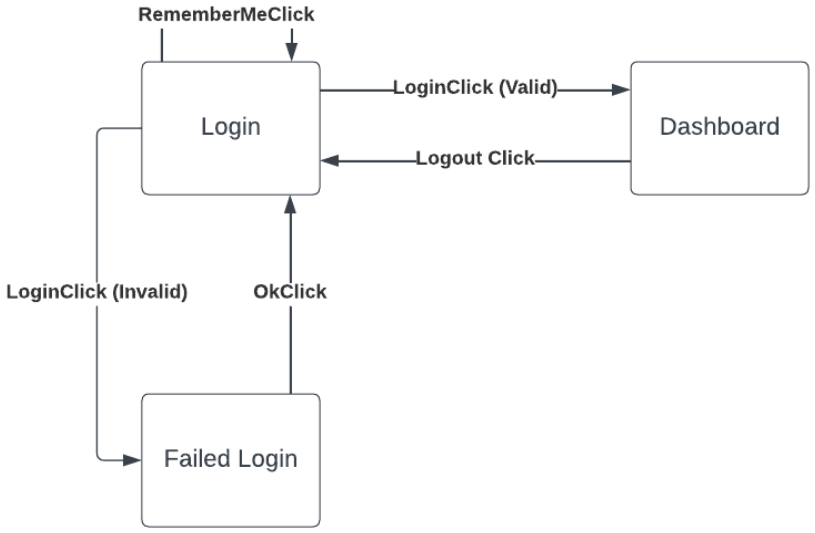
Several users can use the same Music server for managing their libraries. Music thus has a user management system, with users needing to login to add music to the library.

For the six tuple {X, X^0, U, f, y, h}

X = { Login, Dashboard, Failed Login }
X^0 = { Login }
U = { RememberMeClick, LoginClick, LogoutClick, OkClick }
Y = { Login Page, Dashboard Page, Failed Login Popup }

f(Login, LoginClick) = Dashboard (for valid credentials)
f(Login, LoginClick) = Failed Login (for invalid credentials)
f(Failed Login, OkClick) = Login
f(Dashboard, LogoutClick) = Login
f(Login, RememberMeClick) = Login

h(Login) = Login Page
h(Dashboard) = Dashboard Page
h(Failed Login) = Failed Login Popup



Library Management

Users can add songs to Music with a two-step process. First, new music is imported in one of two ways - uploading local files, or downloading files from external sources. Once the file is imported, they can be added to the library after adding metadata information. Users can also edit existing songs.

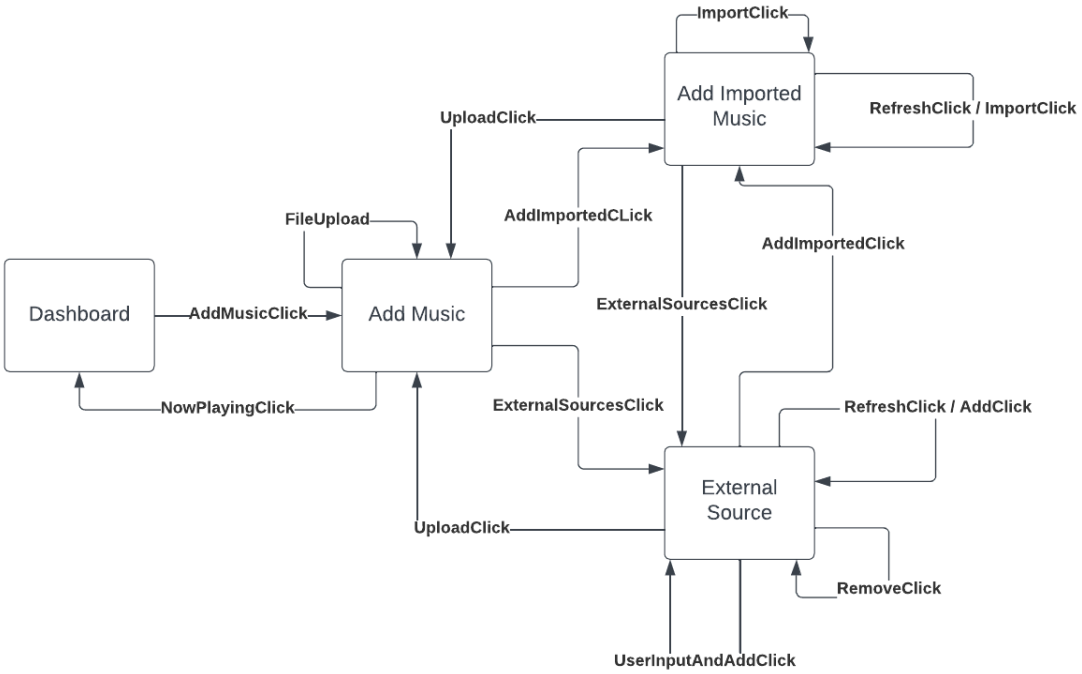
The library management system is complex and so it has been divided into further 2 subsystems - Add Music Subsystem, Manage Playlist Subsystem

