Aplicación 1

Chat IA

Planteamiento

Creación de un chat web inteligente implica:

- Servicio de **IA / Procesamiento de texto**
- Servicio de **Mensajería**
- Servicio de **Usuarios y autenticación**
- Servicio de **Base de datos**
- Servicio de **Backend API**
- Servicio de **Frontend / Interfaz**
- Servicio de **Infraestructura / DevOps**
- Servicio de **Seguridad y moderación**
- Servicio de **Memoria / Contexto conversacional**
- **Servicio de notificaciones**

Proceso

- Recibir mensajes del usuario.
- Procesar (enviar a un modelo IA o API).
- Devolver una respuesta.
- Guardar el historial.
- Manejar usuarios y sesiones.

Implementación incorrecta

1. leer el mensaje
2. limpiar texto
3. llamar API IA
4. guardar en base de datos
5. emitir respuesta

Functional Decomposition

Functional Decomposition

- Consiste en **dividir el sistema en funciones pequeñas y bien definidas**, cada una con una sola responsabilidad (“separation of concerns”).

Ventajas:

- Cada función puede probarse unitariamente.
- Puedes reemplazar el motor IA o el almacenamiento sin tocar el controlador.
- Mayor claridad y mantenibilidad.

Desventaja:

- Si no se controla, puede haber demasiadas funciones dispersas (debes estructurar bien por dominio).

Separación por niveles de responsabilidad

Presentación

- Controladores que reciben y devuelven datos

Lógica de negocio

- Procesos que dan sentido a la app

Servicios

- Comunicación con recursos externos

Utilidades

- Funciones puras o helpers

Estructura

- chat_app/
  - app.py
- controllers/ # Capa de presentación
  - chat_controller.py
- core/ # Lógica de negocio
  - chat_manager.py
- services/ # Capa de servicios
  - ai_service.py
  - db_service.py
  - auth_service.py
- utils/ # Utilidades generales
  - text_utils.py
  - time_utils.py
- models/ # Definición ORM (SQLAlchemy)
  - chat_message.py

Servicios de Utilidad

- Limpieza y normalización de texto.
- Logs o manejo de excepciones.
- Conversión de formatos (JSON ↔ texto).
- Utilidades de tiempo (timestamp, zonas horarias).
- Validación de datos genérica.

Servicios de Entidad

- Guardar y recuperar mensajes de chat.
- Registrar usuarios.
- Obtener el historial de una sesión.

Entidades:

- Usuario
- Sesión
- Mensaje

Servicios de Negocio

1. Validar entrada.
2. Consultar IA.
3. Guardar conversación.
4. Devolver respuesta al usuario.

1 Validar la entrada

- El usuario envía un mensaje.
- Antes de procesarlo, el sistema debe validar formato, tamaño, limpieza, seguridad, etc.

Qué servicios intervienen

- **Servicio de utilidad**
- Encargado de operaciones generales o funciones puras
- **Servicio de negocio**
- Decide qué hacer si el texto es inválido (regla de negocio)

2 Consultar IA

- El sistema necesita generar una respuesta inteligente usando un modelo de lenguaje.

Qué servicios intervienen

- **Servicio de entidad**
- Encapsula la lógica de acceso a la IA (OpenAI, LLaMA, modelo local).
- **Servicio de negocio**
- Decide cómo usar esa respuesta: ¿se guarda?, ¿se filtra?, ¿se resume?

3 Guardar conversación

- El sistema persiste el mensaje del usuario y la respuesta del bot.

Qué servicios intervienen

- **Servicio de entidad**
- Maneja la comunicación con la base de datos.
- **Servicio de negocio**
- Orquesta el flujo: cuándo y qué guardar.
- **Servicio de utilidad**
- Opcionalmente formatea fechas, logs, IDs.

4 Devolver la respuesta al usuario

- Se envía la respuesta del bot al frontend, por WebSocket o HTTP.

