

APP REPORT – PART 2 TACHIYOMI

Last Update: 2023-11-25

Corrections:

- Create sub-section in the UI/UX section that includes what we like in terms of the UI/UX design
- General Improvements to UI/UX section
- Improve the libraries section including how some of them are implemented in Tachiyomi and how the app uses them.

Link to the repository: <https://github.com/tachiyomiorg/tachiyomi>

1. Identified design and architectural patterns

a. Factory Method

As a brief context, the intent of the Factory Method design pattern is to define an interface for creating an object and let subclasses decide which class it is going to be instantiated. The Application class can't predict the characteristics of the object to instantiate, it only knows when a new object should be created (Gamma et al., 1995).

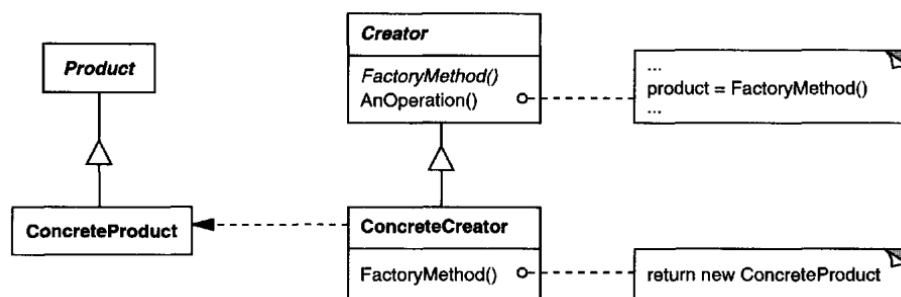
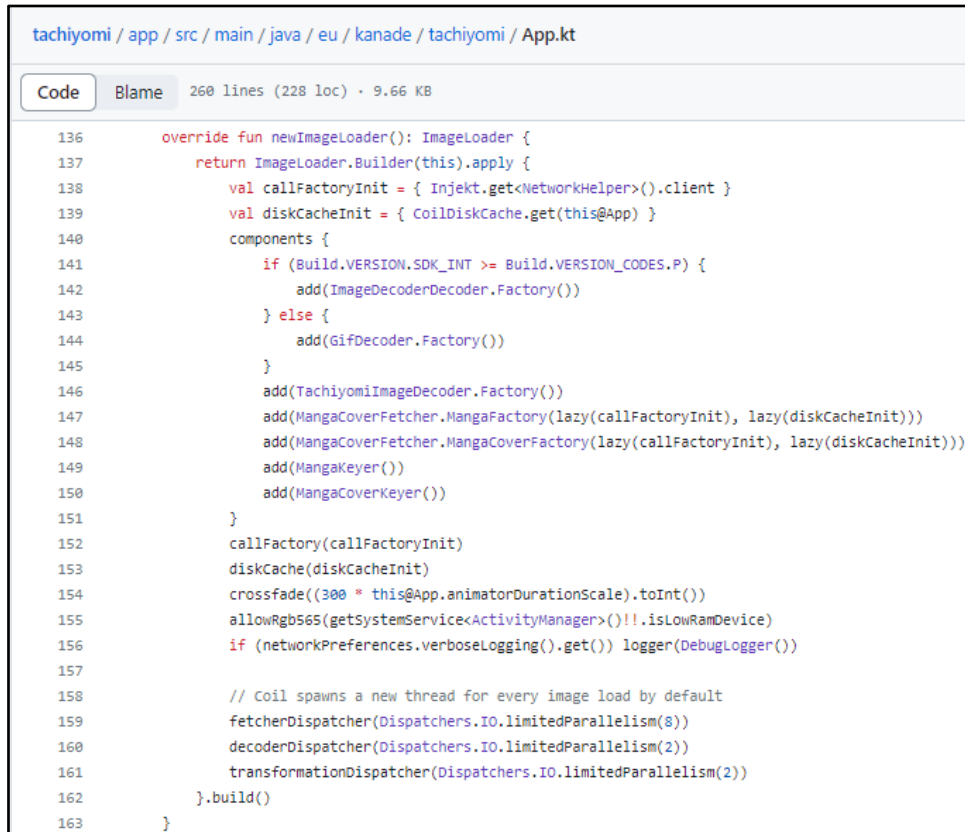


Figure 1. Structure of Factory Method pattern (Gamma et al., 1995)

Product defines the interface of objects the factory method creates. ConcreteProduct oversees the implementation of the Product interface. Creator is the one that declares the factory method, which returns an object of type Product. The Creator class can also define a default implementation of the factory method that returns a default ConcreteProduct object (Gamma et al., 1995).

In Tachiyomi, we can notice the implementation of the factory method pattern by seeing how Manga and MangaCover objects are created in the main App class Kotlin file, specifically in lines 138, 147, 148 and 152.



```
tachiyomi / app / src / main / java / eu / kanade / tachiyomi / App.kt
Code Blame 260 lines (228 loc) · 9.66 KB

136     override fun newImageLoader(): ImageLoader {
137         return ImageLoader.Builder(this).apply {
138             val callFactoryInit = { Inject.get<NetworkHelper>().client }
139             val diskCacheInit = { CoilDiskCache.get(this@App) }
140             components {
141                 if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.P) {
142                     add(ImageDecoderDecoder.Factory())
143                 } else {
144                     add(GifDecoder.Factory())
145                 }
146                 add(TachiyomiImageDecoder.Factory())
147                 add(MangaCoverFetcher.MangaFactory(lazy(callFactoryInit), lazy(diskCacheInit)))
148                 add(MangaCoverFetcher.MangaCoverFactory(lazy(callFactoryInit), lazy(diskCacheInit)))
149                 add(MangaKeyer())
150                 add(MangaCoverKeyer())
151             }
152             callFactory(callFactoryInit)
153             diskCache(diskCacheInit)
154             crossfade((300 * this@App animatorDurationScale).toInt())
155             allowRgb565(getSystemService<ActivityManager>()!!.isLowRamDevice)
156             if (networkPreferences.verboseLogging().get()) logger(DebugLogger())
157
158             // Coil spawns a new thread for every image load by default
159             fetcherDispatcher(Dispatchers.IO.limitedParallelism(8))
160             decoderDispatcher(Dispatchers.IO.limitedParallelism(2))
161             transformationDispatcher(Dispatchers.IO.limitedParallelism(2))
162         }.build()
163     }
```