Add Music Subsystem

X = { Dashboard, AddMusic, AddImportedMusic, ExternalSource }
X^0 = { Dashboard }
U = { AddMusicClick, NowPlayingClick, FileUpload, UploadClick, AddImportedClick, ImportClick, RefreshClick, ImportClick, ExternalSourceClick, AddClick, RemoveClick, UploadClick}
Y = { Dashboard Page, Add Music Page, Add Imported Page, External Source Page }

f(Dashboard, AddMusicClick) = AddMusic
f(AddMusic, NowPlaylingClick) = Dashboard
f(AddMusic, FileUpload) = AddMusic
f(AddMusic, AddImportedClick) = AddImportedMusic
f(AddMusic, ExternalSourcesClick) = ExternalSources
f(AddImportedMusic, UploadClick) = AddMusic
f(AddImportedMusic, ImportClick) = AddImportedMusic
f(AddImportedMusic, ExternalSourceClick) = ExternalSource
f(AddImportedMusic, RefreshClick) = AddImportedMusic
f(AddImportedMusic, ImportClick) = AddImportedMusic
f(ExternalSources, AddImportedClick) = Add Imported Music
f(ExternalSources, RefreshClick) = ExternalSources
f(ExternalSources, AddClick) = ExternalSources
f(ExternalSources, RemoveClick) = ExternalSources
f(ExternalSources, UploadClick) = AddMusic

h(Dashboard) = Dashboard Page
h(AddMusic) = Add Music Page
h(AddImportedMusic) = My Imported Page
h(External Source) = External Source Page



