Arquitectura CMI V0.2.4 Alpha

Autor Diego Moreano Merino
Desarrollador, Utoqing App

Figure 1

Guía Oficial de la Arquitectura Modular y Escalable para Proyectos Flutter y Multiplataforma

Introducción

La **Arquitectura CMI** (Capas, Módulos y Contenedores recursivos) es una propuesta moderna diseñada para enfrentar los desafíos de organización, escalabilidad y mantenibilidad en proyectos de software, especialmente en entornos donde el crecimiento acelerado y el trabajo colaborativo son constantes.

A diferencia de arquitecturas tradicionales como MVC o Clean Architecture, CMI establece una estructura jerárquica clara basada en tres pilares fundamentales: **Capas**, **Módulos** y **Contenedores**. Cada elemento posee límites bien definidos y responsabilidades específicas, lo que garantiza un bajo acoplamiento y una alta cohesión en todo el proyecto.

CMI no solo facilita el desarrollo inicial, sino que también prepara la aplicación para su evolución a largo plazo, permitiendo integrar nuevas funcionalidades sin comprometer la estabilidad ni generar desorden.

Esta guía está dirigida a desarrolladores, arquitectos de software y equipos que buscan una solución flexible y robusta para gestionar proyectos Flutter, multiplataforma o cualquier entorno que requiera un control estricto de la estructura.

Al recorrer este documento, descubrirás cómo aplicar CMI para lograr proyectos más ordenados, sostenibles y preparados para el futuro.

¿Por qué CMI?

En el desarrollo de software, es común enfrentarse a problemas como:

- Código desorganizado y difícil de escalar.
- Dificultad para integrar nuevas funcionalidades sin romper lo existente.
- Proyectos donde distintos equipos pisan el trabajo de otros por falta de una estructura clara.
- Mantenimiento costoso debido a la alta dependencia entre componentes.

La Arquitectura CMI surge como respuesta a estos desafíos, ofreciendo:

- Una estructura clara y jerárquica, basada en **Capas**, **Módulos** y **Contenedores**.
- Flexibilidad para adaptarse a cambios de requisitos o tecnologías.
- Escalabilidad, permitiendo que las aplicaciones crezcan de manera ordenada.
- Mantenibilidad, facilitando que nuevos desarrolladores comprendan el proyecto rápidamente.
- Reutilización efectiva de componentes y lógica de negocio.

Comparativa con otras arquitecturas

Table 1. *Comparativa con otras arquitecturas*

Característica	**Clean Architecture**	**MVVM**	**MVC**	**CMI**
Separación de capas	Sí	Sí	Sí	Sí
Modularidad avanzada	Parcial	Limitada	No	Sí
Flexibilidad	Alta	Media	Baja	**Muy Alta**
Reutilización	Media	Baja	Baja	**Alta**
Complejidad inicial	Alta	Baja	Baja	Media
Adaptable a Flutter	Sí	Sí	Sí	**Diseñada para Flutter**

Continued on next page

Table 1. *Comparativa con otras arquitecturas* (Continued)

Escalabilidad	Alta	Limitada	Limitada	**Muy Alta**

¿Cuándo usar CMI?

CMI es ideal para:

- Proyectos **multiplataforma** con Flutter donde se prevé un crecimiento constante.
- Equipos de desarrollo que buscan una estructura clara para trabajar de forma colaborativa.
- Aplicaciones que requieren una **alta mantenibilidad**, como productos a largo plazo o SaaS.
- Aplicaciones donde se desea una gestión eficiente de dependencias y lógica de negocio desacoplada.

Este documento presenta la versión **0.2.4 alpha** de l*a Arquitectura CMI, que está en constante evolución, por lo que se recomienda a los desarrolladores aportar sugerencias o adaptaciones según sus experiencias en distintos entornos tecnológicos.*

Filosofía y Principios de la Arquitectura CMI

Filisofia CMI

La Arquitectura CMI nace con una visión clara: ofrecer un marco estructurado que garantice orden, cohesión y escalabilidad sostenible en proyectos de software. A diferencia de enfoques más permisivos, CMI establece normas precisas que previenen el desorden típico que surge en aplicaciones con crecimiento acelerado o en equipos colaborativos.

CMI entiende que un proyecto exitoso no solo debe funcionar, sino que debe ser fácil de mantener, extender y adaptar a nuevas necesidades sin comprometer su estabilidad. Por ello, propone una estructura jerárquica basada en **Capas**, **Módulos** y **Contenedores**, donde cada estructura tiene responsabilidades bien definidas.

Aunque CMI es estricta en su núcleo, permite flexibilidad controlada para adaptarse a casos específicos, siempre bajo principios de claridad y coherencia.

Principios Fundamentales

Single Responsibility Principle (SRP)

La arquitectura CMI aplica el principio de responsabilidad única la cual establece que cada **Capa**, **Módulo** y **Contenedor** en un proyecto debe tener una

única responsabilidad o motivo para cambiar. Esto significa que cada elemento debe hacer una sola cosa y hacerla bien.

Open/Closed Principle (OCP)

La arquitectura CMI aplica el principio de abierto/cerrado, sugiere que las distintos elementos deben estar abiertos para la extensión, pero cerrados para la modificación. Es decir se debe poder agregar una nueva funcionalidad a una parte existente sin cambiar todo su código original.

Liskov Substitution Principle (LSP)

La arquitectura CMI aplica el principio de sustitución de Liskov la cual indica que las clases derivadas o subtipos deben poder ser sustituidos por sus clases base o tipos sin alterar el correcto funcionamiento de la aplicación. Esto significa que un objeto de una clase hija debe ser intercambiable con un objeto de la clase padre sin que el usuario del objeto necesite saber la diferencia.

Interface Segregation Principle (ISP)

La arquitectura CMI aplica este principio que sugiere que es mejor tener múltiples interfaces específicas y pequeñas en lugar de una única interfaz general. Las clases no deberían verse obligadas a depender de métodos que no utilizan. Al dividir las interfaces en grupos más pequeños, se garantiza que los usuarios solo tengan que conocer y usar las funcionalidades que realmente necesitan, reduciendo el acoplamiento y mejorando la claridad del código.

Dependency Inversion Principle (DIP)

La arquitectura CMI aplica el principio de inversión de dependencias lo que establece que los elementos de alto nivel no deben depender de los elementos de bajo nivel. Ambos deben depender de abstracciones. Además, las abstracciones no deben depender de los detalles, sino que los detalles deben depender de las abstracciones. Esto se implementa en la capa **Core** utilizando clases abstractas en el modulo **Rule**, lo que permite que las dependencias sean invertidas y desacopladas, facilitando la inyección de dependencias y pruebas unitarias.

Principio de Separación de Intereses (Separation of Concerns)

La arquitectura CMI aplica el principio que implica dividir un proyecto en distintas **Capas**, **Módulos** y **Contenedores**, donde cada una aborda una preocupación específica o una funcionalidad concreta. La separación de intereses mejora la modularidad del proyecto, permitiendo que los desarrolladores trabajen en partes individuales del proyecto sin necesidad de comprender toda la aplicación en su totalidad. Esto también facilita la identificación y corrección de errores, ya que los problemas suelen estar localizados en un solo lugar.

Principio DRY (Don't Repeat Yourself)

La arquitectura CMI aplica el principio DRY, que busca evitar la duplicación de código o lógica. Cada parte de conocimiento o funcionalidad debe tener una representación única y sin duplicados en el proyecto. Siguiendo este principio, se reduce la cantidad de código redundante, lo que facilita la modificación y el mantenimiento. Si una pieza de lógica necesita ser cambiada, solo hay que hacerlo en un lugar, minimizando el riesgo de inconsistencias y errores.

Principio KISS (Keep It Simple, Stupid)

La arquitectura CMI aplica el principio que enfatiza la importancia de mantener el diseño y la implementación de la aplicación lo más simple y directa posible. La simplicidad reduce la complejidad y facilita la comprensión, el mantenimiento y la modificación del código. Evita agregar funcionalidades complejas de forma innecesaria que puedan complicar el desarrollo o dificultar el entendimiento.

Encapsulamiento

La arquitectura CMI aplica el principio de encapsulamiento para ocultar los detalles internos de un objeto y exponer solo lo necesario a través de una interfaz pública. Esto protege la integridad del estado interno del objeto y reduce el acoplamiento entre diferentes partes de la aplicación. Al encapsular los datos y las operaciones, se permite un mayor control sobre cómo las partes interactúan entre sí, lo que mejora la modularidad y la seguridad de la aplicación.

Alta Cohesión y Bajo Acoplamiento

La arquitectura CMI aplica el principio de alta cohesión que se refiere a la medida en que los elementos dentro de una **Capa**, **Módulo** o **Contenedor** están relacionados y trabajan juntos para cumplir una función específica.

Una **Capa**, **Módulo** o **Contenedor** con alta cohesión es más fácil de entender, mantener y reutilizar. Por otro lado, el bajo acoplamiento indica que los diferentes niveles tienen pocas o nulas dependencias entre sí, lo que permite que se modifiquen o evolucionen de manera independiente, reduciendo el impacto de los cambios en la aplicación.

Principio de Inversión de Control (IoC)

La arquitectura CMI aplica el principio de inversión de control, el cual sugiere que el flujo de control en una aplicación debe ser manejado por un gestor de estados o inyección de dependencias, en lugar de la interfaz de usuario lo haga directamente. Esto significa que en lugar de que la interfaz de usuario de la aplicación invoque directamente a las dependencias, el control es entregado a al modulo **Logic** compartido o al contenedor **Logic** de cada estructura especifica de la capa UI que inyecta

las dependencias necesarias en tiempo de ejecución. Esto promueve la flexibilidad y permite la creación de aplicaciones más modulares.

Modularidad

La arquitectura CMI aplica el principio de modularidad que divide una aplicación en **Capas**, **Módulos** y **Contenedores**, independientes y autónomos, donde cada nivel encapsula una funcionalidad específica y bien definida. Esto facilita el desarrollo, la prueba, el mantenimiento y la reutilización de código.

Conceptos Fundamentales

La Arquitectura CMI se basa en una estructura jerárquica compuesta por **Capas**, **Módulos**, **Contenedores** y **Enrutadores**. Estos elementos son la base de la organización y deben aplicarse de forma estricta para garantizar la coherencia, el orden y la escalabilidad del proyecto.

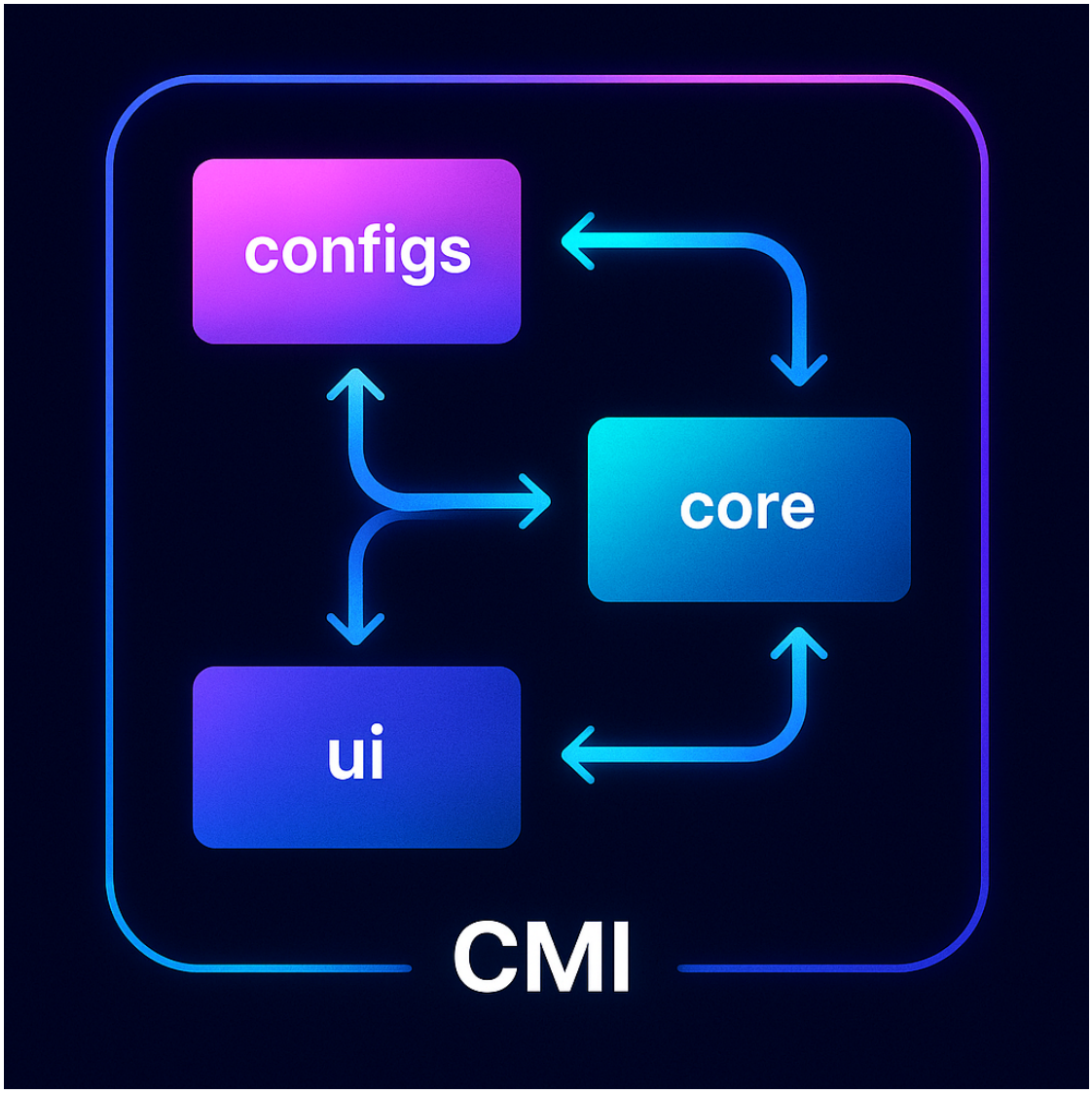

Figure 2. *Capas*

A continuación, se definen cada uno de estos conceptos esenciales.