Qué servicios intervienen

- **Servicio de negocio**
- Determina qué información devolver y en qué formato.
- **Servicio de utilidad**
- Serializa o transforma datos.
- **Presentación**
- Envía al cliente (controlador Flask).

Actividad U3_1

- Diseñar una base de datos para la aplicación
- Contemplar las siguientes entidades:

Entidades:

- Usuario
- Sesión
- Mensaje

- Presentar diagrama E-R

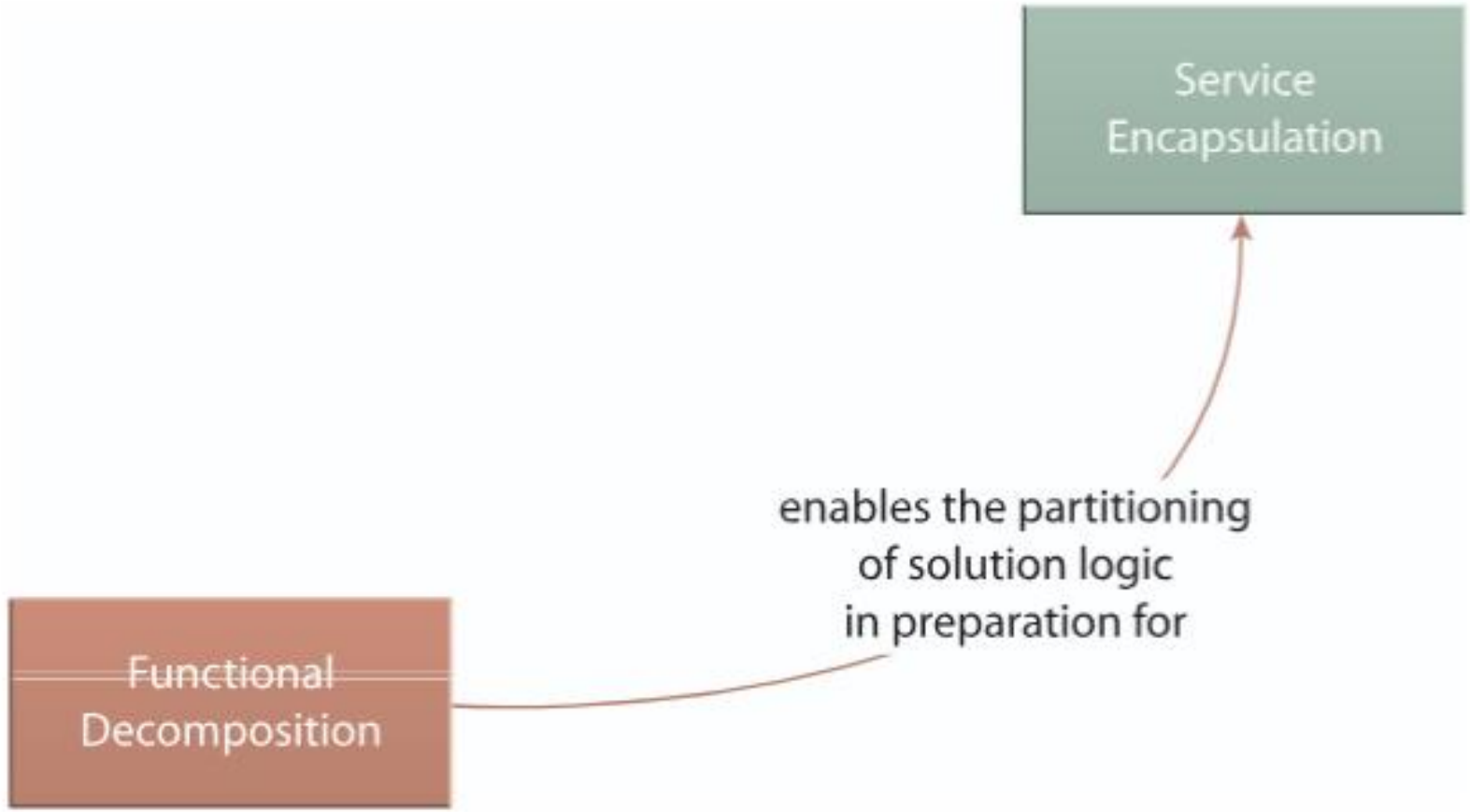


Figure 11.6

This displayed relationship simply establishes how the reasoning behind decomposing functionality is to make the decomposed parts available for potential encapsulation by services.