Figure 2. Implementation of Manga and MangaCover loader in main App file

When we go to the MangaCoverFetcher file, we can see that it is the class file in which the classes MangaFactory and MangaCoverFactory (while implementing a Fetcher interface) define and create the Manga and MangaCover objects. The characteristics of both classes are expressed in each of the constructors.

```

tachiyomi / app / src / main / java / eu / kanade / tachiyomi / data / coil / MangaCoverFetcher.kt
Code Blame 319 lines (287 loc) · 11.5 KB
267     class MangaFactory(
268         private val callFactoryLazy: Lazy<Call.Factory>,
269         private val diskCacheLazy: Lazy<DiskCache>,
270     ) : Fetcher.Factory<Manga> {
271
272         private val coverCache: CoverCache by injectLazy()
273         private val sourceManager: SourceManager by injectLazy()
274
275         override fun create(data: Manga, options: Options, imageLoader: ImageLoader): Fetcher {
276             return MangaCoverFetcher(
277                 url = data.thumbnailUrl,
278                 isLibraryManga = data.favorite,
279                 options = options,
280                 coverFileLazy = lazy { coverCache.getCoverFile(data.thumbnailUrl) },
281                 customCoverFileLazy = lazy { coverCache.getCustomCoverFile(data.id) },
282                 diskCacheKeyLazy = lazy { MangaKeyer().key(data, options) },
283                 sourceLazy = lazy { sourceManager.get(data.source) as? HttpSource },
284                 callFactoryLazy = callFactoryLazy,
285                 diskCacheLazy = diskCacheLazy,
286             )
287         }
288     }

```

Figure 3. MangaFactory class in MangaCoverFetcher file

```

tachiyomi / app / src / main / java / eu / kanade / tachiyomi / data / coil / MangaCoverFetcher.kt
Code Blame 319 lines (287 loc) · 11.5 KB
290     class MangaCoverFactory(
291         private val callFactoryLazy: Lazy<Call.Factory>,
292         private val diskCacheLazy: Lazy<DiskCache>,
293     ) : Fetcher.Factory<MangaCover> {
294
295         private val coverCache: CoverCache by injectLazy()
296         private val sourceManager: SourceManager by injectLazy()
297
298         override fun create(data: MangaCover, options: Options, imageLoader: ImageLoader): Fetcher {
299             return MangaCoverFetcher(
300                 url = data.url,
301                 isLibraryManga = data.isMangaFavorite,
302                 options = options,
303                 coverFileLazy = lazy { coverCache.getCoverFile(data.url) },
304                 customCoverFileLazy = lazy { coverCache.getCustomCoverFile(data.mangaId) },
305                 diskCacheKeyLazy = lazy { MangaCoverKeyer().key(data, options) },
306                 sourceLazy = lazy { sourceManager.get(data.sourceId) as? HttpSource },
307                 callFactoryLazy = callFactoryLazy,
308                 diskCacheLazy = diskCacheLazy,
309             )
310         }
311     }

```

Figure 4. MangaCoverFactory class in MangaCoverFetcher file

b. Data Mapper

According to Fowler & Rice (2003), in the Data Mapper pattern, the general procedure is to put a layer of “Mappers” that move data between objects of the application domain and the database while keeping them independent of each other and the mapper itself.

The Data Mapper layer separates the in-memory objects from the database; hence its responsibility is to transfer data between them and isolate them from each other. Because of

this, the in-memory objects don't need a SQL interface code nor knowledge of the database schema, in fact, the Data Mapper layer itself is even unknown to the domain layer.

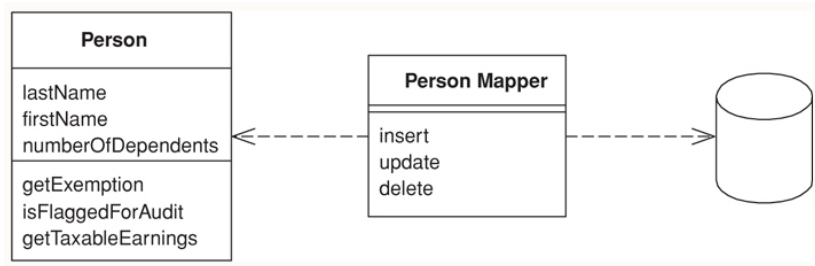


Figure 5. Example of Data Mapper architectural pattern (Fowler & Rice, 2003)

It is possible to see the implementation of this pattern when we take a closer look at how data-related classes are built. Each one of the classes that are defined in the “data” directory has its own Mapper.