Capas

Las **Capas** en la Arquitectura CMI representan la división estructural más rígida y fundamental. Esta separación no es arbitraria, sino el resultado de una necesidad clara: **mantener el orden a nivel macro** en cualquier proyecto, sin importar su tamaño o complejidad.

En muchas arquitecturas flexibles, es común que las responsabilidades se mezclen con el tiempo, especialmente bajo presión de crecimiento o cambios rápidos. Este desorden puede empezar de forma imperceptible —una función de lógica dentro de la

interfaz, una configuración incrustada en el núcleo de la aplicación— pero a medida que el proyecto evoluciona, estas pequeñas decisiones generan una estructura difícil de mantener, escalar y entender.

CMI resuelve este problema desde el inicio estableciendo un sistema de capas **fijas e inalterables** que separan claramente:

- La **configuración global (configs)**.
- La **lógica de negocio y datos (core)**.
- La **presentación e interacción con el usuario (ui)**.

¿Por qué CMI impone capas estrictas?

- Para evitar que los desarrolladores "personalicen" la estructura según criterios subjetivos.
- Para garantizar que, sin importar quién trabaje en el proyecto o cuánto crezca, siempre exista una organización reconocible y coherente.
- Para asegurar que cada cambio se realice dentro del contexto correcto, minimizando riesgos de errores estructurales.

Esta rigidez no es una limitación, sino una herramienta para **proteger la salud del proyecto a largo plazo**.

Consecuencias de ignorar la separación de capas

- Dificultad para localizar o modificar funciones específicas.
- Aumento del acoplamiento entre partes de la aplicación.
- Crecimiento desordenado, donde la configuración, la lógica y la interfaz se entrelazan sin control.
- Refactorizaciones costosas en etapas avanzadas del desarrollo.

Módulos

Los **Módulos** en la Arquitectura CMI son la respuesta a uno de los problemas más comunes en el desarrollo de software: el crecimiento descontrolado dentro de una misma área funcional. Aunque las **Capas** establecen una división clara a nivel macro, es dentro de cada **Capa** donde el código tiende a expandirse rápidamente, y sin una estructura adecuada, este crecimiento puede convertirse en caos.

Aquí es donde entran los módulos. Su propósito es **aislar funcionalidades específicas** dentro de cada **Capa**, garantizando que cada bloque de código tenga un espacio definido, evitando concentraciones excesivas de archivos y responsabilidades difusas.

¿Por qué existen los Módulos en CMI?

- Para dividir de manera lógica y ordenada las diferentes áreas funcionales dentro de cada **Capa**.
- Para aplicar de forma efectiva el principio de **separación de responsabilidades**, no solo a nivel de clases o funciones, sino también en la organización del proyecto.
- Para facilitar la escalabilidad, permitiendo que cada funcionalidad crezca de forma independiente sin afectar otras partes de la aplicación.
- Para evitar el llamado "*cajón de sastre*", donde todo termina en una sola carpeta sin criterio.

La diferencia clave con otras arquitecturas

En muchas arquitecturas, la creación de **Módulos** queda a criterio del equipo, lo que puede derivar en estructuras inconsistentes entre proyectos o incluso dentro del mismo equipo de trabajo.

CMI estandariza esta práctica definiendo **Módulos obligatorios** (llamados Módulos estándar) y establece criterios claros para la creación de Módulos adicionales solo cuando es estrictamente necesario.

Errores comunes al gestionar Módulos

- No utilizar **Módulos**, dejando crecer desordenadamente las **Capas**.
- Crear **Módulos** por costumbre sin una necesidad real, generando estructuras vacías o innecesarias.
- Duplicar funcionalidades entre Módulos por falta de claridad en sus responsabilidades.
- Omitir los enrutadores dentro de los Módulos, lo que rompe el control de acceso al código interno.

¿Cuándo crear un módulo adicional?

Aunque CMI permite la creación de módulos adicionales, esta decisión debe ser siempre **justificada**. La creación de un nuevo módulo debe responder a una necesidad clara de aislamiento funcional que no pueda ser resuelta mediante contenedores dentro de un módulo existente.

Crear módulos por "comodidad" o para evitar pensar en la estructura es uno de los errores más costosos a largo plazo.

Contenedores

En la Arquitectura CMI, los **Contenedores** son mucho más que simples carpetas; son una herramienta fundamental para mantener el orden a nivel micro dentro de cada Módulo. Su función es organizar de manera precisa los diferentes tipos de archivos que componen una funcionalidad, asegurando que cada elemento esté ubicado en el lugar correcto según su responsabilidad.

A medida que un proyecto crece, la tendencia natural es que los archivos comiencen a mezclarse si no existe una guía estricta. Los contenedores en CMI evitan este problema, estableciendo un sistema claro para **clasificar y aislar** componentes, vistas, lógica, diálogos, rutas, entre otros.

¿Por qué son esenciales los contenedores en CMI?

- Permiten que cualquier desarrollador pueda identificar rápidamente dónde encontrar o colocar un archivo.
- Refuerzan el principio de **alta cohesión**, agrupando elementos relacionados en un solo lugar.
- Evitan la creación de carpetas arbitrarias basadas en preferencias personales o criterios inconsistentes.
- Preparan la estructura para el crecimiento futuro, incluso si al inicio algunos contenedores están vacíos.
- Facilitan la mantenibilidad y la escalabilidad a largo plazo, especialmente en equipos colaborativos.

Sin contenedores bien definidos, un proyecto puede volverse caótico en poco tiempo, dificultando tanto la comprensión como la extensión de su funcionalidad.

Tipos de Contenedores en CMI

CMI no deja a criterio del desarrollador cómo organizar los archivos. Define contenedores específicos para cada propósito:

- **Contenedores Estándar:** De uso obligatorio, organizan los elementos más comunes como components/, views/, dialogs/, logic/ y router/.
- **Contenedores de Dominio:** Agrupan funcionalidades específicas dentro de Módulos que requieran varios grupos de contenedores, como en el caso del Módulos de layouts.
- **Contenedores Adicionales:** Utilizados solo cuando es necesario organizar elementos que no encajan en los contenedores estándar.
- **Recursividad en Contenedores:** Permite mantener el orden en niveles más profundos cuando la complejidad lo requiere.

Errores comunes al trabajar con contenedores

- Ignorar los contenedores estándar por considerar que "no son necesarios en proyectos pequeños".
- Crear carpetas personalizadas donde ya existe un contenedor estandar.
- Mezclar diferentes tipos de archivos en un mismo contenedor (por ejemplo, colocar vistas dentro de components/).
- Abusar de la recursividad, generando estructuras innecesariamente profundas.
- No utilizar el contenedor router/, comprometiendo el control de rutas internas.

Organización de la Capa UI en CMI: Layouts, Pages, Views y Part

En la Arquitectura CMI, la capa UI está organizada de manera estricta para garantizar modularidad, escalabilidad y claridad.

Para lograrlo, se distinguen claramente varios conceptos dentro de la interfaz gráfica:

- **Layouts**
- **Pages internas**
- **Pages independientes**
- **Views**
- **Part**

Cada uno cumple un rol específico y su correcta organización es fundamental para mantener la cohesión estructural en proyectos de cualquier tamaño.

Layouts

Un **Layout** es una pantalla compuesta por:

- **Vistas estáticas**, como el header, sidebar o footer, que permanecen visibles sin importar el contenido cargado.
- **Vistas dinámicas**, que corresponden a sus **Pages internas**, y cambian según la navegación interna del Layout.

La navegación entre Pages internas está completamente gestionada por el Layout, mediante su propio router interno. Cada Page interna **se considera una vista dinámica del Layout**, pero su composición **es por vistas estáticas**, ya que **no posee navegación interna propia**.

Las Pages internas pueden incluir views/, components/ y otras estructuras, pero **nunca gestionan rutas internas**. Solo el Layout tiene control sobre qué Page interna se muestra en cada momento.

Ubicación en el proyecto:

```
lib/ui/layouts/[layout_name]/
```

***Listing 1**. Ubicación en el proyecto - Layouts*

Ejemplos comunes:

- Un dashboard de administración con sidebar fijo y contenido dinámico.
- Un sitio web donde el header y footer se mantienen al cambiar de sección.

Pages Internas (dentro de un Layout)

Una **Page interna** es una **vista dinámica** que se muestra **dentro del Layout**.

- Son controladas por el Layout.
- Cambian únicamente el contenido dinámico del Layout.
- El resto del Layout (header, sidebar, etc.) permanece fijo.

Cada Page interna puede dividirse en **Views** y **Part** para organizar su contenido.

Ubicación típica:

```
lib/ui/layouts/[layout_name]/views/
```

***Listing 2**. Pages Internas (dentro de un Layout)*

Dentro de views/, cada archivo principal representa una Page interna.

Ejemplos comunes:

- Página de lista de notas en un Layout de usuario.
- Página de perfil de usuario dentro de un Layout general.

Pages Independientes

Una **Page independiente** es una pantalla completa que:

- No pertenece a ningún Layout.
- No comparte navegación o estructura visual persistente.
- Se comporta como una pantalla aislada.

Cada **Page Independiente** es controlada exclusivamente por la **navegación global**, y su estructura es fija durante la ejecución. Está compuesta por **vistas estáticas** (views/), que pueden subdividirse opcionalmente en parts/ si se requiere separación interna.

Las Pages Independientes **no poseen navegación interna**, no controlan el flujo entre rutas ni cargan otras páginas desde su interior.

Ubicación típica:

```
1 lib/ui/pages/[page_name]/
```

***Listing 3**. Pages Independientes*

Ejemplos comunes:

- Página de Error 404.

Views

Una **View** representa la **estructura principal visible** de una Page (interna o independiente).

- Puede ser una pantalla completa o una gran sección funcional.
- Se organizan dentro de views/.
- Son responsables de construir toda la estructura visual base de una página.

Una **View** puede componerse directamente de **View** o de **Part** si su estructura interna es más compleja.

Part

Los part son clases o métodos que forman parte de una clase principal más extensa, y su objetivo es mejorar la legibilidad y organización del código, sin romper su cohesión. En la arquitectura, se utilizan principalmente en **Views**, para dividir visualmente partes complejas como secciones específicas o subcomponentes de la misma vista.

Un **Part** no se utiliza en **componentes reutilizables o globales**, ya que estos deben mantenerse desacoplados y autónomos para ser reutilizados en otros contextos sin depender de la estructura de una vista o del archivo principal.

Casos comunes de uso de part:

- Subdividir una **View** larga en part views como HeaderPartContact, FormPartContact, ListPartContact.
- Separar lógica visual específica de un **Overlay**, **Dialog** o cualquier vista modal que no amerite un archivo completo aparte.
- Nunca se utiliza en componentes reutilizables (Widget, Logic, Controller, etc.) ya que estos tienen propósito y ciclo de vida distinto.