Service Encapsulation

Service Encapsulation

- Cada servicio debe **ocultar su implementación interna** detrás de una **interfaz clara**.
- Los demás componentes solo ven “qué hace”, no “cómo lo hace”.

Ventajas:

- Puedes cambiar de proveedor IA o backend sin tocar el resto del código.
- Facilita testing (puedes simular IA durante pruebas).
- Mejora la separación de dominios (cada servicio encapsula su lógica).

Desventaja:

- Más abstracción → más código inicial.
- Puede ser excesivo si tu app es muy pequeña.

Service Encapsulation

En una app de chat:

- La lógica de *gestionar sesiones de conversación*,
- *comunicarse con la IA*,
- *almacenar mensajes en base de datos*,
  puede considerarse un **recurso empresarial** si es útil para más de una aplicación (por ejemplo, para un chatbot web, un chatbot móvil o una API interna).

Por tanto, esa lógica **no debe estar atada a una plataforma ni al frontend**, sino encapsulada como un servicio modular.

Pasos para implementar el patrón Service Encapsulation

Paso 1: Identificar la lógica de solución

- Analiza qué componentes del chat tienen **funcionalidad empresarial** (no solo de UI):

1. Generación de respuesta con IA.
2. Persistencia de mensajes.
3. Administración de usuarios o sesiones.
4. Procesamiento o filtrado de texto.

Criterios para identificar la lógica adecuada

Antes de encapsular, debes **filtrar la lógica:**

- **¿Es útil más allá del límite inmediato de la app?**
- La función podría ser usada por otras apps (chat web, soporte interno, analítica).
- **¿Aprovecha o representa un recurso empresarial?**
- El modelo de IA y las conversaciones guardadas son recursos reutilizables.
- **¿Tiene limitaciones técnicas que impidan encapsularla?**
- Si depende fuertemente de un framework o base de datos específica, habría que desacoplarla primero.

Nota: Solo la lógica que cumpla estos criterios debe convertirse en servicio encapsulado.

lógica	Tipo de Servicio	¿Es útil más allá del límite inmediato de la app?	¿Aprovecha o representa un recurso empresarial?	¿Tiene limitaciones técnicas que impidan encapsularla?	Evaluación general / Acción recomendada
Usuario	Entidad	Sí. Puede ser usado por cualquier sistema que gestione cuentas o autenticación.	Si, identidad y datos de usuarios.	Podría depender del framework o una base de datos concreta; conviene desacoplar.	**Encapsular** como UserService con interfaz estable para registro, autenticación y consulta.
Mensaje	Entidad	Sí. Los mensajes pueden ser analizados, auditados o usados en reportes fuera del chat.	Si, contiene información de comunicación.	Podría depender del framework o una base de datos concreta; conviene desacoplar.	**Encapsular totalmente**, definiendo una entidad “canónica” de mensaje.
Consultar IA	Negocio	Sí. Podría ser usado por distintas aplicaciones (chat web, asistentes, dashboards, etc.).	Si, es clave: lógica de generación de conocimiento / respuesta.	Puede depender del proveedor (OpenAI API, HuggingFace).	**Encapsular completamente** como AIService o ConversationalAIServic e.
Guardar conversación	Negocio	Sí. Puede ser reutilizado por analítica, reportes o integraciones con CRM.	Si (historial de comunicación).	Si depende de un ORM o motor de base de datos específico, encapsular acceso mediante un repositorio.	**Encapsular totalmente** como ConversationService, desacoplado de la persistencia.