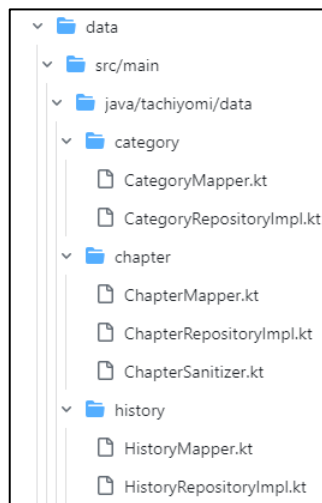


Figure 6. Data directory from repository

Taking the `ChapterMapper` file as an example we can detail how the code is transforming information from the database into objects of the `Chapter` class (which is an abstract representation of a manga chapter). The function `chapterMapper` gets all the information of a “chapter” as parameters and constructs a `Chapter` object that is going to be returned.

```
tachiyomi / data / src / main / java / tachiyomi / data / chapter / ChapterMapper.kt

Code Blame 36 lines (34 loc) · 866 Bytes

3  import tachiyomi.domain.chapter.model.Chapter
4
5  val chapterMapper: (
6      Long,
7      Long,
8      String,
9      String,
10     String?,
11     Boolean,
12     Boolean,
13     Long,
14     Double,
15     Long,
16     Long,
17     Long,
18     Long,
19 ) -> Chapter =
20     { id, mangaId, url, name, scanlator, read, bookmark, lastPageRead, chapterNumber, sourceOrder, dateFetch, dateUpload, lastModifiedAt ->
21         Chapter(
22             id = id,
23             mangaId = mangaId,
24             read = read,
25             bookmark = bookmark,
26             lastPageRead = lastPageRead,
27             dateFetch = dateFetch,
28             sourceOrder = sourceOrder,
29             url = url,
30             name = name,
31             dateUpload = dateUpload,
32             chapterNumber = chapterNumber,
33             scanlator = scanlator,
34             lastModifiedAt = lastModifiedAt,
35         )
36     }
```

Figure 7. Code of ChapterMapper

In the same “data” directory, we can also see the `AndroidDatabaseHandler` file, which appears to be responsible for obtaining the information from the database through queries and defining how these mappings are done within the logic of the application.

```
tachiyomi / data / src / main / java / tachiyomi / data / AndroidDatabaseHandler.kt

Code Blame 104 lines (88 loc) · 3.55 KB

64  override fun <T : Any> subscribeToList(block: Database.() -> Query<T>): Flow<List<T>> {
65      return block(db).asFlow().mapToList(queryDispatcher)
66  }
67
68  override fun <T : Any> subscribeToOne(block: Database.() -> Query<T>): Flow<T> {
69      return block(db).asFlow().mapToOne(queryDispatcher)
70  }
71
72  override fun <T : Any> subscribeToOneOrNull(block: Database.() -> Query<T>): Flow<T?> {
73      return block(db).asFlow().mapToOneOrNull(queryDispatcher)
74  }
```

Figure 8. Mapping handling in `AndroidDatabaseHandler`

c. MVP (Model-View-Presenter)

In the MVC pattern, responsibilities are separated across three components of the system: the model, the view and the presenter. The model is responsible for implementing business logic, business behaviors and state management. In the second place, the view is responsible for rendering the UI elements of the app/system. Finally, the presenter is the one that interacts with the view and model, separating them from each other (Baeldung, 2022).

The presenter triggers the business logic and enables the view to update. It also receives data that is sent from the model and shows it through the view. In this way, testing the correct functioning of the presenter is much easier.

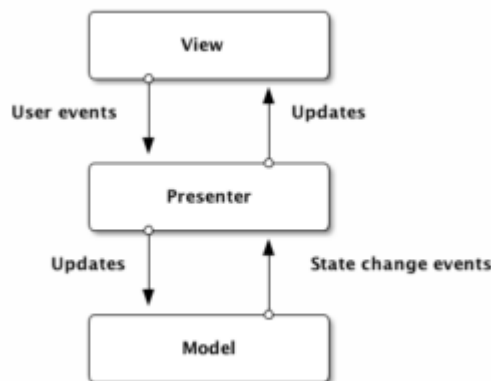


Figure 9. Diagram of MVP pattern (Baeldung, 2022)

We can see the implementation of this pattern in the Tachiyomi app by seeing the “presentation” directory. As its name says, this directory contains the classes of the app that work as presenters. Taking the example of showing the manga library to the user, we’ll detail the LibraryList class file. First, some components of the UI (i.e., the view) are imported:

```
8     import androidx.compose.ui.Modifier
9     import androidx.compose.ui.unit.dp
10    import androidx.compose.ui.util.fastAny
11    import eu.kanade.tachiyomi.ui.library.LibraryItem
```

Figure 10. UI imports in LibraryList file

Some components of the model are also imported, these are the LibraryManga and MangaCover classes:

```
12    import tachiyomi.domain.library.model.LibraryManga
13    import tachiyomi.domain.manga.model.MangaCover
```

Figure 11. Model imports in LibraryList file

Finally, we can see how, by getting the list of mangas from the model using a query, the presenter uses elements of the view to subsequently show the information to the user.

```

tachiyomi / app / src / main / java / eu / kanade / presentation / library / components / LibraryList.kt
Code Blame 75 lines (72 loc) · 2.81 KB
17 @Composable
18 internal fun LibraryList(
19     items: List<LibraryItem>,
20     contentPadding: PaddingValues,
21     selection: List<LibraryManga>,
22     onClick: (LibraryManga) -> Unit,
23     onLongClick: (LibraryManga) -> Unit,
24     onClickContinueReading: ((LibraryManga) -> Unit)?,
25     searchQuery: String?,
26     onGlobalSearchClicked: () -> Unit,
27 ) {
28     FastScrollLazyColumn(
29         modifier = Modifier.fillMaxSize(),
30         contentPadding = contentPadding + PaddingValues(vertical = 8.dp),
31     ) {
32         item {
33             if (!searchQuery.isNullOrEmpty()) {
34                 GlobalSearchItem(
35                     modifier = Modifier.fillMaxWidth(),
36                     searchQuery = searchQuery,
37                     onClick = onGlobalSearchClicked,
38                 )
39             }
40         }
41
42         items(
43             items = items,
44             contentType = { "library_list_item" },
45         ) { libraryItem ->
46             val manga = libraryItem.libraryManga.manga

```

Figure 12. Code of the LibraryList file

This is not the only example that is present in the app, in fact, the presentation directory implements these characteristics for other elements of the logic, like the history, the manga categories or some of the manga's information.