Subdivisión de vistas superpuestas:

Los componentes como *Overlays* y *Dialogs* también pueden dividirse en partes cuando su contenido es extenso o heterogéneo. Aunque no se consideran una **View principal** como una página o pantalla completa, pueden estructurarse siguiendo los mismos principios: cuando su lógica o UI crece, es válido separarlos en **P*art*** para mantener claridad y cohesión.

Por ejemplo, un Overlay que contiene tanto un formulario como información descriptiva puede dividirse en dos partes: _FormularioOverlayPart y _InformacionOverlayPart, ambas privadas dentro del mismo archivo o, si se justifica, en archivos separados, respetando el nivel de jerarquia.

Ubicación típica:

```
lib/ui/layouts/[layout_name]/views/parts/
lib/ui/pages/[page_name]/views/parts/
```

Listing 4. *Part*

Relación Jerárquica entre Layouts, Pages, Views y Part

```
Layout
    Page
          View
                Part
```

```
          Overlay/Dialog (controlado externamente)
                Part (opcional, para dividir contenido)
```

Listing 5. *Relación Jerárquica entre Layouts, Pages, Views y Part*

o, para una Page Independiente:

```
[Page Independiente (View)]
     [Part]
```

Listing 6. *Relación Jerárquica - Page Independiente*

Resumen Conceptual

Table 2. *Resumen Conceptual - Relación Jerárquica entre Layouts, Pages, Views y Part*

Concepto	Descripción	Ubicación típica
Layout	Estructura global fija con navegación interna.	lib/ui/layouts/[layout]/
Page Interna	Página dinámica dentro del Layout.	lib/ui/layouts/[layout]/views/
Page Independiente	Página completa sin Layout.	lib/ui/pages/[page]/
View	Representación visual principal de una Page.	views/
Part	Fragmento funcional de una View, overlay, dialog, component y etc similares a una view.	Dentro de la misma view o de forma excepcional en archivos dentro de un container de la view o similares.

Enrutadores

En la Arquitectura CMI, los **enrutadores** son uno de los elementos más distintivos y esenciales para mantener el control, la cohesión y la integridad de la estructura del proyecto.

A diferencia de otras arquitecturas donde los archivos se importan libremente desde cualquier ubicación, CMI establece que **toda interacción entre capas, módulos o contenedores debe realizarse exclusivamente a través de sus enrutadores**, los cuales definen de forma precisa qué elementos pueden ser accesibles desde el exterior y cuáles deben permanecer encapsulados.

Un **enrutador no es simplemente un archivo de exportación**, sino un **mecanismo estructural normativo** que permite garantizar el bajo acoplamiento entre estructuras y controlar la exposición externa de los elementos internos de cada contexto (como services, logic, components o views). En ciertos casos especiales —particularmente en **contenedores de dominio de la capa ui/**, como los utilizados en **layouts** o **pages independientes**—, un enrutador puede además **actuar como componente visual principal del contenedor de dominio**, al representar directamente una pantalla raíz (por ejemplo, authentication_layout.dart o landing_page.dart). Este patrón, conocido como **enrutador compuesto**, permite encapsular y centralizar la estructura visual y funcional de una pantalla autocontenida, siempre que se cumplan las reglas establecidas para su uso (ver sección "Enrutadores Compuestos").

Este comportamiento especial no reemplaza la estructura jerárquica de CMI; los enrutadores compuestos **no actúan directamente sobre módulos o capas**, sino exclusivamente dentro de **contenedores de dominio específicos de la capa ui/** que agrupan funcionalidades visuales de una pantalla concreta.

¿Por qué son fundamentales los enrutadores en CMI?

- **Control de acceso**: Evitan que los desarrolladores accedan directamente a archivos internos, protegiendo la estructura de dependencias desordenadas.

- **Encapsulamiento real**: Permiten exponer solo lo necesario, manteniendo ocultos los detalles de implementación que no deben ser utilizados fuera de su contexto.

- **Facilitan el mantenimiento**: Si la estructura interna de una capa, módulo o contenedor cambia, mientras el enrutador mantenga su contrato, el resto de la aplicación no se verá afectada.

- **Claridad en las dependencias**: Cuando un archivo importa desde un enrutador, queda claro que está accediendo a una interfaz pública controlada, no al "interior" de la aplicación.

- **Estandarización en la exposición de pantallas raíz**: En el caso de **enrutadores compuestos** ubicados dentro de contenedores de dominio de la capa ui/ (como *_page.dart o *_layout.dart), permiten encapsular no solo las dependencias internas, sino también la estructura visual principal de una pantalla autocontenida, ofreciendo un punto de acceso unificado para la vista y su contexto.

Tipos de Enrutadores en CMI

CMI establece el uso de enrutadores en distintos niveles:

- ser el **único punto de acceso autorizado** para su contexto.

- **Enrutadores Normales:** Son los enrutadores utilizados en cualquier nivel estructural de CMI (capas, módulos o contenedores). Controlan qué elementos son accesibles externamente y cuáles permanecen encapsulados, actuando como punto de acceso único a su contexto.

- **Enrutadores Compuestos:** Son enrutadores ubicados dentro de contenedores de dominio (o adicionales que lo necesite) de la capa ui/ que, además de su función de control de acceso y exportación, contienen directamente la vista raíz de una pantalla (Widget) autocontenida (por ejemplo, archivos *_page.dart o *_layout.dart que incluyen su propio build() y subcomponentes internos).

Cada enrutador, normal o compuesto, mantiene la responsabilidad de ser **el único punto de acceso autorizado a su contexto**, garantizando la cohesión, el encapsulamiento y la claridad en las dependencias.

> Las reglas detalladas sobre la estructura y uso de los enrutadores se encuentran en el capítulo Estructuras Estándar de la Arquitectura CMI. (Nota - Sobre reglas de enrutadores)

Errores comunes al gestionar enrutadores

- **Omitir enrutadores donde son obligatorios**: No crear un enrutador en una capa, módulo o contenedor que expone elementos hacia otras estructuras, violando la regla de control de acceso. *(Nota: los contenedores genéricos son la única excepción, ya que no requieren enrutador propio si no que usan el de su padre).*

- **Exposición indiscriminada**: Exportar todos los elementos internos sin control, rompiendo el principio de encapsulamiento y haciendo públicos detalles que deberían permanecer internos.

- **Acceso directo no autorizado**: Importar archivos internos directamente, sin pasar por el enrutador correspondiente, generando acoplamiento indebido.

- **Enrutador desactualizado**: No actualizar el enrutador al agregar, eliminar o modificar elementos expuestos, provocando errores de dependencia o accesos rotos.
- **Confundir rol de enrutador y vista**: Incluir lógica visual o componentes de presentación directamente dentro de un archivo enrutador que solo debería gestionar exportaciones (excepto en los casos permitidos de enrutadores compuestos en contenedores de dominio de ui/).

Relación Jerárquica

1. Capa
2. Módulo
3. Contenedor
    (a) Elementos: servicios, vistas, lógica, componentes

Cada nivel superior **desconoce los detalles internos** del nivel inferior gracias al principio de **abstracción**, comunicándose siempre a través de su enrutador correspondiente.

Ejemplo Visual

```
lib/
 configs/
     configs.dart         # Enrutador de Configs
 core/
     rules/
         services/
             socket_service_rule.dart
     use/
 ui/
     layouts/
         authentication/
             components/
             views/
     shared/
 main.dart
```

***Listing 7**. Ejemplo Visual CMI*

Patrones de Diseño en CMI

La **Arquitectura CMI** no solo define una estructura clara basada en Capas, Módulos y Contenedores, sino que también recomienda el uso de ciertos **Patrones de Diseño** para resolver problemas recurrentes en el desarrollo de software de manera eficiente, mantenible y coherente con sus principios.

Estos patrones ayudan a reforzar conceptos como la **abstracción**, el **bajo acoplamiento**, la **reutilización** y la **flexibilidad**.

Singleton

El patrón **Singleton** asegura que una clase tenga una única instancia a lo largo de toda la aplicación, proporcionando un punto de acceso global controlado. Este patrón es especialmente útil cuando se necesita gestionar configuraciones o servicios que deben ser consistentes en toda la aplicación.

En el contexto de CMI, el Singleton encuentra su lugar ideal en la capa **configs**, donde es fundamental mantener configuraciones globales unificadas, como el tema de la aplicación, la configuración de idiomas o servicios auxiliares como notificaciones.

Implementar este patrón en CMI permite evitar la creación innecesaria de múltiples instancias, optimizando recursos y garantizando un comportamiento consistente.

Ejemplo en CMI:

```
1 class ThemeConfig {
2   static final ThemeConfig _instance = ThemeConfig._internal
     ();
3
4   factory ThemeConfig() => _instance;
5
6   ThemeConfig._internal();
7
8   final ThemeData theme = ThemeData.light();
9 }
```

***Listing 8**. Singleton*

Aquí, cualquier parte de la aplicación puede acceder al tema sin preocuparse por la gestión de instancias.

Factory

El patrón **Factory** permite delegar la creación de objetos o instancias a una clase especializada, ocultando la lógica de decisión sobre qué implementación concreta utilizar. Este patrón es útil cuando se requiere flexibilidad para seleccionar diferentes variantes de un mismo tipo de objeto según el contexto.

En CMI, aunque los **Services** se mantienen como implementaciones concretas dentro de core/use/services, es en los **Consumers** donde realmente **se aplica el patrón Factory**. Los Consumers gestionan la lógica para decidir qué Service usar en función de las condiciones del entorno, configuración o necesidades del usuario.

De esta manera, el Consumer actúa como una "fábrica" que abstrae al resto de la aplicación de conocer qué Service específico está utilizando.

Ejemplo en CMI:

```
class AuthenticationConsumerUse implements
   AuthenticationConsumerRule {
  final AuthenticationServiceRule _service;

  AuthenticationConsumerUse({required bool isLocal})
      : _service = isLocal ? LocalAuthServiceUse() :
   ApiAuthServiceUse();

  @override
  Future<void> login(String user, String pass) {
    return _service.login(user, pass);
  }
}
```

***Listing 9**. Factory*

Así, puedes cambiar la lógica de autenticación con solo modificar un parámetro.

Repository

El patrón Repository actúa como una capa intermedia entre la lógica de negocio y las fuentes de datos, proporcionando una interfaz consistente para acceder a la información, independientemente de su origen. Este patrón es clave para desacoplar el dominio de la aplicación de las particularidades de las bases de datos, servicios web o sistemas de almacenamiento.

En CMI, el Repository se expresa mediante las **abstracciones** que se definen en core/rules/, las cuales actúan como contratos formales de la lógica de negocio. La UI y otras capas superiores **interactúan exclusivamente con estas abstracciones**, sin preocuparse por si los datos provienen de una API REST, una base de datos local o un sistema de caché.

Las **implementaciones** de estos contratos residen en core/use/, donde se aplica la lógica concreta usando servicios, adaptadores u orígenes de datos según corresponda. Esta separación permite cambiar o combinar fuentes de datos sin afectar ni la lógica de negocio definida en rules/ ni la interfaz de usuario. Este enfoque refuerza el principio de inversión de dependencias, uno de los pilares de la arquitectura CMI.

```
abstract class TaskConsumerRule {
  Future<List<Task>> fetchTasks();
}
class TaskConsumerUse implements TaskConsumerRule {
  Future<List<Task>> fetchTasks() async {
  // Code
  }
}
```

***Listing 10**. Repository*

Así, cualquier cambio en la fuente de datos queda encapsulado dentro del repository.

Adapter

El patrón **Adapter** es esencial cuando se necesita que dos interfaces incompatibles trabajen juntas. Su función principal es traducir o transformar los datos o comportamientos de una interfaz externa para que sean compatibles con el sistema interno y viciversa.

En la Arquitectura CMI, los **adapters** de la capa core/use cumplen precisamente esta función, convirtiendo los datos crudos que provienen de los **origins** (como respuestas de APIs) en estructuras limpias y consistentes definidas en los **data rules**.

Esto garantiza que el resto de la aplicación siempre trabaje con datos predecibles y alineados con el dominio del negocio, independientemente de cómo lleguen esos datos desde el exterior.