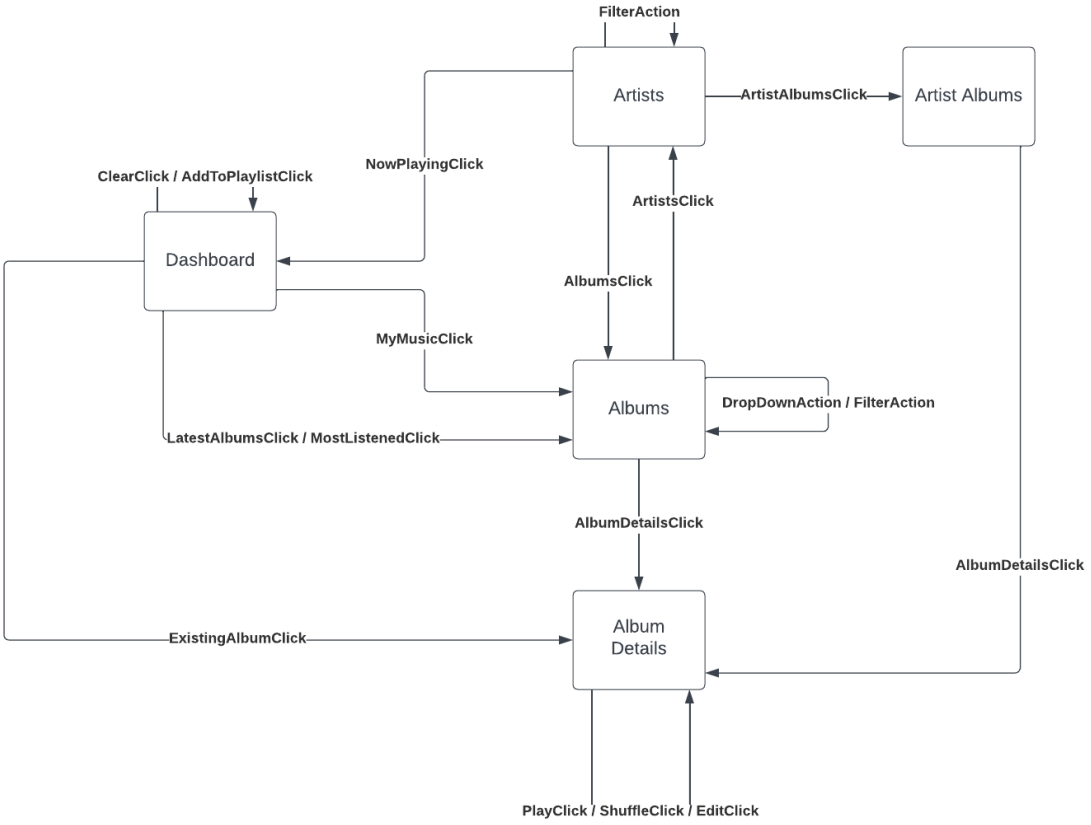
Manage Playlist Subsystem

X = { Dashboard, Albums, Artists, Artist Albums, Album Details }
X^0 = { Dashboard }
U = { ClearClick, AddToPlaylistClick, MyMusicClick, ExistingAlbumClick, DropDownAction, FilterAction, ArtistsClick, NowPlayingClick, LatestAlbumsClick, MostListenedClick, AlbumDetailsClick, PlayClick, ShuffleClick, EditClick, AlbumsClick, ArtistAlbumsClick}
Y = { Dashboard Page, Albums Page, Artists Page, Artist Albums Page, Album Details Page}

f(Dashboard, MyMusicClick) = AddMusic
f(Dashboard, LatestAlbumsClick) = Albums
f(Dashboard, MostListenedClick) = Albums
f(Dashboard, ClearClick) = Dashboard
f(Dashboard, AddToPlaylisstClick) = Dashboard
f(Dashboard, ExistingAlbumClick) = AlbumDetails
f(Albums, DropDownAction) = Albums
f(Albums, FilterAction) = Albums
f(Albums, AlbumDetailsClick) = Album Details
f(Albums, ArtistsClick) = Artists
f(Album Details, PlayClick) = Album Details
f(Album Details, ShuffleClick) = Album Details
f(Album Details, EditClick) = Album Details

f(Artists, FilterClick) = Artists
f(Artists, NowPlayingClick) = Dashboard
f(Artists, AlbumsClick) = Albums
f(Artists, ArtistAlbumsClick) = ArtistAlbums
f(Artist Albums, AlbumDetailsClick) = AlbumDetails

h(Dashboard) = Dashboard Page
h(Artists) = Artists Page
h(Albums) = Albums Page
h(Album Details) = Album Details Page
h(Artist Albums) = Artist Albums Page



Note: The Dashboard can be reached from all states using StartPartyClick action and hence has not been mentioned in f or shown in the diagram.

Note: The music can be paused / resumed / stopped from all states and hence has not been mentioned or shown.