2. Use your knowledge in UI/UX to describe and analyze the design of your app. Describe all the basic components of the UI/UX of the selected app (color, fonts, design metaphor, etc.), answer the following questions: what do you think can be improved? What did you like? For the groups of 3, analyze in detail at least 3 views.

Design System:

A design system serves as the foundation for achieving consistency, user-friendliness, and aesthetic excellence in products. It encompasses a wide range of principles, resources, and elements, which aid design and development teams in preserving brand unity and simplifying the product creation process.

Visual Identity:

- **Color Palette:** Tachiyomi's color palette is a blend of three distinct colors: Tang blue, Office green, and Raisin black. Tang blue adds a sense of vibrancy and energy,

creating an engaging and dynamic visual experience. Office green brings a touch of freshness and harmony, making the app feel welcoming and balanced. Raising black provides a solid and sophisticated foundation, adding depth and contrast to the overall design. Together, these colors contribute to Tachiyomi's unique and visually appealing user interface, enhancing the reading experience for manga enthusiasts.

Primario			#0057CE
	#003A8A		#358AFF
	#0049AC		#579EFF
	#0057CE		#79B2FF
	#0065F0		#9BC5FF
	#1377FF		#BDD9FF

Figure 9. Tang Blue Color – Monocromatic Palette

Primario			#006E17
	#004C10		#00F633
	#006E17		#19FF49
	#00901E		#3BFF64
	#00B225		#5DFF7F
	#00D42C		#7FFF9A

Figure 10. Office Green Color – Monocromatic Palette

Secundario			#1B1D24
	#1B1D24		#747E97
	#313541		#9199AD
	#474D5E		#AFB4C2
	#5D657B		#CCCFD8
	#646D85		#E9EAEE

Figure 11. Raisin Black Color – Monocromatic Palette

- Typography:**
 - Heebo is a sans-serif font with a clean and contemporary design. Its legibility and versatility make it an excellent choice for digital interfaces, including apps like Tachiyomi. Heebo's balanced proportions and rounded letterforms contribute to a pleasant and accessible reading experience.
 - Tachiyomi with Heebo Font: When Tachiyomi is configured to use the Heebo font, it brings a polished and streamlined look to the app. The text in manga titles, descriptions, and navigation elements becomes more readable and aesthetically pleasing. Heebo's crisp and well-defined characters ensure that readers can enjoy their favorite manga titles with clarity and comfort.

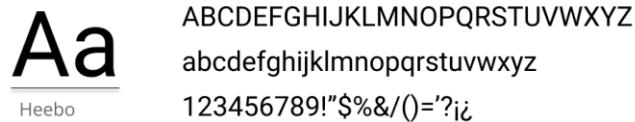


Figure 12. Sans Serif Media (font family) – Heebo (font style)

- **Design Metaphor:**

- **Navigation Structure:** Tachiyomi employs tabs to categorize various app sections, such as "Browse," "Library," and "Downloads." These tabs function similarly to pages in a book, aiding users in moving through the app's content seamlessly.
- **Reading Reminders:** Users can place bookmarks in Tachiyomi to mark specific chapters or pages within a manga. This feature acts as a digital equivalent of inserting a physical bookmark in a book to remember where the reader left off.
- **Visual Selection:** While exploring manga titles in Tachiyomi, users often encounter cover art for each series. This mimics the role of a physical book cover, assisting users in assessing content and making choices based on visual appeal.

- **Views Analysis:**

Tachiyomi's "History", "Browse" and "Settings" views are designed with a clean and minimalistic layout, which is a hallmark of its user interface. This design approach prioritizes simplicity and organization, ensuring that users can easily access and navigate their reading history without distractions or navigate through the different options of the UI in a simple way.

Users often have the flexibility to choose between a list view and a grid view. This feature caters to different user preferences, allowing them to select the visual layout that suits them best.

Each entry typically displays essential information such as the chapter number, the title and incorporates timestamps into each entry. This presentation allows users to quickly identify which chapters they have read and which ones they may want to revisit.

The app employs a high-contrast color scheme, enhancing readability and ensuring that users can enjoy their manga reading experience without visual strain.

Tachiyomi stands out not only for its extensive library and user customization but also for its thoughtful use of icons. These visual cues play a pivotal role in guiding users through the app's multifaceted features and functions.