Ejemplo en CMI:

```
import 'package:project/core/rules/data/task_data_rule.dart
  ';

class TaskAdapterUse {
  /// Convierte de JSON (formato externo) al modelo interno
   (TaskDataRule)
  TaskDataRule fromJson(Map<String, dynamic> json) {
    return TaskDataRule(
      title: json['title'] as String,
      completed: json['done'] as bool,
    );
  }

  /// Convierte del modelo interno (TaskDataRule) a JSON (
   formato externo)
  Map<String, dynamic> toJson(TaskDataRule task) {
```

```
14      return {
15        'title': task.title,
16        'done': task.completed,
17      };
18    }
19  }
```

***Listing 11**. Adapter*

De este modo, se mantiene la integridad del modelo de datos interno.

Dependency Injection (DI)

La **Inyección de Dependencias (DI)** es un patrón que promueve el desacoplamiento al suministrar las dependencias de una clase desde el exterior, en lugar de que la clase misma las cree. Esto facilita la reutilización, el testing y la flexibilidad del sistema.

En CMI, este patrón es fundamental para la comunicación entre la **UI** y la capa **core/use**, donde las vistas o controladores reciben sus dependencias a través de mecanismos de inyección, comúnmente gestionados con herramientas como **Riverpod** o **GetIt** en Flutter.

La DI permite cambiar implementaciones o realizar pruebas unitarias sin modificar el código principal de las vistas o lógica de presentación.

Ejemplo en CMI:

```
final taskConsumerProvider = Provider<TaskConsumerUse>((ref)
    {
  return TaskConsumerUse(service: TaskServiceUse());
});
```

***Listing 12**. Dependency Injection (DI)*

Así, la UI permanece limpia y desacoplada de las instancias concretas.

Observer

El patrón **Observer** permite que múltiples objetos estén "suscritos" a otro objeto y sean notificados automáticamente cuando su estado cambia. Es la base de la programación reactiva, donde la UI responde dinámicamente a los cambios de datos.

En CMI, este patrón se aplica dentro del contenedor **logic/** de la UI, donde se gestiona el estado reactivo utilizando herramientas como ChangeNotifier, ValueNotifier o soluciones más robustas como **Riverpod**.

Gracias a este patrón, la interfaz puede mantenerse sincronizada con los datos sin necesidad de refrescos manuales o lógicas complejas de actualización.

Ejemplo en CMI:

```
class TaskNotifier extends ChangeNotifier {
  List<Task> tasks = [];

  void addTask(Task task) {
    tasks.add(task);
    notifyListeners();
  }
}
```

***Listing 13**. Observer*

Esto permite que cualquier vista que escuche a TaskNotifier se actualice automáticamente.

Strategy

El patrón **Strategy** permite definir múltiples algoritmos o comportamientos intercambiables, que pueden ser seleccionados en tiempo de ejecución según las necesidades del contexto.

En CMI, este patrón se aplica principalmente en los **consumers**, donde puede ser necesario alternar entre diferentes estrategias de procesamiento, como distintos métodos de autenticación, formas de ordenar datos o modos de sincronización.

Este enfoque facilita extender funcionalidades sin modificar el código existente, simplemente añadiendo nuevas estrategias.

Ejemplo en CMI:

```
abstract class AuthStrategy {
  Future<void> authenticate();
}

class GoogleAuthConsumerRule implements AuthStrategy { ... }
class EmailAuthConsumerRule implements AuthStrategy { ... }

class GoogleAuthConsumerUse implements
   GoogleAuthConsumerRule {
  @override
  Future<void> authenticate() {
    // óLgica concreta para Google
  }
}

class EmailAuthConsumerUse implements EmailAuthConsumerRule
   {
  @override
```

```
17   Future<void> authenticate() {
18     // óLgica concreta para Email
19   }
20 }
```

Listing 14. *Strategy*

Consideraciones al Usar Patrones de Diseño en CMI

Si bien los patrones de diseño son herramientas poderosas para resolver problemas recurrentes en el desarrollo de software, su uso indebido puede generar complejidad innecesaria o dificultar el mantenimiento del código. En el contexto de CMI, es importante aplicar estos patrones con criterio y solo cuando realmente aporten valor.

- **Singleton:** Úsalo únicamente cuando sea estrictamente necesario garantizar una única instancia global. Abusar de este patrón puede generar dependencias ocultas y dificultar las pruebas.
- **Strategy:** Aplícalo solo cuando existan múltiples algoritmos o comportamientos intercambiables. No fuerces su uso en casos donde una simple condicional sería suficiente.
- **Factory:** Si bien facilita la creación de objetos, evita sobreingeniería en casos donde una instancia directa es más clara.
- **Adapter:** Úsalo únicamente para resolver incompatibilidades reales entre interfaces. No lo apliques cuando ambas partes ya son compatibles de forma natural.
- **Repository:** Evita duplicar lógica en los Consumers si el Repository no aporta una abstracción real sobre las fuentes de datos.

Aplicar patrones por “moda” o por querer seguir siempre una estructura compleja puede ser más perjudicial que beneficioso. En CMI, la simplicidad controlada es clave.

Resumen de Patrones Recomendados

***Table 3**. Patrones Recomendados*

Patrón	**Capa/Módulo Recomendado**	**Propósito Principal**
Singleton	Configs	Instancia global controlada
Factory	Core/Use	Crear implementaciones flexibles
Repository	Core/Rules - Core/Use	Abstraer acceso a datos
Adapter	Core/Use (Adapters)	Convertir datos externos
Dependency Injection	UI - Core/Use	Gestionar dependencias
Observer	UI (Logic)	Estados reactivos
Strategy	Core/Consumers	Lógica intercambiable

Patrones Internos y Compatibilidad con la Arquitectura CMI

La Arquitectura CMI establece un marco estructural para organizar proyectos de software, definiendo cómo deben dividirse y relacionarse las capas, módulos, contenedores y enrutadores.

Sin embargo, CMI no impone un modelo específico para la interacción interna entre clases, pantallas, controladores o modelos.

En este contexto, los **patrones internos** como **MVC (Model-View-Controller)**, **MVVM (Model-View-ViewModel)**, **MVP (Model-View-Presenter)** y **Clean Architecture** juegan un rol importante, ya que definen **cómo fluye la información dentro de un módulo o contenedor**.

Este capítulo analiza la compatibilidad de CMI con estos patrones internos y cómo se complementan para construir proyectos organizados y sostenibles.

¿Qué son los Patrones Internos?

Los patrones internos son modelos de diseño que regulan **cómo interactúan** diferentes elementos dentro de una funcionalidad o componente de software.

Mientras CMI organiza el proyecto **desde afuera hacia adentro** (capas, módulos, contenedores), los patrones internos organizan la **interacción entre clases o componentes dentro** de un módulo o layout.

Ejemplos comunes de Patrones Internos:

- **MVC (Model-View-Controller):** Separa los datos, la interfaz y la lógica de control.
- **MVVM (Model-View-ViewModel):** Añade una capa de vinculación automática entre vista y modelo.
- **MVP (Model-View-Presenter):** Similar a MVC, pero el Presenter toma un rol más activo.
- **Clean Architecture:** Propone una separación de anillos concéntricos para proteger las reglas de negocio del resto de la aplicación.

Compatibilidad de CMI con Patrones Internos

La Arquitectura CMI es **totalmente compatible** con el uso de patrones internos, siempre que se respeten los principios básicos de:

- **Separación de responsabilidades.**
- **Bajo acoplamiento entre elementos.**
- **Alta cohesión dentro de módulos.**
- **Modularidad clara en la estructura.**

En esencia:

CMI organiza el proyecto estructuralmente; los patrones internos organizan el flujo de datos y la interacción entre componentes. (Nota - Compatibilidad de CMI con Patrones Internos)

Ambos enfoques no se contradicen, sino que **se complementan**.

Compatibilidad específica con Patrones Internos

Table 4. *Compatibilidad específica con Patrones Internos*

Patrón Interno	**Compatibilidad con CMI**	**Notas**
MVC	Totalmente compatible	Organiza Model, View y Controller dentro de contenedores apropiados (components/, views/, logic/).
MVVM	Totalmente compatible	Ideal en frameworks reactivos; ViewModel puede residir en logic/ y coordinar vistas y modelos.
MVP	Compatible	Se puede mapear Presenter en logic/ y mantener alta cohesión.
Clean Architecture	Parcialmente compatible	Algunos principios de Clean Architecture, como reglas de negocio independientes y separación de capas, ya están integrados en CMI. Puede aplicarse internamente en core/ si se requiere mayor control de dominio.

Ejemplo Conceptual de Integración

```
ui/
 layouts/
      authentication/
          components/
          views/
          logic/
```

```
7           router/
```

***Listing 15**. Organización externa (CMI)*

Organización interna (por ejemplo, MVC):

- Model: Puede estar en core/rules/ o core/use/.
- View: Dentro de views/ en el layout.
- Controller: Dentro de logic/ en el layout.

Así, un flujo de login podría ser:

- **LoginView** (evento de login) **LoginController** (valida y usa) **LoginModel**

Todo organizado sin romper la estructura de CMI.

Notas importantes

- CMI **no obliga** a un patrón interno específico. Cada equipo puede elegir el que mejor se adapte a su dominio y experiencia.
- **No todos los proyectos requieren patrones internos complejos.** En proyectos pequeños, una separación simple entre vista y lógica puede ser suficiente.
- Cuando se usan patrones internos, **deben integrarse respetando las reglas de CMI**: No mezclar roles (por ejemplo, un ViewModel no debe actuar como un Service). Mantener la modularidad de contenedores.

Estructuras Estándar de la Arquitectura CMI

La **Arquitectura CMI** establece una organización jerárquica basada en tres niveles principales:

Capa → Módulo → Contenedor

Dentro de esta jerarquía, existen las denominadas **estructuras estándar o dominio**, las cuales son de **uso obligatorio** para garantizar la coherencia, escalabilidad y mantenibilidad de cualquier proyecto. Estas estructuras definen cómo debe organizarse el código desde su base, asegurando que todos los desarrolladores trabajen bajo un mismo esquema.

Sin embargo, CMI también contempla la necesidad de adaptarse a proyectos más complejos o específicos. Por ello, junto a las estructuras estándar y dominio, permite la creación de **módulos** y **contenedores adicionales**, siempre que se respeten los principios fundamentales de la arquitectura.

Cuando se requiere una organización más profunda, es posible extender la jerarquía mediante **contenedores recursivos**, manteniendo el orden sin sacrificar flexibilidad.

Este capítulo presenta las estructuras estándar y dominio obligatorias y explica cómo y cuándo se pueden ampliar mediante estructuras adicionales.

Capas Estándar

Las **capas** son la división principal y más rígida de CMI. Son **fijas**, no pueden ser modificadas, eliminadas ni ampliadas. Toda aplicación bajo CMI debe estructurarse sobre estas tres capas:

Table 5. *Capas Estándar*

Capa	**Descripción**
configs/	Gestión de la configuración global.
core/	Lógica de negocio y datos.
ui/	Interfaz de usuario y presentación.

Estas capas aseguran la separación de responsabilidades a nivel macro, evitando mezclas entre configuración, lógica y presentación.

Reglas:

1. Las capas en CMI están **predefinidas** y deben mantenerse constantes en cualquier implementación, garantizando una estructura coherente y universal.

2. Cada capa ocupa un lugar específico dentro del proyecto, formando parte de la **raíz estructural**. Esto asegura que la organización sea predecible y fácil de navegar.

3. La arquitectura está diseñada para operar con un número limitado de capas, evitando la proliferación innecesaria de divisiones que puedan complicar la gestión del proyecto.

4. La interacción entre capas debe ser siempre **controlada y ordenada**, aplicando los principios de **abstracción**, **bajo acoplamiento** y utilizando mecanismos claros de comunicación, como los enrutadores.

5. Cada capa debe gestionar su comunicación con el exterior a través de un **punto de acceso controlado**, asegurando que solo se exponga lo necesario y manteniendo encapsulada su lógica interna.

6. Toda capa debe cumplir una función definida y contener las subdivisiones necesarias (módulos o estructuras equivalentes) que permitan gestionar sus responsabilidades de manera eficiente.

Módulos

Módulos Estándar

Cada capa contiene módulos que organizan sus responsabilidades internas. Los siguientes módulos son **obligatorios** dentro de sus respectivas capas:

En la Capa core/:

Table 6. *Módulos Estándar - Capa core*

Módulo	**Descripción**
rules/	Encargado de definir las abstracciones de la aplicación . Aquí se responde a la pregunta "*¿Qué debe hacer mi aplicación?*".
use/	Responsable de implementar esas abstracciones. Aquí decides "*¿Cómo lo va a hacer?*".