Last.fm Integration

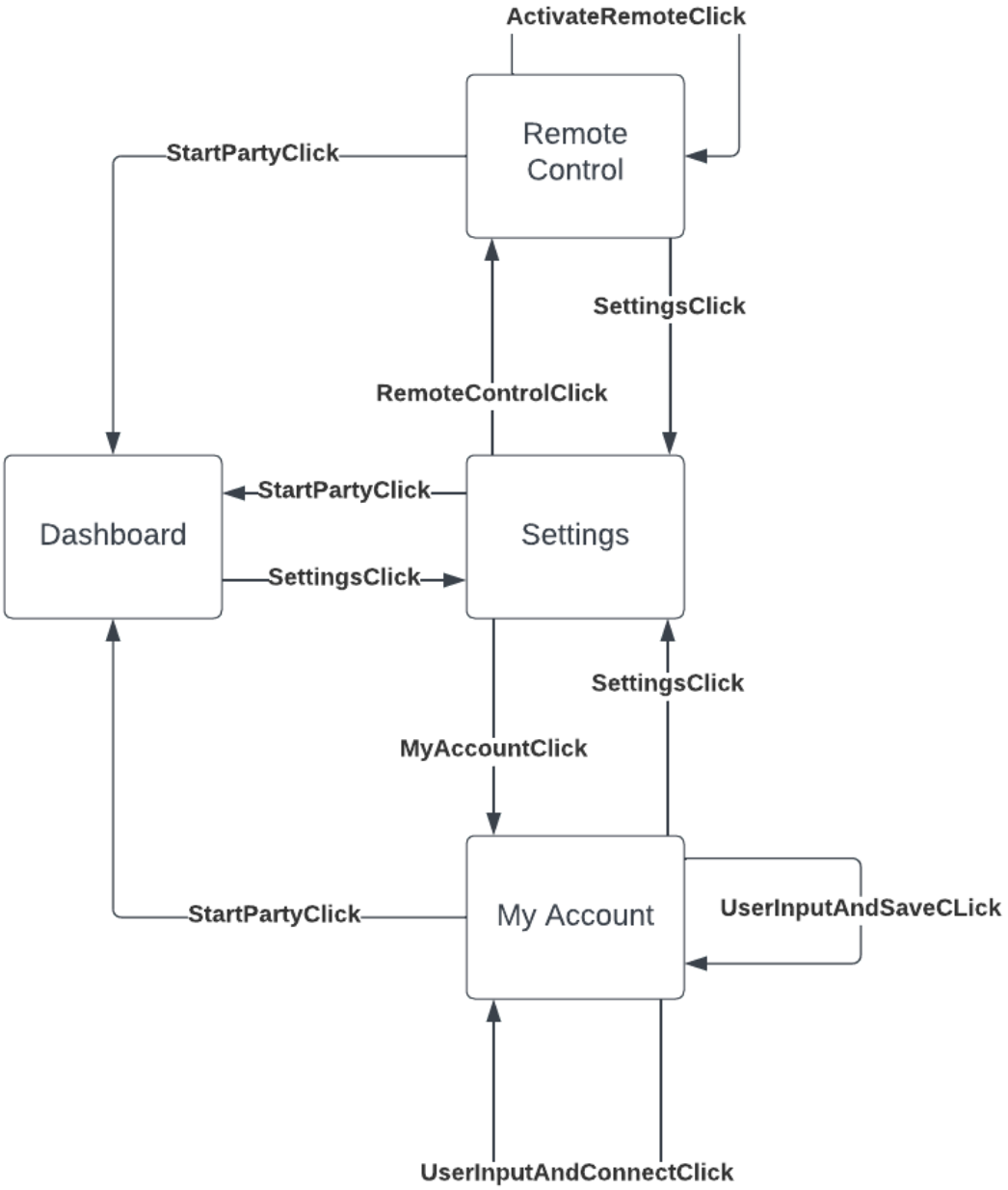
Music allows users to link their Last.fm profile, and sync their listening history with it.

For the six tuple {X, X^0, U, f, y, h}

X = { Dashboard, Settings, My Account, Remote Control }
X^0 = { Dashboard }
U = { SettingsClick, UserInputAndConnectClick, UserInputAndSaveClick, ActivateRemoteClick, MyAccountClick, RemoteControlClick, StartPartyClick }
Y = { My Account Page, Dashboard Page, Settings Dropdown, Remote Control Page }

f(Dashboard, SettingsClick) = Settings
f(Settings, StartPartyClick) = Dashboard
f(Settings, MyAccountClick) = My Account
f(Settings, Remote ControlClick) = Remote Control
f(Remote Control, ActivateRemoteClick) = Remote Control
f(Remote Control, StartPartyClick) = Dashboard
f(Remote Control, SettingsClick) = Settings
f(My Account, UserInputAndConnectClick) = My Account
f(My Account, UserInputAndSaveClick) = My Account
f(My Account, StartPartyClick) = Dashboard
f(My Account, SettingsClick) = Settings

h(Dashboard) = Dashboard Page
h(Settings) = Settings Dropdown
h(My Account) = My Account Page
h(Remote Control) = Remote Control Page


ActivateRemoteClick
Remote Control
StartPartyClick
SettingsClick
RemoteControlClick
Dashboard
StartPartyClick
Settings
SettingsClick
SettingsClick
MyAccountClick
My Account
UserInputAndSaveCLick
StartPartyClick
UserInputAndConnectClick

Administrator Features

The administrator has further privileges. They can create and delete user accounts and change the local directory to which the music is stored, or add new directories.