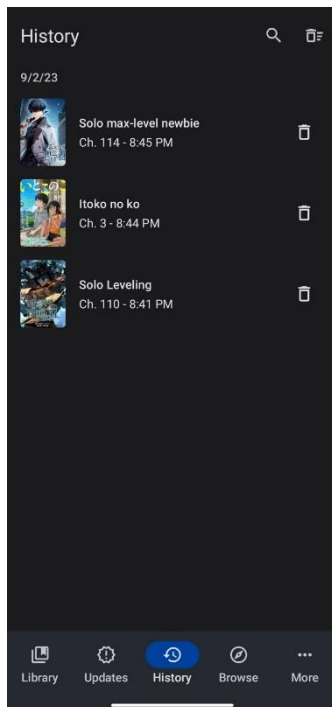


Figure 13. History View

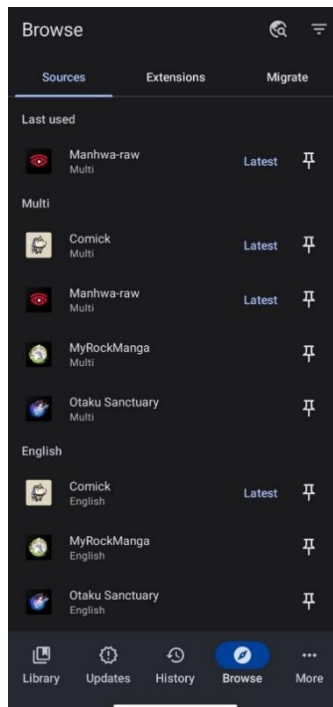


Figure 14. Browse View

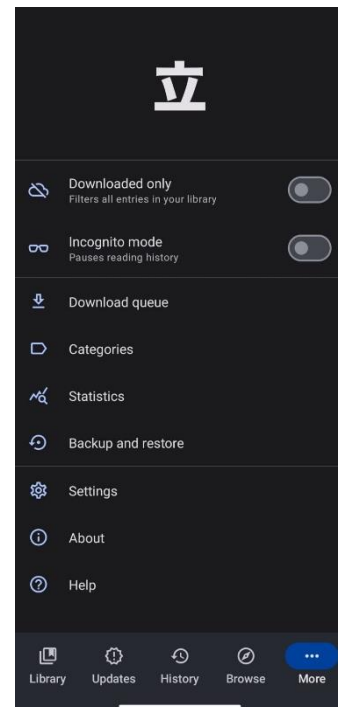


Figure 15. Settings View

- **Things to Improve:**

- **Onboarding Experience:** Improving the initial onboarding experience for new users could help them understand the app's features and functionalities more easily. This could include interactive tutorials or tooltips to guide users through the app's core functions.
- **Enhanced Search:** The search feature is crucial for users to discover new manga. Enhancements in search functionality, such as filters, sorting options, and advanced search criteria, could make it easier for users to find specific manga titles or genres.
- **User Feedback Integration:** Streamlining the process for users to provide feedback or report issues directly within the app can help the development team address bugs and gather valuable insights for future improvements.
- **Personalized Recommendations:** Implementing a recommendation system based on a user's reading history and preferences can help users discover new manga titles that align with their interests.

- **What we like**

- **Customizability:** Tachiyomi allows users to customize the reading experience to their preferences. This includes customizable themes, reading direction, and various view options, making it easier for users to read manga comfortably.
- **User-Friendly Interface:** Tachiyomi's interface is intuitive and easy to navigate, making it convenient for users to discover, manage, and organize their manga collections. The app's user-friendly design enhances the overall reading experience and enables users to access their favorite manga titles quickly.

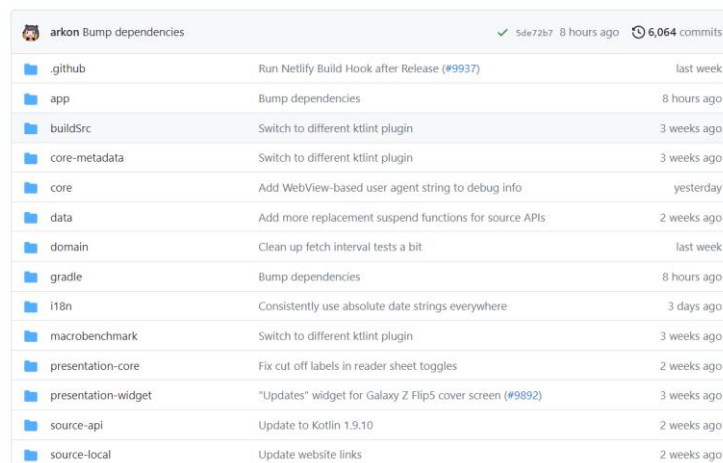
- **Library Support:** Tachiyomi supports multiple manga sources, allowing users to access a vast collection of manga from various online repositories. This extensive library support ensures that users can find and read a diverse range of manga titles within the app.

3. Based on your analysis, what quality attributes is your app managing? How are they doing it? For the groups of 3, describe at least 3 quality attributes.

Maintainability

Tachiyomi is a highly maintainable app. This allows Tachiyomi to be a community app where literally everyone can appport. There are three main factors in the development that are responsible for this quality attribute:

- **Modular Design:** Tachiyomi modularity makes adding new features easy and allows fixing bugs without affecting the rest of the app. In the repository, it's clear which are these compile-independent modules.



arkon Bump dependencies		
.github	Run Netlify Build Hook after Release (#9937)	last week
app	Bump dependencies	8 hours ago
buildSrc	Switch to different ktlint plugin	3 weeks ago
core-metadata	Switch to different ktlint plugin	3 weeks ago
core	Add WebView-based user agent string to debug info	yesterday
data	Add more replacement suspend functions for source APIs	2 weeks ago
domain	Clean up fetch interval tests a bit	last week
gradle	Bump dependencies	8 hours ago
i18n	Consistently use absolute date strings everywhere	3 days ago
macrobenchmark	Switch to different ktlint plugin	3 weeks ago
presentation-core	Fix cut off labels in reader sheet toggles	2 weeks ago
presentation-widget	"Updates" widget for Galaxy Z Flip5 cover screen (#9892)	3 weeks ago
source-api	Update to Kotlin 1.9.10	2 weeks ago
source-local	Update website links	2 weeks ago

Figure 16. Repository of Tachiyomi app

Each main folder (first level in the hierarchy) contains a module. That's why is a *build.gradle* file in each main folder allowing independent compilation.

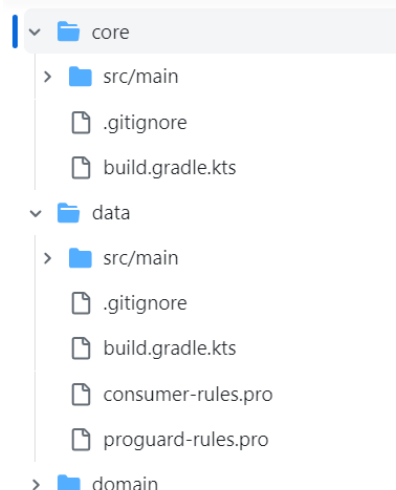


Figure 17. Directories of the repository

- **Extensive documentation:** The extensive documentation makes it easy for new developers to contribute to the project and for existing developers to maintain the code.