En la Capa ui/:

Table 7. *Módulos Estándar - Capa ui*

Módulo	**Descripción**

Continued on next page

Table 7. *Módulos Estándar - Capa ui* (Continued)

app/	Punto de entrada de la aplicación.
layouts/	Estructura principal de vistas.
shared/	Elementos reutilizables.
pages/	Pantallas independientes (opcional según el proyecto).

Módulos Adicionales

CMI permite la creación de **módulos adicionales** cuando el proyecto lo requiere, por ejemplo, para gestionar funcionalidades específicas que no encajan en los módulos estándar. Estos módulos deben:

- Mantenerse dentro de la jerarquía **Capa → Módulo**.
- Respetar las normas de organización interna (uso de contenedores y enrutadores).
- Ser coherentes con los principios de separación y claridad.

Reglas:

1. Todo módulo debe tener un **propósito específico**, representando una funcionalidad o dominio claramente identificado dentro de la capa en la que se encuentra.
2. Los módulos deben ser **independientes entre sí**, evitando dependencias innecesarias que puedan generar acoplamiento excesivo.
3. La estructura de un módulo debe facilitar la **reutilización**, permitiendo su expansión o modificación sin afectar negativamente al resto de la aplicación.
4. Cada módulo debe gestionar su comunicación con el exterior a través de un **punto de acceso controlado**, asegurando que solo se exponga lo necesario y manteniendo encapsulada su lógica interna.
5. La creación de módulos debe responder a necesidades reales del sistema, evitando divisiones artificiales o genéricas que no aporten valor a la organización del proyecto.

Contenedores

Contenedores de Dominio

CMI contempla el uso de **contenedores de dominio** como parte de su estructura flexible y escalable.

Los contenedores de dominio permiten organizar el código según áreas funcionales o contextos específicos dentro de un módulo. No tienen nombres predefinidos, ya que dependen directamente de la lógica del proyecto.

Ejemplos de contenedores de dominio:

- authentication/
- dashboard/
- profile/
- settings/
- reports/

Estos contenedores actúan como **carpetas raíz internas**, y dentro de ellos es donde se aplican los **contenedores estándar** obligatorios.

```
ui/
 layouts/
      authentication/
          router/
          components/
          views/
          logic/
```

***Listing 16**. Ejemplos de contenedores de dominio*

Contenedores Estándar

Los contenedores son obligatorios dentro de los módulos o contenedores de dominio donde aplican. Organizan elementos como servicios, lógica, componentes visuales, etc.

En core/rules/:

El módulo rules/ define las **abstracciones** del sistema. Aquí, los contenedores estándar garantizan que las interfaces, la lógica abstracta y los modelos de datos estén correctamente separados.

Table 8. *Contenedores Estándar - Modulo core/rules*

Contenedor	**Propósito**
services/	Declaración de interfaces para servicios externos.
consumers/	Definición abstracta de la lógica de negocio.
data/	Modelos de datos puros, sin dependencias externas.

En core/use/:

El módulo use/ contiene las **implementaciones** concretas de las abstracciones definidas en rules/. Aquí, los contenedores estándar organizan las implementaciones, la adaptación de datos y el manejo de fuentes externas.

Table 9. *Contenedores Estándar - Modulo core/use*

Contenedor	**Propósito**
services/	Implementaciones concretas de los servicios.
adapters/	Transformación de datos externos a modelos internos y viciversa.
origins/	Manejo de datos crudos (APIs, bases de datos, etc.).
consumers/	Implementación de la lógica de negocio abstracta.

En ui/layouts/:

El módulo layouts/ organiza las vistas y paginas dentro de una pantalla. Aquí los contenedores estándar ayudan a mantener la interfaz limpia, reutilizable y desacoplada.

Table 10. *Contenedores Estándar - Modulo ui/layouts/auth(example)*

Contenedor	**Propósito**
views/	Subdivisión del layout
components/	Widgets reutilizables específicos del layout.
logic/	Controladores y estado local del layout.
dialogs/	Modales exclusivos del layout.
pages/	Pestañas del layout (pueden manejar contedores internos como views/ o otros, para subdividir la pagina)

En ui/shared/:

Table 11. *Contenedores Estándar - Modulo ui/shared*

Contenedor	**Propósito**
components/	Widgets reutilizables globales.
logic/	Controladores y estado globales.
dialogs/	Modales globales.

Contenedores adicionales

Los **contenedores adicionales** son unidades de organización creadas dentro de un **contenedor estándar** o un **contenedor de dominio**. Su propósito principal es **extender la organización interna y permitir la recursividad estructural de la arquitectura.**

Permiten dividir la lógica, componentes o recursos dentro de un contenedor superior, manteniendo una estructura jerárquica sin romper los principios de modularidad y encapsulamiento.

Tipos de contenedores adicionales. A partir de su propósito y relación con el contenedor superior, los contenedores adicionales se dividen en **dos tipos:**

Contenedor adicional interno:

- Es un contenedor que **solo organiza elementos internos** dentro del contenedor estándar o de dominio.
- **No continúa la recursividad estructural**: no contendrá nuevos contenedores adicionales dentro.
- **No expone funcionalidades directamente** hacia otros contenedores o capas externas.
- **No requiere su propio enrutador.** Toda exposición externa se sigue gestionando a través del **enrutador del contenedor superior**.

```
1 logic/
2     logic_landing.dart  <-- enrutador de Logic
3     scroll/
4           scroll_logic.dart    <-- contenedor adicional
    interno
```

Listing 17. *Contenedor adicional interno - Ejemplo*

Contenedor adicional expuesto:

- Es un contenedor que **sí continúa la recursividad estructural**: contendrá nuevos contenedores adicionales o subdivisiones internas.
- **Expone funcionalidades directamente** hacia el nivel superior.
- **Requiere su propio enrutador.** Esto asegura una frontera de acceso clara para su estructura interna y sus exposiciones públicas.

```
ui/
 layouts/
      reports/
          router/
          components/
          views/
          exports/   (contenedor adicional expuesto)
              router/
              csv/
              pdf/
              excel/
```

Listing 18. *Contenedor adicional expuesto - Ejemplo*

Reglas:

1. **Todo contenedor adicional** requiere un enrutador si continuará organizando más contenedores adicionales en su interior.
2. **Todo contenedor adicional** requiere un enrutador si necesita exponer directamente funcionalidades o recursos a otros contenedores o capas.
3. Un **contenedor adicional interno** no requiere un enrutador si solo organiza elementos y su acceso sigue pasando por el enrutador del contenedor superior.
4. Un **contenedor adicional expuesto** debe declarar su propio enrutador incluso si depende parcialmente del enrutador del contenedor superior.
5. Un **contenedor adicional** nunca debe exponer directamente sin pasar por un enrutador (propio o del contenedor superior).
6. Un **contenedor adicional interno** solo puede existir si está anidado en un contenedor con enrutador que gestione sus exposiciones.
7. Si un **contenedor adicional interno** necesita crecer en el futuro, deberá migrar su estructura a un contenedor adicional expuesto creando su propio enrutador.

Gestión de la estructura de la aplicación

La **gestión de la estructura** en la Arquitectura CMI no consiste solo en seguir un conjunto de reglas predefinidas, sino en aplicar esas reglas con criterio, entendiendo que cada decisión estructural impacta directamente en la claridad, mantenibilidad y escalabilidad del proyecto.

En el capítulo anterior, se definieron las **estructuras de CMI**: las capas, módulos y contenedores que forman la base inmutable de CMI. Sin embargo, conocer

estas estructuras no es suficiente. La verdadera fortaleza de CMI se demuestra cuando sabes **cómo implementar** esa jerarquía en un proyecto real, cómo expandirla cuando sea necesario y cómo mantener el orden a medida que el sistema crece.

Este capítulo te guiará paso a paso en la aplicación práctica de la estructura CMI, desde sus elementos más básicos hasta las variantes flexibles como módulos adicionales, contenedores de dominio y recursivos. Además, abordaremos buenas prácticas, criterios de expansión y errores comunes que debes evitar para no comprometer la integridad de tu arquitectura.

El objetivo es que no solo sigas un esquema, sino que **comprendas el porqué de cada decisión estructural** y construyas proyectos preparados para evolucionar sin caer en el desorden.

Aplicación Correcta de la Estructura CMI

En CMI, la estructura **no depende del tamaño del proyecto**. Ya sea una aplicación pequeña o un sistema complejo, siempre se debe aplicar la **estructura completa** desde el inicio. Esto se debe a que CMI está diseñada para proyectos que están destinados a crecer, y la mejor forma de garantizar un crecimiento ordenado es establecer una base sólida y coherente desde el primer día.

La Regla Principal: La Forma Siempre es Completa

No importa si hoy tienes solo un servicio o un par de vistas. La jerarquía de CMI —con sus capas, módulos, contenedores de dominio y contenedores estándar— debe estar presente desde el inicio.

Esto evita que en el futuro tengas que realizar reestructuraciones dolorosas cuando el proyecto escale, ya que todo estará listo para soportar nuevas funcionalidades de manera ordenada.

Ejemplo General de Estructura Inicial

A continuación, se presenta cómo debería lucir cualquier proyecto que implemente CMI correctamente desde el inicio, incluso si es un proyecto “pequeño”:

```
lib/
 configs/
      routes/
      theme/
      configs.dart

 core/
      rules/
          services/
          consumers/
          data/
```

```
12      use/
13          services/
14          adapters/
15          origins/
16          consumers/
17      core.dart
18
19  ui/
20      app/
21      layouts/
22         authentication/
23             router/
24             components/
25             views/
26             dialogs/
27             logic/
28      shared/
29          components/
30          dialogs/
31          logic/
32      ui.dart
33
34  main.dart
```

Listing 19. *Ejemplo General de Estructura Inicial*

Jerarquía en Acción

La fortaleza de la Arquitectura CMI radica en su jerarquía bien definida. Cada proyecto, sin importar su complejidad, debe construirse siguiendo este flujo lógico:

> Capa → Módulo → Contenedor de Dominio (En caso Especiales) → Contenedores Estándar → Contenedores Internos (si aplica) (Jerarquía en Acción - Flujo)

A continuación, veremos cómo aplicar cada nivel de esta jerarquía, entendiendo el propósito de cada elemento y su correcta implementación.

Capas Estándar: La Base Inamovible

Toda aplicación CMI comienza con las **tres capas estándar**:

```
1 lib/
2  configs/
```

```
3  core/
4  ui/
```

Listing 20. *Capas Estándar: La Base Inamovible*

Estas capas representan la separación fundamental:

- **configs/**: Centraliza configuraciones globales.
- **core/**: Contiene la lógica de negocio y gestión de datos.
- **ui/**: Gestiona la interfaz y experiencia de usuario.

> Regla: Las capas son fijas, obligatorias y nunca deben ser modificadas, eliminadas o duplicadas. (Capas Estándar - Regla)

Módulos Estándar y Módulos Adicionales

Dentro de cada capa, defines los **módulos estándar** que organizan las responsabilidades principales.

Ejemplo en core/:

```
1 core/
2  rules/
3  use/
```

Listing 21. *Módulos Estándar y Módulos Adicionales - Ejemplo en core/*

Ejemplo en ui/:

```
1 ui/
2  app/
3  layouts/
4  shared/
```

Listing 22. *Módulos Estándar y Módulos Adicionales - Ejemplo en ui/*

Si el dominio del proyecto lo requiere, puedes crear **módulos adicionales**. Por ejemplo, en configs/ podrías añadir un módulo environment/ para gestionar configuraciones de entornos específicos.

> Criterio: Los módulos adicionales deben responder a necesidades claras, nunca ser creados sin propósito definido. (Módulos Estándar y Módulos Adicionales - Regla)

Contenedores de Dominio: Organizando por Funcionalidad

En módulos como layouts/, el siguiente paso es crear **contenedores de dominio** que agrupen elementos relacionados a una funcionalidad específica.

Ejemplo en layouts/:

```
ui/
 layouts/
      authentication/
      dashboard/
```

Listing 23. *Contenedores de Dominio: Organizando por Funcionalidad - Ejemplo en layouts/*

Cada contenedor de dominio establece un contexto funcional, evitando mezclar componentes de distintas áreas de la aplicación.

Contenedores Estándar: La Organización Interna Obligatoria

Dentro de cada contenedor de dominio, se aplican los **contenedores estándar** definidos por CMI.