For the six tuple {X, X^0, U, f, y, h}

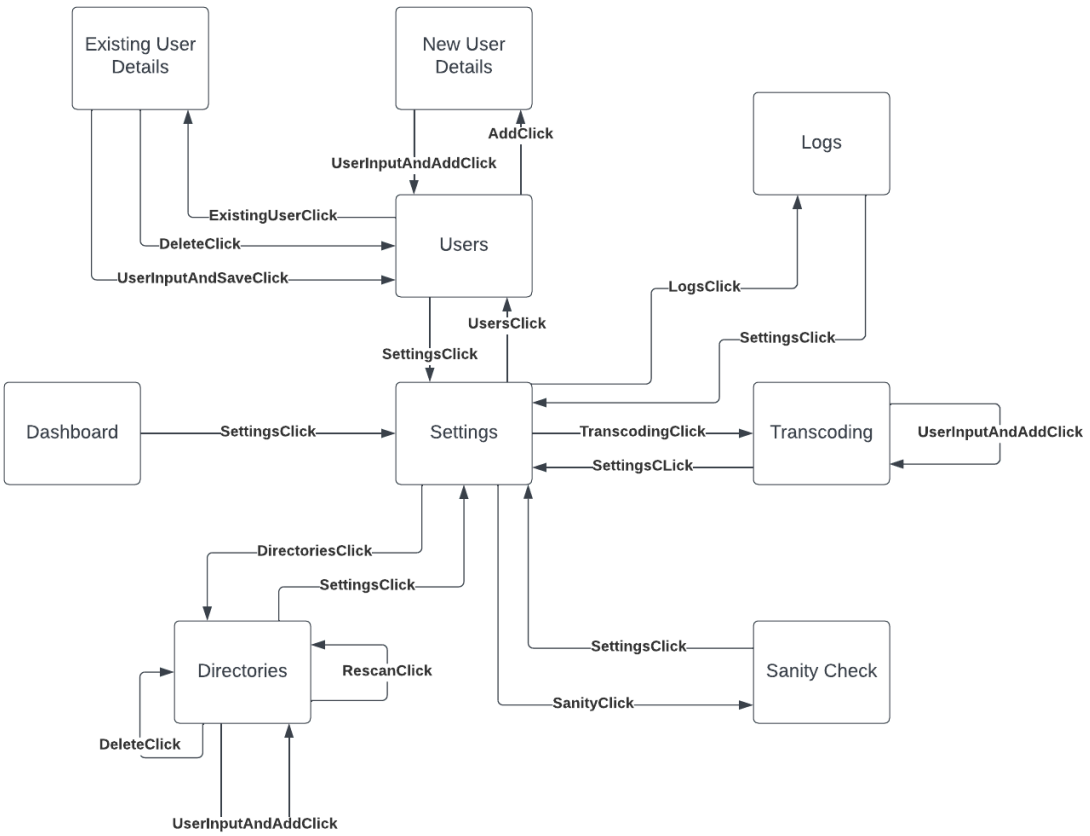
X = { Dashboard, Settings, Directories, Users, New User Details, Existing User Details Sanity Check, Transcoding, Logs }
X^0 = { Dashboard }
U = { SettingsClick, DirectoriesClick, UserInputAndAddClick, UserInputAndSaveClick, RescanClick, DeleteClick, UsersClick, AddClick, ExistingUserClick, TranscodingClick, SanityClick, LogsClick, StartPartyClick }
Y = { Dashboard Page, Settings Dropdown, Directories Page, Users Page, New User Page, Existing User Page, Sanity Check Page, Transcoding Page, Logs Page}