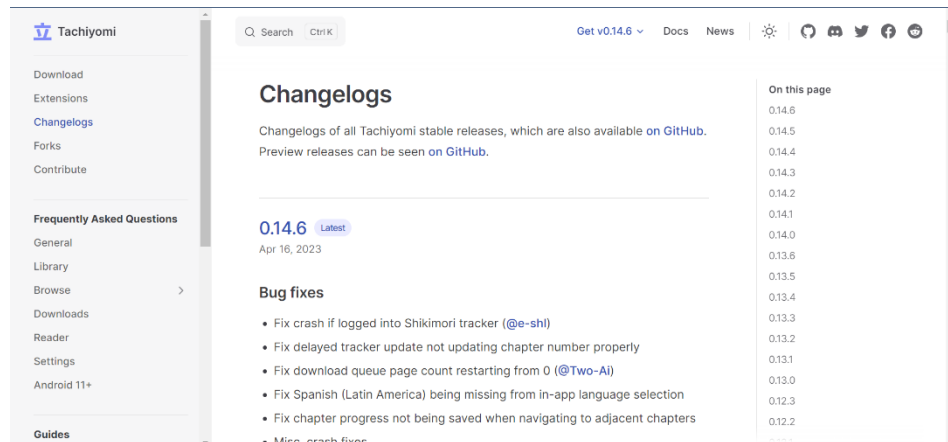


Figure 18. Documentation of Tachiyomi

In the repository documentation, they are very clear in the process of contributing to the repository. The developers must follow strict guidelines and follow several steps for community verification of the new code. For example, they gave examples of how and how not to deal with a bug.

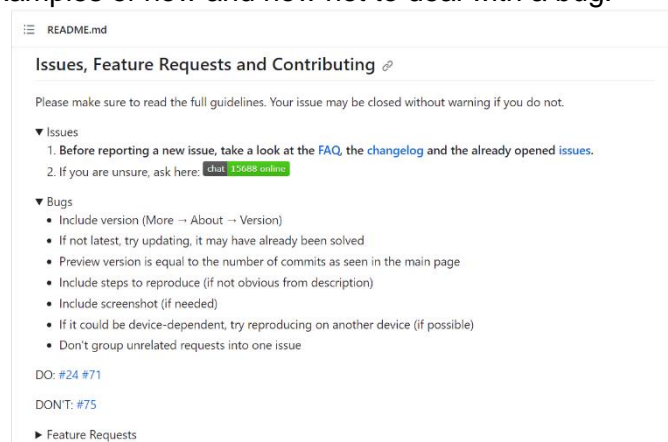


Figure 19. Tachiyomi repository README

- **Community:** The community at Tachiyomi is very proactive. They have a Discord server where if someone finds a bug, they can post it and the developers will be quickly notified allowing for promptly fixing it.

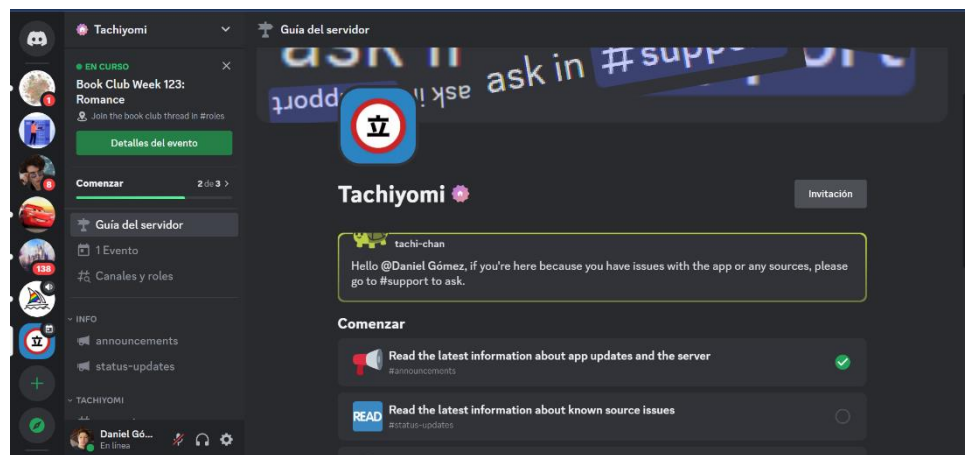


Figure 20. Tachiyomi Discord server

Usability

Tachiyomi is a very user-friendly app, with a simple and intuitive interface. It's quite clear the developer's focus was to make it simple while maintaining it feature-rich.

In the previous part of this App Report, when exploring functionalities, we observed that the reader view is a simple view that maintains a very robust set of configurations. In the (a) figure the main configuration is shown: the reading mode (way to navigate through the manga). Also, in the tab bar at the bottom, the option to crop the image and rotation mode selection. But in the settings menu the user goes from changing behavior when tapping, to adjust color filters for an enhanced reading experience.