Ejemplo:

```
ui/
 layouts/
      authentication/
          router/
          components/
          views/
          dialogs/
          logic/
```

Listing 24. *Contenedores Estándar: La Organización Interna Obligatoria - Ejemplo en un layout*

> Regla: Los contenedores estándar son obligatorios donde corresponda, aunque inicialmente tengan poco contenido. (Contenedores Estándar: La Organización Interna Obligatoria - Regla)

Contenedores Internos y Recursividad

Cuando la cantidad de elementos dentro de un contenedor estándar crece, puedes crear **contenedores internos** para mejorar la organización.

Ejemplo:

```
components/
 buttons/
       primary_button.dart
       secondary_button.dart
```

Listing 25. *Contenedores Internos y Recursividad - Ejemplo*

Puedes utilizar contenedores con nombre o genéricos (container/), según el caso.

> Criterio: La recursividad debe aplicarse solo cuando aporte claridad, evitando sobreestructurar innecesariamente. (Contenedores Internos y Recursividad - Criterio)

Enrutadores: Controlando el Acceso

Cada capa, módulo y contenedor en CMI debe contar con su **enrutador**, el cual actúa como único punto de acceso autorizado hacia el exterior de su contexto estructural.

En algunos casos, especialmente en la capa ui/, un enrutador ubicado dentro de un contenedor de dominio también puede actuar como **vista raíz** de una pantalla autocontenida. Este patrón, conocido como **enrutador compuesto**, es común en archivos como *_layout.dart o *_page.dart, donde el enrutador no solo expone elementos internos, sino que además implementa la estructura visual principal de la pantalla.

Ejemplo:

```
core/
 core.dart

ui/
 layouts/
       authentication/
           authentication_layout.dart
```

Listing 26. *Enrutadores: Controlando el Acceso - Ejemplo*

Los enrutadores refuerzan la abstracción y protegen la integridad de la estructura, evitando dependencias directas hacia carpetas internas.

Reglas:

- Todo enrutador (ya sea de capa, módulo o contenedor) es el único punto de acceso autorizado para exponer elementos internos hacia otras partes de la aplicación.

- Los enrutadores garantizan el control de acceso, el encapsulamiento y la comunicación ordenada entre estructuras, evitando accesos directos o dependencias indebidas.

- Los enrutadores deben respetar la jerarquía de CMI: capas → módulos → contenedores. Los contenedores genéricos no requieren enrutador, ya que usan el de su padre.

- La existencia de un enrutador solo se justifica si la estructura necesita exponer elementos fuera de su contexto; si todo su contenido es interno, no se requiere.

- Mantener el enrutador actualizado al agregar, eliminar o modificar los elementos que expone es esencial para preservar la coherencia y estabilidad del proyecto.

Criterios para Expandir la Estructura

La **Arquitectura CMI** está diseñada para ofrecer un equilibrio entre una estructura normativa sólida y la capacidad de adaptarse a las necesidades reales de cada proyecto. Aunque las capas, módulos y contenedores estándar son obligatorios, existen situaciones donde es necesario **expandir la estructura** para mantener el orden a medida que el sistema crece.

Sin embargo, esta expansión debe realizarse siguiendo **criterios claros**. Expandir no significa crear carpetas por costumbre, sino tomar decisiones conscientes que respondan a problemas reales de organización y escalabilidad.

Creación de Módulos Adicionales

La estructura estándar de CMI está diseñada para cubrir casi todas las necesidades de un proyecto bien organizado. Por eso, la creación de **módulos adicionales** debe ser algo **poco frecuente** y siempre justificado.

Cada capa tiene un nivel distinto de flexibilidad para expandirse. Aquí te explicamos **cómo y cuándo** hacerlo correctamente.

Configs: La Capa Más Flexible. En configs/ es normal crear módulos adicionales. Esta capa está pensada para crecer según las necesidades del proyecto.

Puedes añadir módulos como:

- security/ Para gestionar certificados o permisos.

- environment/ Para configuraciones por entorno.

- Cualquier configuración global que requiera orden adicional.

Aquí expandir es parte natural de su función.

Core: La Capa Más Rígida. En core/, la regla es simple:

Regla: Si puedes resolverlo con rules/ y use/, no necesitas más módulos.
(Core: La Capa Más Rígida - Regla)

No importa si trabajas con APIs, servicios del dispositivo, sensores o almacenamiento local. Todo debe pasar por el flujo estándar de **services(origins adapters) consumers**.

Solo en casos **extremos**, como un motor autónomo de IA o procesamiento especializado que no encaje en este flujo, podrías crear un módulo adicional.

UI: Usa Bien lo Estándar. La capa ui/ ya es flexible por diseño. Flujos como onboarding, tutoriales o configuraciones visuales **se organizan perfectamente** en:

- **layouts/** Para flujos complejos.
- **shared/** Para elementos reutilizables.
- **pages/** Para pantallas independientes.

Crear un módulo adicional en UI debe ser algo **muy raro**. La clave está en **aprovechar bien los módulos estándar**.

Uso de Contenedores Internos y Recursividad

La recursividad en contenedores es una herramienta poderosa para organizar grandes volúmenes de código dentro de los contenedores estándar, especialmente en components/, views/ o services/.

¿Cuándo aplicar recursividad?

- Cuando un contenedor estándar empieza a manejar demasiados elementos de diferentes tipos o propósitos.
- Para separar por categorías funcionales, por ejemplo:

```
components/
 buttons/
 cards/
 forms/
```

***Listing 27**. Uso de Contenedores Internos y Recursividad - Ejemplo*

- Cuando necesitas agrupar elementos específicos sin justificar una categoría, puedes usar un contenedor genérico:

```
components/
 container/
```

Listing 28. *Uso de Contenedores Internos y Recursividad - Ejemplo Contenedor Generico*

Errores a evitar:

- Crear niveles innecesarios que compliquen la navegación del proyecto.
- Recursividad excesiva que dificulte localizar archivos.
- No usar enrutadores en contenedores con nombre que exponen elementos.

Uso Correcto de Contenedores Genéricos (container/)

El contenedor container/ es útil cuando necesitas agrupar elementos pero no hay una categoría definida o cuando el conjunto es muy específico del contexto.

Criterios para usar container/:

- Úsalo solo dentro de contenedores estándar o internos.
- No abuses de él para evitar pensar en una organización adecuada.
- Recuerda que **no requiere enrutador propio**, ya que depende del contenedor padre.

Ejemplo de Expansión Bien Aplicada

Supongamos que tu módulo shared/ empieza a manejar muchos tipos de formularios reutilizables y transiciones animadas:

```
ui/
 shared/
      components/
      dialogs/
      forms/
          container/
              auth_forms.dart
              profile_forms.dart
          forms_shared.dart           # Manejador de rutas internas de form
      transitions/
      logic/
```

Listing 29. *Ejemplo de Expansión Bien Aplicada*

Aquí se han creado dos **contenedores adicionales** (forms/ y transitions/), organizando elementos globales que no encajan en los contenedores estándar existentes.

Ejemplo Visual de Estructura CMI

A continuación, se presenta un ejemplo completo de cómo debe organizarse un proyecto siguiendo la **Arquitectura CMI**. Este ejemplo refleja la correcta aplicación de:

- Las **tres capas estándar**.
- Los **módulos estándar**.
- Los **contenedores de dominio** donde corresponda.
- Los **contenedores estándar** obligatorios.
- La presencia de **enrutadores** en cada nivel clave.
- Un uso básico de contenedores internos para organización adicional.

Este esquema es válido tanto para proyectos pequeños como para proyectos en crecimiento, ya que CMI siempre establece la estructura completa desde el inicio.

Estructura General del Proyecto CMI:

```
lib/
 main.dart
   # Punto de entrada de la óaplicacin.
   # Inicializa dependencias y lanza el widget íraz (App).

 configs/
     configs.dart
       # Enrutador principal de la capa Configs.
       # áExpondr las configuraciones generales como rutas, temas y ajustes globales.

 core/
     core.dart
      # Enrutador de la capa Core.
      # áExpondr los ómdulos Rules y Uses hacia el exterior.


     rules/
        data/
```

```
            data_rules.dart
              # Enrutador del contenedor Data en Rules.
              # Define las entidades, contratos de datos del dominio.

        services/
            services_rules.dart
              # Enrutador del contenedor Services en Rules.
              # Define los contratos de servicios de dominio.

        consumers/
            consumers_rules.dart
              # Enrutador del contenedor Consumers en Rules.
              # Define los contratos de consumidores de dominio.

        rules.dart
          # Enrutador principal del ómdulo Rules.

    uses/
        adapters/
            adapters_uses.dart
              # Enrutador del contenedor Adapters en Uses.
              # Define transformaciones entre entidades y estructuras de almacenamiento o infraestructura.

        services/
            services_uses.dart
              # Enrutador del contenedor Services en Uses.
              # Implementa ólgica de servicios de óaplicacin (uso de reglas, operaciones sobre datos, etc.).

        consumers/
            consumers_uses.dart
              # Enrutador del contenedor Consumers en Uses.
              # Implementa ólgica de consumo que conecta UI y servicios.

        origins/
            origins_uses.dart
```

```
53              # Enrutador del contenedor Origins en Uses.
54              # Define acceso o óadaptacin de datos de
  fuentes externas (API, almacenamiento local, etc.).
55
56          uses.dart
57            # Enrutador principal del ómdulo Uses.
58
59 ui/
60     ui.dart
61      # Enrutador principal de la capa UI.
62      # áExpondr ómdulos como Layouts, Pages, Shared hacia
  la ópresentacin.
63
64     app/
65         app.dart
66           # Widget íraz de la óaplicacin (MaterialApp,
  óconfiguracin de rutas, temas, etc.).
67
68     layouts/
69         layouts.dart
70           # Enrutador principal de Layouts.
71           # Encargado de gestionar los layouts generales de
   la app (estructura visual fija).
72
73     pages/
74         pages.dart
75           # Enrutador principal de Pages.
76           # óGestin de ápginas independientes de layouts (
  como login, registro, error 404, etc.).
77
78     shared/
79         components/
80             components_shared.dart
81               # Enrutador del contenedor Components en
  Shared.
82               # Elementos visuales reutilizables (botones,
  inputs, etc.).
83
84         logic/
85             logic_shared.dart
86               # Enrutador del contenedor Logic en Shared.
87               # óLgica reutilizable a nivel general (
```

```
   providers, helpers, etc.).
88
89         dialogs/
90             dialogs_shared.dart
91               # Enrutador del contenedor Dialogs en Shared.
92               # Ventanas emergentes, modales y componentes
   de óinteraccin secundaria.
93
94         shared.dart
95           # Enrutador principal de Shared.
```

***Listing 30**. Estructura General del Proyecto CMI*

Convenciones de Nomenclatura

Una parte fundamental de la **Arquitectura CMI** es la consistencia en la manera de nombrar y estructurar cada elemento del proyecto. El respeto por estas normas garantiza claridad, facilita el trabajo en equipo y evita errores o confusiones en proyectos de cualquier tamaño.

Esta sección define las **reglas precisas** para nombrar **capas**, **módulos**, **contenedores**, **archivos** y, en especial, los **enrutadores**, asegurando que toda la estructura siga un estándar comprensible y mantenible.

Principios Generales

- **Contexto primero:** Todo nombre debe reflejar su lugar dentro de la arquitectura. La ruta + nombre del archivo deben ser suficientes para entender su propósito.
- **Evita nombres genéricos:** Nunca uses nombres como utils.dart, data.dart o router.dart sin especificar contexto.
- **Claridad sobre brevedad:** Prefiere nombres descriptivos aunque sean un poco más largos, antes que abreviaciones o nombres ambiguos.
- **Consistencia total:** Aplica las mismas reglas en todo el proyecto, sin excepciones.

Nomenclatura de Capas

Las capas tienen nombres **predefinidos y únicos**:

***Table 12**. Nomenclatura de Capas*

Capa	Nombre de Carpeta
Configs	configs/
Core	core/
UI	ui/

Estas carpetas no deben renombrarse, ya que forman la base estructural de CMI.

Nomenclatura de Módulos

Módulos por Defecto = Nombre Fijo y Estándar

En CMI, ciertos módulos ya están **predefinidos** por la arquitectura. Estos deben mantener su nombre **exacto y sin modificaciones**, porque representan funciones universales dentro del esquema de CMI.

Lista de Módulos por Defecto en CMI:

***Table 13**. Módulos por Defecto = Nombre Fijo y Estándar*

Capa	Módulo por Defecto
Core	rules/, use/
UI	layouts/, shared/, pages/, app/
Configs	Depende de la configuración (ej: routes/, theme/, etc.)