f(Dashboard, SettingsClick) = Settings
f(Settings, DirectoriesClick) = Directories
f(Settings, UsersClick) = Users
f(Settings, TranscodingClick) = Transcoding
f(Settings, SanityClick) = Sanity Check
f(Settings, LogsClick) = Logs
f(Directories, DeleteClick) = Directories
f(Directories, UserInputAndAddClick) = Directories
f(Directories, RescanClick) = Directories
f(Directories, SettingsClick) = Settings
f(Users, AddClick) = New User Details
f(New User Details, UserInputAndAddClick) = Users
f(Users, ExistingUserClick) = Existing User Details
f(Existing User Details, DeleteClick) = Users
f(Existing User Details, UserInputAndSaveClick) = Users
f(Users, SettingsClick) = Settings
f(Transcoding, SettingsClick) = Settings
f(Transcoding, UserInputAndAddClick) = Transcoding
f(Logs, SettingsClick) = Settings
f(Sanity Check, SettingsClick) = Settings

h(Dashboard) = Dashboard Page
h(Settings) = Settings Dropdown
h(Directories) = Directories Page
h(Users) = Users Page
h(New User Details) = New User Page
h(Existing User Details) = Existing User Page
h(Sanity Check) = Sanity Check Page
h(Transcoding) = Transcoding Page

h(Logs) = Logs Page

Note: The Dashboard can be reached from all states using StartPartyClick action and hence has not been mentioned in f or shown in the diagram.



Automatic Refactoring

Automatic refactoring is the process of using software tools to modify source code in a structured and consistent way, without changing its behavior. The goal of automatic refactoring is to make code easier to read and understand, while reducing the risk of introducing errors.

The process of automatic refactoring generally involves the following steps:

1. Analyzing the code: The code to be refactored is analyzed to determine its structure, dependencies, and patterns.
2. Identifying refactoring opportunities: Based on the analysis, the tool looks for opportunities to improve the code, such as identifying redundant code, unused variables, or complex control flow.
3. Applying refactorings: Once the refactoring opportunities have been identified, the tool automatically applies the appropriate changes to the code, while ensuring that the behavior of the code remains unchanged. Examples of refactorings include renaming variables or methods, extracting common code into functions, and simplifying complex conditionals.
4. Verifying the changes: After the refactorings are applied, the tool checks that the code still behaves correctly, using techniques such as unit testing or static analysis.

Automatic refactoring tools can be integrated into an Integrated Development Environment (IDE) or run as standalone applications. Some IDEs, such as IntelliJ IDEA and Visual Studio, include built-in refactoring tools that make it easy to apply common refactorings with just a few clicks

Here we suggest some ways to identify places where the code can be refactored automatically.

Automatic Function Creation

We look for repeated blocks of code. While doing so we try to ignore the whitespaces in the lines as the blocks may be at different levels of indentation. After identifying the blocks, we can create a single function for them and replace the blocks with that function. This is easy to implement.

Problems Solved

1. This can help in solving the long method code smell as the block of code is now replaced by a single line.
2. Reduces insufficient modularization design smell, as creating more functions can make the code more modular.

This reduces the lines of code metric and is often helpful in projects which require repeated use of some specific functions.

Removing Unused Variables

Unused variables can be identified using various methods.

1. Simulation: We can periodically run the compiler in the background with warnings enabled which will tell us which variables are unused.
2. Live Code Inspection: As soon as a variable is defined, we can mark it as unused. Whenever a variable is used for some operation, we mark it as used.
3. Static Code Analysis: Static analysis is debugging the program without actually running it. IDEs mainly used tools based on these to identify unused variables.

Problems Solved

1. This makes the code less cluttered as unnecessary declarations are removed.
2. Code becomes more readable and easier to understand when you don't have to keep track of a few extra variables.

Removing unused variables mainly help with readability of the code. It may also slightly reduce the code size .

Checking Naming Conventions

Usually organizations follow specific naming conventions like make classes PascalCase and functions/variables camelCase. To make the casing consistent across a project we can iterate through all variables to check if they follow the required naming conventions like camel case, pascal case, etc and rewrite the name if necessary.

Problems Solved

1. This helps maintain consistency across a project or codebase. By following a standard naming convention, you can ensure that your code is easy to read and understand, and that other developers can easily navigate and work with your code.
2. Naming conventions make your code more readable and understandable to others, as well as to your future self.
3. When working on a team, naming conventions can help everyone stay on the same page.