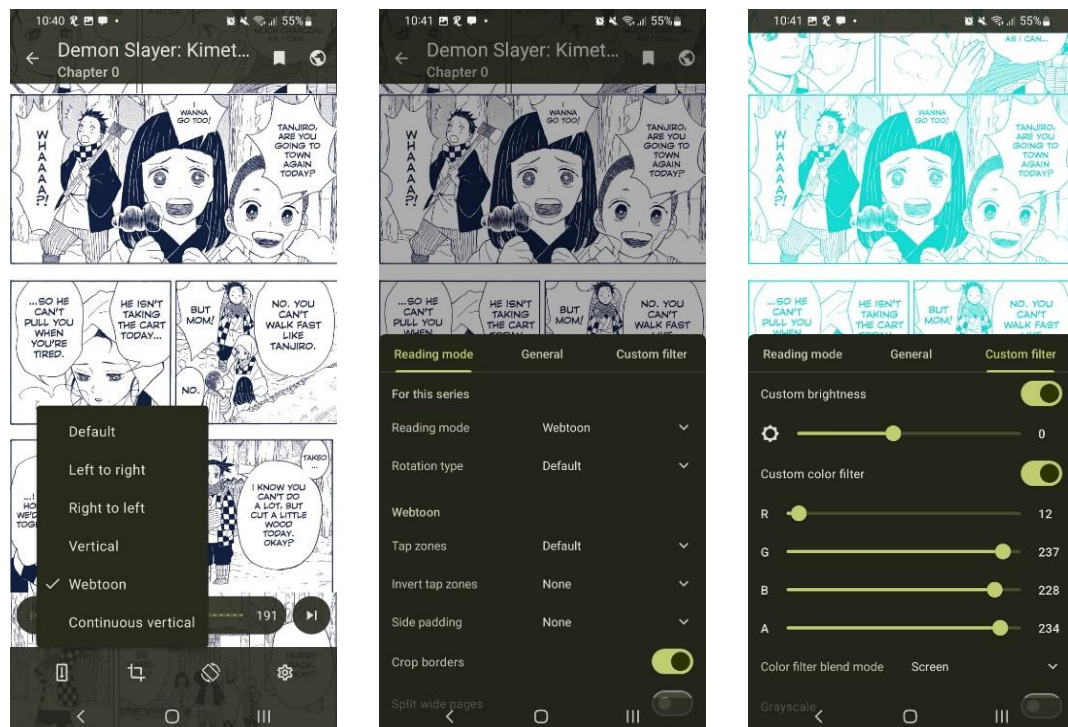


Figure 21. Settings of the reader view

Another feature that goes in direction of improving usability it's the powerful search that allows users to easily find the manga series that they are looking for. The users can search by title, author, genre, and other criteria.

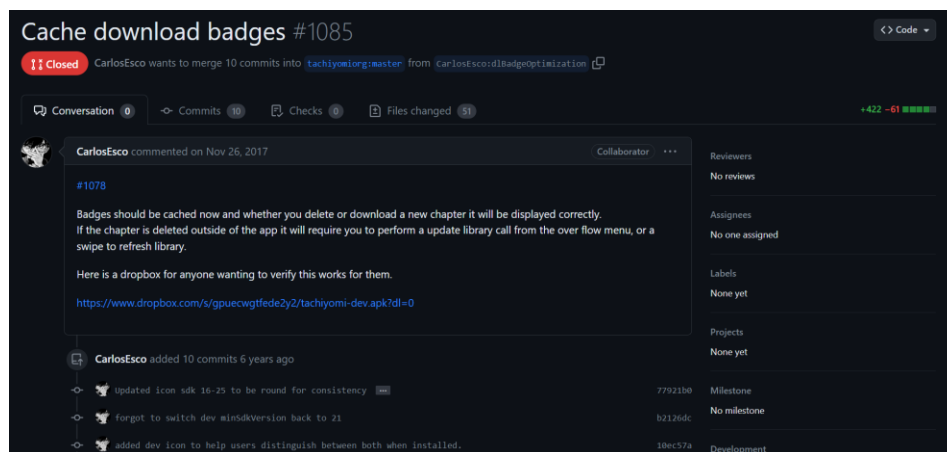
Tachiyomi is also compatible with system dictated accessibility features. For example the components are labeled so if TalkBack is enabled, the user could navigate through voice.

Performance

Tachiyomi is a very performant app, with fast loading times. This attribute is achieved through several optimizations such as caching images and using a lightweight user interface.

Tachiyomi is designed to work with most Android devices, starting with Android 6.0. So, to make it work for all these devices they need to be very careful with the use of system resources, because in a big portion of the Android devices the optimization is critical. This well-implement performance improvement are delivered by the active development community that constantly improves the application.

For example, some years ago users noticed a slow performance after reading 3-5 chapters (<https://github.com/tachiyomiorg/tachiyomi/issues/1063>). That error was due to the memory usage, Tachiyomi was consuming a lot of resources maintaining in memory all the pages already read. A developer 15 days later contributed and solved the issue including image caching (<https://github.com/tachiyomiorg/tachiyomi/pull/1085>).



4. Describe: What libraries and dependencies do your app implement? What are they for?

The dependencies are available on the build.gradle files. As explained in the previous section, Tachiyomi application is divided in multiple modules that are compiled independently.

In the core module all the other modules are called as dependencies:

```
dependencies {
    implementation(project(":i18n"))
    implementation(project(":core"))
    implementation(project(":core-metadata"))
    implementation(project(":source-api"))
    implementation(project(":source-local"))
    implementation(project(":data"))
    implementation(project(":domain"))
    implementation(project(":presentation-core"))
    implementation(project(":presentation-widget"))
}
```

We merged all dependencies and grouped by their functionalities or packages:

- I18n: Library added to support internalization in the application. That means having different languages on the same app.

That is noticeable in <https://github.com/tachiyomiorg/tachiyomi/tree/d4dfa9a2c2a6e627256e99efb08e150a6d234964/i18n/src/commonMain/resources/MR> when all the languages supported by Tachiyomi are there. For example in the sub-folder es we found:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="name">Nombre</string>
    <string name="label_settings">Ajustes</string>
    <string name="label_download_queue">Cola de
descargas</string>
    <string name="label_library">Biblioteca</string>
    <string name="label_recent_manga">Historial</string>
    <string
name="label_recent_updates">Actualizaciones</string>
...
</resources>
```

In most of the view files, they import the i18n strings (import tachiyomi.core.i18n.stringResource) to use it in different languages without recreating a view for each language.

- Compose Libraries: These libraries support the generation of the UI using a declarative style. Allows to use Material components and Animations as well.