Reglas:

1. Mantener el nombre fijo: Los módulos como rules/, use/, shared/, layouts/, entre otros, deben conservar su nombre exacto.

2. Evitar redundancias en nombres de archivos: No repitas el nombre del módulo dentro de los archivos que ya están contextualizados por la ruta.

3. Enrutadores de módulos: Usa la convención [módulo]_[capa].dart para nombrar los enrutadores de estos módulos.

4. Contenedores: Si creas contenedores dentro de estos módulos, aplica las normas generales de nomenclatura, siempre priorizando la claridad y el contexto.

Nomenclatura de Contenedores

Los contenedores deben reflejar el tipo de elementos que agrupan:

Table 14. *Nomenclatura de Contenedores*

Tipo de Contenedor	**Nombre Sugerido**
Componentes	components/
Vistas	views/
Lógica	logic/
Diálogos	dialogs/
Servicios	services/
Adaptadores	adapters/
Orígenes	origins/

Para subcontenedores específicos, añade descripciones claras:

```
components/container/ //Generico
components/buttons/ //Especifico
```

Listing 31. *Nomenclatura de Contenedores*

Nomenclatura de Archivos

La estructura general para nombrar archivos es:

```
[nombre_base]_[tipo]_[contexto].dart
```

Listing 32. *Nomenclatura de Archivos-Estructura general*

Ejemplos Correctos:

Table 15. *Nomenclatura de Archivos*

Elemento	Nombre Correcto
Componente	button_component_shared.dart
View	login_view_authentication.dart
Layout	authentication_layout.dart
Adapter	chat_adapter_use.dart
Service Rule	task_service_rule.dart

Normas Específicas para Enrutadores

Los **enrutadores** son casos especiales dentro de la nomenclatura.
Reglas:

1. **El nombre del enrutador** debe ser el de la capa, módulo o contenedor que expone**:**
   (a) configs/configs.dart
   (b) layouts/authentication/authentication_layout.dart
2. **No uses nombres genéricos** como **router.dart** o **routes.dart.**
3. Si hay riesgo de confusión, añade el contexto superior:
   (a) Ejemplo en subniveles: views/profile/profile_view.dart
4. Contenedores genéricos no deben tener enrutador propio.

Manejo de Nombres Repetidos

Si te encuentras con posibles conflictos de nombres:

- **Agrega el contexto funcional:**
  - En lugar de repetir data.dart, usa: user_data_rule.dart y message_data_rule.dart.
- **Evita abreviaturas oscuras o genéricas.**
- Siempre prioriza la legibilidad y la identificación rápida del propósito del archivo.

Clases

La correcta elección de nombres en las clases es fundamental para mantener la claridad, coherencia y mantenibilidad dentro de la Arquitectura CMI.

Reglas Generales:

1. **Descriptivos y Precisos:** Utiliza nombres que reflejen claramente la **responsabilidad** y el **propósito** de la clase. Evita términos genéricos como Manager, Handler o Data sin contexto adicional.
2. **Uso de Sustantivos Claros:** Las clases deben representar **entidades**, **conceptos** o **roles** dentro del dominio de la aplicación. Ejemplos correctos:
   (a) NoteConsumerRule
   (b) TaskAdapterUse
   (c) ThemeLogic
3. **Consistencia con la Estructura CMI** Los nombres deben facilitar la comprensión inmediata de **dónde** y **para qué** se usa cada clase:
   (a) En rules/: Añade sufijos como Rule para indicar abstracción.
   (b) En use/: Usa sufijos como Use para señalar implementación.
   (c) En ui/logic/: Termina en Logic + contexto, o LogicShared si es global.
   (d) En adapters/: Siempre terminar en AdapterUse.
4. **Evitar Abreviaturas Ambiguas:** No utilices siglas o abreviaturas que no sean universalmente reconocidas. Prefiere siempre la **claridad** sobre la brevedad excesiva.
   (a) Ejemplo incorrecto: NtCnsmrRl
   (b) Ejemplo correcto: NoteConsumerRule

Convenciones de Sufijos en Clases:

Cada clase debe terminar con un sufijo que indique su propósito, según su ubicación en la estructura CMI.

Table 16. *Convenciones de Sufijos en Clases*

Ubicación	**Ejemplo de Clase**	**Sufijo**
core/rules/services/	NoteServiceRule	Rule
core/use/services/	NoteServiceUse	Use
core/use/adapters/	TaskAdapterUse	AdapterUse
core/rules/data/	TaskDataRule	DataRule
ui/**/logic/	NotesLogic	Logic
ui/**/components/	CustomButtonComponent	Component
ui/layouts/authentication/	AuthenticationLayout	Layout
configs/routes/	AppRoutesConfig	RoutesConfig
configs/theme/	ThemeConfig	Config

Ejemplos Aplicados:

```
// Componente UI dentro de layouts/authentication/components
class PrimaryElevatedButtonComponent extends StatelessWidget
    {}

// Logic de estado en UI
class NotesLogic {}

// óAbstraccin en core/rules/services
```

```
8  abstract class NoteServiceRule {}
9
10 // óImplementacin en core/use/services
11 class NoteServiceUse implements NoteServiceRule {}
12
13 // Adaptador bidireccional
14 class TaskAdapterUse {}
15
16 // Enrutador de layout
17 class NotesLayout extends StatelessWidget {}
```

***Listing 33**. Forma correcta de nombrar las clases*

Errores Comunes:

- Nombres vagos como Processor, Helper, Service sin contexto.
- Mezclar idiomas o estilos (ej: TareaService en un código en inglés).
- Usar nombres largos por describir **todo**. Busca el equilibrio.

Funciones

En la Arquitectura CMI, la nomenclatura de las funciones debe ser clara, precisa y coherente con las acciones que representan. Una función correctamente nombrada facilita la lectura del código, mejora la comprensión del dominio de la aplicación y asegura la mantenibilidad a largo plazo.

A continuación, se establecen las reglas que deben seguirse para definir nombres de funciones dentro de cualquier proyecto basado en CMI.

Reglas Generales:

1. **Ser descriptivo y conciso:** El nombre de una función debe reflejar claramente su propósito. Se deben evitar términos genéricos o ambiguos que no indiquen con exactitud la operación que realiza. Es preferible utilizar nombres directos que comuniquen la acción sin necesidad de revisar la implementación de la función.
   (a) **Ejemplo correcto:**
      i. createNote()
      ii. deleteTask()
   (b) **Ejemplo incorrecto:**
      i. handleData()
      ii. processInfo()

2. **Utilizar verbos que indiquen acción:** Toda función debe comenzar con un **verbo** que denote la acción principal que ejecuta. Esto aplica a cualquier contexto, ya sea en Consumers, Services, Adapters o Logics. Los verbos deben ser claros y relacionados con el dominio de la aplicación.
    (a) **Ejemplos:**
        i. fetchNotes()
        ii. saveUserProfile()
        iii. toggleDarkMode()
3. **Mantener coherencia en el estilo:** Se debe utilizar el estilo **camelCase** para todas las funciones. Es obligatorio mantener un estilo uniforme en todo el proyecto, evitando mezclar idiomas o estilos de nomenclatura.
    (a) **Ejemplo correcto:** updateUserSettings()
    (b) **Ejemplo incorrecto:**
        i. Update_user_settings()
        ii. actualizarUsuario()
4. **Reflejar el dominio del problema:** Los nombres de las funciones deben incorporar términos propios del **dominio** en el que se está trabajando. Esto permite que el código sea más expresivo y entendible dentro del contexto de la aplicación.
    (a) **Ejemplo correcto:**
        i. archiveNote() (si la acción es específica de un sistema de notas)
    (b) **Ejemplo incorrecto:**
        i. setFlag() (demasiado genérico)
5. **Evitar verbos ambiguos:** Se deben evitar verbos como handle, process, manage, salvo que describan una acción concreta y necesaria dentro del dominio. Siempre que sea posible, optar por verbos más específicos.
    (a) **Ejemplo correcto:** assignTaskToUser()
    (b) **Ejemplo incorrecto:** handleTask()

```
1 //   configs/theme/theme_config.dart
2 ThemeData getAppTheme(bool isDarkMode) {
3   return isDarkMode ? ThemeData.dark() : ThemeData.light();
4 }
5
```

```
//   configs/routes/routes_config.dart
String generateNoteDetailPath(String noteId) {
  return '/notes/details/$noteId';
}

//   utils/validators.dart
bool isValidEmail(String email) {
  return RegExp(r'^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$').
   hasMatch(email);
}

//   utils/date_utils.dart
String formatDate(DateTime date) {
  return '${date.day.toString().padLeft(2, '0')}/'
         '${date.month.toString().padLeft(2, '0')}/'
         '${date.year}';
}

//   utils/string_utils.dart
String capitalize(String text) {
  if (text.isEmpty) return '';
  return text[0].toUpperCase() + text.substring(1).
   toLowerCase();
}
```

***Listing 34**. Forma correcta de nombrar las funciones*

Metodos

En la Arquitectura CMI, los **métodos** son el principal medio para definir el comportamiento dentro de las clases, ya sea en Consumers, Services, Adapters, Logics o componentes UI. Por ello, su nomenclatura debe ser precisa, coherente y alineada con la responsabilidad de la clase donde se definen.

Reglas Generales:

1. **Usar verbos que indiquen acción:** Todo método debe comenzar con un **verbo** claro que describa la operación que realiza, reflejando la intención de manera directa.

2. **Evitar redundancia con el nombre de la clase:** Si la clase ya define el contexto o dominio (como NoteConsumerUse), no es necesario repetir ese contexto en el nombre del método.

   (a) **Ejemplo correcto:** create()

   (b) **Ejemplo incorrecto:** createNote()

3. **Ser específico y conciso:** El método debe describir claramente su propósito sin ser excesivamente largo ni ambiguo. Evitar términos genéricos como handle(), process(), o doWork().

4. **Métodos que gestionan eventos en UI:** Es recomendable usar el prefijo **on** para métodos que respondan a eventos, especialmente en Logic o Componentes:

   (a) onSubmitPressed()

   (b) onNoteSelected()

5. **Métodos privados claros:** Los métodos privados deben comenzar con guion bajo (_) y ser igual de descriptivos que los públicos:

   (a) **Ejemplo correcto:** _validateInput()

   (b) **Ejemplo incorrecto:** _doSomething()

6. **Consistencia en el estilo:**

   (a) Utilizar siempre **camelCase**.

   (b) Mantener coherencia en todo el proyecto.

   (c) Evitar mezclar idiomas.

```
1  // Consumer de notas (core/use/consumers/)
2  class NoteConsumerUse implements NoteConsumerRule {
3    void create(String content) {}
4    List<String> fetchAll() => [];
5    void delete(String id) {}
6  }
7
8  // Servicio de óautenticacin (core/use/services/)
9  class AuthenticationServiceUse implements
       AuthenticationServiceRule {
10   Future<void> login(String username, String password) async
       {}
11   Future<void> logout() async {}
12 }
13
14 // Adaptador de tareas (core/use/adapters/)
15 class TaskAdapterUse {
16   TaskDataRule fromJson(Map<String, dynamic> json) { /*...*/
       }
```

```
  Map<String, dynamic> toJson(TaskDataRule task) { /*...*/ }
}

// Logic en UI (ui/layouts/notes/logic/)
class NotesLogic {
  void toggleSelectionMode() {}
  void onNoteSelected(String id) {}
  void _filterNotesByDate(DateTime date) {}
}

// Componente UI (ui/shared/components/)
class ConfirmButtonComponent extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () => onConfirmPressed(),
      child: const Text('Confirm'),
    );
  }

  void onConfirmPressed() {
    // óAccin al confirmar
  }
}
```

Listing 35. *Forma correcta de nombrar los metodos*

Variables

Las variables son elementos fundamentales en cualquier proyecto. En la Arquitectura CMI, el uso correcto de nombres para variables garantiza un código **claro**, **legible** y fácil de mantener. Una variable bien nombrada evita confusiones y facilita la comprensión tanto del contexto como de la intención del código.

A continuación, se establecen las recomendaciones obligatorias para la nomenclatura de variables en proyectos basados en CMI.