Thus, naming conventions are important because they make your code more readable, maintainable, and error-free, which in turn can save you time and help you create better quality software.

Checking Unused Imports

An unused import refers to an import statement in your code that brings in a module or package that is not actually used in the code. Unused imports don't necessarily cause problems in your code, but they can make your code harder to read, and they can slow down the compilation or loading of your code by adding unnecessary overhead.

To solve this issue we can keep track of the import used for a file and check if any function or variable from that import has been used. Whenever we encounter such a function or variable we mark that particular import as used. At the end of the analysis, we can just remove the unused imports.

Problems Solved

1. Helps reduce compilation time as you don't need to compile extra unused packages.
2. Makes the code easier to read and more understandable.
3. Can help prevent accidental cyclic dependency.

Removing unused imports can slightly reduce the lines of code metric. An IDE or text editor generally includes a code analysis tool which can be used, or you can use a standalone tool like Pylint, Flake8, or Pyflakes. These tools can scan your code and identify unused imports, making it easier to clean up your code and improve its performance.

Inlining

Small functions with only 2-3 lines of code can be automatically identified and declared inline. Inline functions give a slight improvement in speed as compared to regular functions as there is no function call and the local variables are not pushed to a stack. We can also simply remove the function and paste it in place of the function call if it has only been used in a few places.

Problems Solved

1. Inline functions give slight improvement in performance as the function stack is not involved
2. Removing the function helps in readability if it has been declared in a separate file as the programmer now does not have to jump between files.
3. Removing the function also helps in resolving the over modularization design smell to some extent.

This helps in making the code more readable at the cost of a slight increase in lines of code. It's important to note that inlining large functions or functions with complex logic can actually have the opposite effect, leading to decreased performance, increased memory overhead, and reduced readability. Therefore, it's important to use discretion when deciding which functions to inline and to test the performance impact of any changes you make.

Reordering

This involves changing the order of elements in your code, such as functions, methods, or class definitions, to improve readability and organization. For example, all the import statements in a file can be identified and then moved to the top. Utility functions can be moved to a separate file. Classes can be given their own file which are then imported.

Problems Solved

1. By reordering code, you can optimize the execution order of statements and reduce the number of redundant or unnecessary operations, which can lead to faster execution times.
2. Reordering code can also improve the readability of your code by making it easier to understand and follow. For example, grouping related statements together can make it easier to see how they are related and what they are doing.
3. By reordering code, you can isolate specific parts of your program and test them independently, which can make it easier to pinpoint and fix issues.

Reordering code can be a powerful technique for optimizing the performance, readability, and maintainability of your code. However, it's important to use discretion and carefully test any changes you make to ensure that they are improving your code rather than introducing new issues or errors.

Automatic Class Creation

We can have the programmer select a set of variables or functions and give them the option to put them in a class. This would mean taking the select code block out and putting it at the top of the file in a class. This way we can implement OOP concepts such as encapsulation, abstraction, etc.

Problems Solved

1. Encapsulation protects the object's data from unwanted access or modification, which helps to prevent errors and security issues in the code.
2. It promotes code clarity and readability by separating the implementation details of an object from its interface.
3. Solves design smells such as deficient or missing encapsulation.

Such an approach of creating classes can help reduce future technical debts and provide a powerful tool for creating modular, reusable, and extensible code. By using classes, we can model real-world entities, encapsulate data and behavior, organize our code, and promote code reuse and extensibility.

Replacing Hard Coded Numbers

Hard coded numbers are fixed, literal values that are embedded directly into the code. We can automatically replace hard-coded numbers with named constants to improve the readability and maintainability of the code. We can write a script that identifies variables which have hard coded numbers assigned to them and then replace them with named constants or macros.

Problems Solved

1. Increases maintainability as even small changes require modifications to multiple locations in the code when numbers are hard coded.
2. Increases reusability as the code is no longer coupled to specific values.
3. It also decreases the chances of errors and bugs. Hard coded numbers can introduce errors into the code, as it is easy to mistype or misinterpret the intended value.

In summary, hard coding numbers can lead to inflexible, unreadable, error-prone, and difficult-to-maintain code. It is generally considered best practice to use named constants or variables instead of hard-coded numbers to improve code quality, maintainability, and readability.