```
implementation(platform(compose.bom))
    implementation(compose.activity)
    implementation(compose.foundation)
    implementation(compose.material3.core)
    implementation(compose.material.core)
    implementation(compose.material.icons)
```

```

implementation(compose.animation)
implementation(compose.animation.graphics)
debugImplementation(compose.ui.tooling)
implementation(compose.ui.tooling.preview)
implementation(compose.ui.util)
implementation(compose.accompanist.webview)
implementation(compose.accompanist.permissions)
implementation(compose.accompanist.themeadapter)
implementation(compose.accompanist.systemuicontroller)

```

Tachiyomi generate the UI in a declarative way, so uses the Kotlin Composer for all view. For example in `app/src/main/java/eu/kanade/presentation/browse/GlobalSearchScreen.kt` they use the next snippet of code to generate the UI when searching in the whole application.

```

@Composable
fun GlobalSearchScreen(
    state: SearchScreenModel.State,
    navigateUp: () -> Unit,
    onChangeSearchQuery: (String?) -> Unit,
    onSearch: (String) -> Unit,
    onChangeSearchFilter: (SourceFilter) -> Unit,
    onToggleResults: () -> Unit,
    getManga: @Composable (Manga) -> State<Manga>,
    onClickSource: (CatalogueSource) -> Unit,
    onClickItem: (Manga) -> Unit,
    onLongClickItem: (Manga) -> Unit,
) {
    Scaffold(
        topBar = { scrollBehavior ->
            GlobalSearchToolbar(
                searchQuery = state.searchQuery,
                progress = state.progress,
                total = state.total,
                navigateUp = navigateUp,
                onChangeSearchQuery =
onChangeSearchQuery,
                onSearch = onSearch,
                sourceFilter = state.sourceFilter,
                onChangeSearchFilter =
onChangeSearchFilter,
                onlyShowHasResults =
state.onlyShowHasResults,
                onToggleResults = onToggleResults,
                scrollBehavior = scrollBehavior,
            )
        },
    ) { paddingValues ->
        GlobalSearchContent(
            items = state.filteredItems,

```



```

        contentPadding = paddingValues,
        getManga = getManga,
        onClickSource = onClickSource,
        onClickItem = onClickItem,
        onLongClickItem = onLongClickItem,
    )
}
}

```

- **AndroidX Libraries:** These are system-related libraries that allow to compile the UI and implement features related to the phone sources.

```

implementation(androidx.annotation)
    implementation(androidx.appcompat)
    implementation(androidx.biometric)
    implementation(androidx.constraintlayout)
    implementation(androidx.core)
    implementation(androidx.splashscreen)
    implementation(androidx.recyclerview)
    implementation(androidx.viewpager)
    implementation(androidx.profileinstaller)

```

- **Job Scheduling:** These are libraries to schedule start of the asynchronous process.

```

implementation(androidx.work.workmanager)

```

Some tasks require a big workload so the task is scheduled. For example, when restoring backups, in the next snippet of code, it can be seen how Tachiyomi uses the Work Manager to initiate a background process.

- **Networking:** To handle internet connections with added features.

```

implementation(libs.bundles.okhttp)
    implementation(libs.okio)
    implementation(libs.conscrypt.android) // TLS 1.3
support for Android < 10

```

- **Data Serialization:** Save data in JSON, XML and other formats.

```

implementation(kotlinx.bundles.serialization)

```

- **Disk related:** Libraries for LRU Cache, handle compressed files, etc.

```

implementation(libs.disklru)
implementation(libs.unifile)
implementation(libs.junrar)

```

Tachiyomi use LRU cache when caching chapter app/src/main/java/eu/kanade/tachiyomi/data/cache/ChapterCache.kt. Here we can observe the initialization of the object.

```
private val diskCache = DiskLruCache.open(
    File(context.cacheDir, "chapter_disk_cache"),
    PARAMETER_APP_VERSION,
    PARAMETER_VALUE_COUNT,
    PARAMETER_CACHE_SIZE,
)
```

After that, they use it in different methods for saving and getting pages. For example fun getPageListFromCache(chapter: Chapter): List<Page> or fun putPageListToCache(chapter: Chapter, pages: List<Page>)

- Image Loading: Allows caching images and handle different image files.

```
implementation(platform(libs.coil.bom))
implementation(libs.bundles.coil)
implementation(libs.subsamplingscaleimageview) {
    exclude(module = "image-decoder")
}
implementation(libs.image.decoder)
```

For example when loading the images from the pages (app/src/main/java/eu/kanade/tachiyomi/ui/reader/viewer/ReaderPageImageView.kt). They set specific policies for saving in disk and in memory.

```
val request = ImageRequest.Builder(context)
    .data(data)
    .memoryCachePolicy(CachePolicy.DISABLED)
    .diskCachePolicy(CachePolicy.DISABLED)
    .target(
        onSuccess = { result ->
            setImageDrawable(result)
            (result as? Animatable)?.start()
            isVisible = true
        }
    )

this@ReaderPageImageView.onImageLoaded()
    },
    onError = {

this@ReaderPageImageView.onImageLoadError()
    },
)
    .crossfade(false)
    .build()
context.imageLoader.enqueue(request)
```

- Logging: Library native of the system that allows labeled logs and is optimized for ADB (android debugger bridge).

```
implementation(libs.logcat)
```

- Crash reports and analytics (telemetry): Sends information related to the crashes and app performances using firebase API. The last dependency allows notifications when the app have exceptions related to memory leaking.

```
implementation(libs.acra.http)
```

```
implementation(libs.bundles.shizuku)
```

```
implementation(libs.leakcanary.plumber)
```

References

- Fowler, M., & Rice, D. (2003). *Patterns of enterprise Application Architecture*. Addison-Wesley Professional.
- Gamma, E., Johnson, R., Helm, R., Johnson, R. E. ., & Vlissides, J. (1995). *Design patterns: Elements of Reusable Object-Oriented Software*. Pearson Deutschland GmbH.
- Baeldung. (2022). Difference between MVC and MVP patterns | Baeldung. Baeldung. <https://www.baeldung.com/mvc-vs-mvp-pattern>