Reglas Generales:

1. **Utilizar nombres descriptivos y específicos:** Toda variable debe reflejar claramente el propósito de su contenido. El nombre debe indicar **qué representa** o **qué almacena**, evitando ambigüedades.
   (a) **Ejemplo correcto:** noteList, isDarkModeEnabled, userEmail
   (b) **Ejemplo incorrecto:** data, value, temp

2. **Evitar nombres genéricos o innecesariamente cortos:** Solo en contextos muy limitados, como iteraciones simples, es aceptable usar nombres como i, j. En cualquier otro caso, los nombres deben ser significativos.
   (a) **Ejemplo correcto:** `for (var index = 0; index < notes.length; index++) { }`
   (b) **Ejemplo incorrecto:** `var x = notes.length;`
3. **Seguir el estilo camelCase:** Todas las variables deben definirse utilizando **camelCase**, sin excepciones.
   (a) **Ejemplo correcto:** selectedTaskId
   (b) **Ejemplo incorrecto:** Selected_task_id
4. **Prefijos para variables booleanas:** Las variables de tipo booleano deben comenzar con palabras que indiquen una condición o estado.
   (a) **Ejemplo correcto:** isActive, hasPermission, canEdit
   (b) **Ejemplo incorrecto:** active, permission, editFlag
5. **Consistencia con el dominio del problema** Utiliza términos que pertenezcan al contexto de la aplicación. Esto facilita que el código sea más intuitivo para el equipo.
   (a) **Ejemplo correcto:** archivedNotesCount
   (b) **Ejemplo incorrecto:** counter (si se refiere al número de notas archivadas)

```
// Ejemplo en un Logic
class NotesLogic {
  List<NoteDataRule> noteList = [];
  bool isSelectionModeEnabled = false;
  String selectedNoteId = '';
}

// Ejemplo en un Service
class AuthenticationServiceUse {
  String userToken = '';
  bool isLoggedIn = false;
}

// Ejemplo en UI
final String confirmButtonLabel = 'Confirm';
```

```
16 final int maxNoteLength = 500;
```

***Listing 36**. Forma correcta de nombrar las variables*

Buenas Prácticas en CMI

La **Arquitectura CMI** no solo proporciona una estructura clara y principios sólidos, sino que también promueve un conjunto de **buenas prácticas** que aseguran que el desarrollo sea limpio, sostenible y eficiente.

Adoptar estas prácticas desde el inicio de un proyecto garantiza que el código sea más fácil de mantener, escalar y comprender, especialmente en entornos colaborativos o en aplicaciones de larga duración.

Organización del Proyecto

- **Respeta la estructura de Capas, Módulos y Contenedores**: Mantén siempre la jerarquía definida por CMI. Evita crear carpetas o archivos fuera de su contexto correspondiente.
- **Evita la sobrecarga de contenedores**: Aunque es posible anidar contenedores, limita la profundidad para no complicar la navegación del proyecto.
- **Centraliza las configuraciones**: Todo lo relacionado con constantes, rutas, temas, idiomas, etc., debe estar en la capa configs. No dupliques configuraciones en otras estructuras.

Manejo de Dependencias

- **Aplica siempre Inyección de Dependencias (DI)**: Nunca instancies directamente servicios o consumidores dentro de la UI. Utiliza herramientas como **Riverpod**, **GetIt**, o el sistema que prefieras, pero mantén las dependencias desacopladas.
- **Usa abstracciones en lugar de implementaciones concretas**: Trabaja siempre con las interfaces definidas en rules y permite que uses provea la implementación.

Gestión del Estado

- **Prefiere estados reactivos y controlados**: Implementa soluciones como **Riverpod**, **Bloc** o similares para manejar el estado de la aplicación. Evita el uso excesivo de estados locales (setState) en proyectos medianos o grandes.
- **Ubica la lógica en su lugar correspondiente**: La lógica debe estar en los contenedores logic/ dentro de la UI. No mezcles lógica en los widgets o consumas directamente los consumers.

Nomenclatura y Legibilidad

- **Sigue estrictamente las convenciones de nomenclatura**: Mantén nombres claros, descriptivos y evita abreviaciones innecesarias. Un buen nombre reduce la necesidad de comentarios adicionales.

- **Documenta las clases y métodos complejos**: Aunque CMI promueve la claridad por estructura, siempre documenta aquellas partes que puedan ser confusas o críticas.

Reutilización y Componentización

- **Crea componentes reutilizables en shared**: Todo elemento UI o lógica que pueda ser usado en más de un módulo debe ir a shared para evitar duplicación de código.

- **No abuses de los componentes genéricos**: Si un componente es muy específico de un layout, debe quedarse dentro del contenedor de dominio correspondiente.

Manejo de Errores

- **Implementa manejo de errores centralizado**: Define estrategias globales para capturar excepciones, especialmente en la capa core/use/services. La UI solo debe mostrar mensajes amigables, nunca manejar la lógica de errores directamente.

Testing

- **Escribe pruebas unitarias para rules y uses**: La abstracción de CMI facilita el testing. Asegúrate de cubrir la lógica de negocio y las integraciones.

- **Pruebas de UI limitadas pero efectivas**: Testea los widgets más críticos, especialmente aquellos que gestionan estados complejos o flujos importantes.

Control de Versiones y Colaboración

- **Organiza ramas siguiendo la estructura de CMI**: Cuando trabajes en equipo, estructura las ramas Git por módulo o funcionalidad siguiendo la lógica de la arquitectura.

- **Utiliza Pull Requests con revisiones enfocadas en respetar la arquitectura**: Asegúrate que cada aporte siga las buenas prácticas y no rompa la coherencia estructural.

Optimización y Desempeño

- **Minimiza la carga en la UI**: Divide las vistas en Part cuando sea necesario y utiliza componentes livianos.
- **Carga perezosa (Lazy Loading)**: Implementa estrategias para cargar datos o vistas solo cuando sean requeridos, especialmente en módulos grandes.

Compatibilidad Tecnológica de la Arquitectura CMI

La Arquitectura CMI fue concebida originalmente en el contexto del desarrollo con **Flutter**, aprovechando las ventajas de modularidad, tipado estricto y escalabilidad que ofrece la plataforma.

Sin embargo, debido a que CMI es un marco de **organización estructural** y no una arquitectura ligada a un lenguaje o tecnología específica, sus principios pueden aplicarse de manera flexible en otros entornos de desarrollo.

Este capítulo presenta una guía de compatibilidad de CMI con distintos **lenguajes de programación**, **frameworks**, **librerías** y **gestores de estado o navegación**.

Principios que permiten la adaptabilidad de CMI

CMI es adaptable a cualquier entorno que permita cumplir con sus principios fundamentales:

- **Modularidad:** División clara en capas, módulos y contenedores.
- **Separación de responsabilidades:** Cada elemento debe cumplir una función específica.
- **Bajo acoplamiento:** Los elementos deben estar conectados de manera mínima y controlada.
- **Alta cohesión:** Los elementos relacionados deben mantenerse agrupados lógicamente.
- **Control de accesos:** A través de mecanismos como enrutadores o puntos de exportación bien definidos.

Siempre que un lenguaje o framework permita aplicar estos principios, CMI puede ser implementada de forma efectiva.

Compatibilidad con Lenguajes de Programación

***Table 17**. Compatibilidad con Lenguajes de Programación*

Lenguaje	Compatibilidad	Notas
Dart (Flutter)	Totalmente compatible	Plataforma base donde se desarrolló CMI. Modularidad natural, tipado estricto y soporta generación de código.
TypeScript (Web, React)	Compatible teórico	Modularidad fuerte, ideal para frontend web moderno.
Kotlin (Android, KMP)	Compatible teórico	Modularización clara en Android. KMP permite estructura multiplataforma compatible con CMI.
Swift (iOS)	Compatible teórico	Puede integrarse respetando la separación de capas y modularización.
C# (Xamarin, MAUI, Blazor)	Compatible teórico	Estructuras naturales en proyectos de C#. Buen soporte para arquitectura limpia.
Java (Android)	Compatible teórico	Con disciplina, puede seguir las divisiones de CMI sin problemas.

Continued on next page

Table 17. *Compatibilidad con Lenguajes de Programación* (Continued)

JavaScript clásico	Compatible limitada	Necesita mucha disciplina para modularizar bien. Mejor usar TypeScript.
Rust (Frontend, WebAssembly)	Experimental	Modularización posible, pero poco natural en frontend tradicional.

Compatibilidad con Frameworks y Librerías

Table 18. *Compatibilidad con Frameworks y Librerías*

Framework / Librería	**Compatibilidad**	**Notas**
Flutter	Totalmente compatible	Arquitectura original de desarrollo de CMI.
React (Web)	Compatible teórico	Modularización natural con componentes y rutas.
Angular (Web)	Compatible teórico	Modularidad de Angular facilita aplicar CMI.
Vue.js	Compatible teórico	Componentización fuerte, adaptable a CMI.
Xamarin / MAUI (C#)	Compatible teórico	Arquitectura por capas facilita la integración.

Continued on next page

Table 18. *Compatibilidad con Frameworks y Librerías* (Continued)

Jetpack Compose (Android/Kotlin)	Compatible teórico	Composable UI modularizable en layouts y componentes.
Next.js (Web, React SSR)	Compatible limitada	El enrutamiento automático puede complicar la separación clara de enrutadores internos.
Blazor (Web, C#)	Compatible teórico	Modularización de componentes en Blazor permite seguir CMI.

Compatibilidad con Gestores de Estado y Navegadores

Los gestores de estado y sistemas de navegación descritos a continuación corresponden al ecosistema Flutter. En otros entornos de desarrollo, se deberá seleccionar herramientas equivalentes que respeten los principios de modularidad, separación de responsabilidades y control de dependencias que CMI exige. (Nota - Compatibilidad con Gestores de Estado y Navegadores)

Gestores de Estado

Table 19. *Compatibilidad con Gestores de Estado*

Gestor de Estado	**Compatibilidad**	**Notas**
Riverpod	Totalmente compatible	Manejo de estado reactivo, generación de código, modularización perfecta.

Continued on next page

Table 19. *Compatibilidad con Gestores de Estado* (Continued)

flutter_bloc / BLoC	Compatible	Modularización por dominio. Alto control de flujo.
Cubit	Compatible	Manejo de estado simple pero modularizado.
Redux	Compatible con advertencias	Requiere esfuerzo extra para modularizar el store de manera limpia por dominios.
Provider clásico	Compatible a pequeña escala	Aceptable en proyectos pequeños; difícil de escalar limpiamente en proyectos grandes.
InheritedWidget	Compatible pero tedioso	Mucho trabajo manual para respetar modularidad.

Sistemas de Navegación

Table 20. *Compatibilidad con Sistemas de Navegación*

Sistema de Navegación	**Compatibilidad**	**Notas**
go_router	Totalmente compatible	Modularización y generación de rutas limpias.

Continued on next page

Table 20. *Compatibilidad con Sistemas de Navegación* (Continued)

auto_route	Compatible	Excelente para proyectos modulares y rutas internas.
Navigator 2.0 manual	Compatible	Declarativo, pero requiere esfuerzo manual.
Navigator 1.0 clásico	Compatible pero desaconsejado	Difícil mantener la claridad de navegación a gran escala.
Beamer	Compatible	Especialmente útil para navegación jerárquica modular.

Notas sobre Aplicación Multiplataforma

CMI está diseñada para ser completamente multiplataforma.
Puede ser aplicada en:

- Aplicaciones móviles (Android, iOS)
- Aplicaciones web (SPA, SSR)
- Aplicaciones de escritorio (Flutter Desktop, Electron, MAUI)
- Aplicaciones multiplataforma (KMP, Flutter, .NET MAUI)

Siempre que se mantenga la organización y modularidad exigida, la arquitectura sigue funcionando de forma efectiva.

Conclusión

La **Arquitectura CMI** establece un marco sólido y normativo para el desarrollo de proyectos de software, especialmente en entornos donde la organización, escalabilidad y mantenibilidad son esenciales. A través de sus pilares —**capas**, **módulos**, **contenedores** y **enrutadores**—, CMI garantiza que cada proyecto crezca de forma ordenada, evitando los problemas comunes de arquitecturas flexibles o mal definidas.

Aplicar CMI no solo implica seguir una estructura, sino adoptar una filosofía de trabajo disciplinada que prioriza la claridad, el control y la preparación para el futuro.

Este documento ha presentado las bases teóricas, normativas y prácticas necesarias para implementar la arquitectura de manera efectiva. A continuación, se desarrollará un ejemplo práctico que ilustrará cómo aplicar estos conceptos en un caso real, consolidando todo lo aprendido.