Pasos para implementar el patrón Service Encapsulation

Paso 2: Separar la lógica agnóstica de la dependiente

- Divide tu código en dos partes:
- **Agnóstica:** no depende de Flask ni del frontend (ej. AIService, DatabaseService).
- **No Agnóstica (dependiente) :** maneja transporte o protocolo (Flask endpoints, WebSocket).

Nota: Esto es el corazón del patrón: **aislar la lógica que representa el valor empresarial**.

Lógica	Grupo	Descripción técnica	Justificación
AIService (Consultar IA)	Agnóstica	Se comunica con un modelo externo (OpenAI, HuggingFace, etc.) mediante cliente API configurable.	No depende de Flask; puede usarse desde cualquier aplicación o script.
ConversationService (Guardar conversación)	Agnóstica	Gestiona persistencia del historial mediante un repositorio o DAO.	Puede encapsularse con una interfaz (ConversationRepository) sin atarse a Flask ni frontend.
UserService (Entidad: Usuario)	Agnóstica	Gestiona usuarios y autenticación lógica.	Puede usar un ORM (SQLAlchemy), pero si se abstrae la persistencia, se mantiene agnóstico.
MessageService (Entidad: Mensaje)	Agnóstica	Define estructura y manejo lógico de mensajes (creación, validación, consulta).	Independiente de HTTP; puede funcionar con cualquier fuente de datos.
ResponseHandler / Devolver respuesta al usuario	No Agnóstica	Envia respuestas al cliente por HTTP o WebSocket.	Completamente dependiente del protocolo Flask / SocketIO.
APIController / Flask Endpoints	No Agnóstica	Expone rutas REST o WebSocket.	No es un servicio empresarial, sino infraestructura de transporte.

Pasos para implementar el patrón Service Encapsulation

Paso 3: Crear servicios encapsulados

- Encapsula cada unidad lógica como un **servicio autónomo**, con interfaz clara:
- # services/ai_service.py
- class AIService:
- def generate(self, text):
- """Genera una respuesta IA; lógica encapsulada."""
- ...
- # services/db_service.py
- class ChatStorageService:
- def save_message(self, user_id, text, sender):
- """Encapsula la persistencia de mensajes."""
- ...

Pasos para implementar el patrón Service Encapsulation

Paso 4: Definir un contrato de servicio (interfaz estable)

- Cada servicio debe tener:
- Una **interfaz documentada** (métodos, parámetros, tipos de datos).
- Un **modelo de datos canónico** común (p. ej. CanonicalMessage).
- Un formato de error estándar.
- Esto asegura **interoperabilidad** y **facilita reemplazos futuros**.

Pasos para implementar el patrón Service Encapsulation

Paso 5: Publicar o exponer el servicio

- Decide cómo será accesible:
- **Internamente**: como clase Python (dentro del proyecto).
- **Externamente**: mediante API REST o WebSocket (si lo usarán otras apps).
- Ejemplo REST:
- @app.route("/api/chat", methods=["POST"])
- def chat():
- data = request.json
- reply = chat_manager.process_message(data)
- return jsonify(reply)
- Aquí el endpoint no contiene lógica empresarial, solo **invoca servicios encapsulados**.

Pasos para implementar el patrón Service Encapsulation

Paso 6: Refinar y validar la reutilización

- Verifica que la lógica encapsulada:
- **No dependa** de controladores ni sesiones Flask.
- **Pueda ser probada unitariamente** sin el resto del sistema.
- **Pueda compartirse** con otras aplicaciones internas (microservicio o módulo Python).

Resultado

Al aplicar el patrón **Service Encapsulation**, obtienes:

1. Una **capa de servicios reutilizables** (negocio y entidad)
2. Una **capa de presentación liviana** (Flask solo enruta y coordina)
3. La posibilidad de **evolucionar hacia microservicios o APIs independientes**
4. Un entorno en el que **los servicios se comparten** dentro de tu empresa.

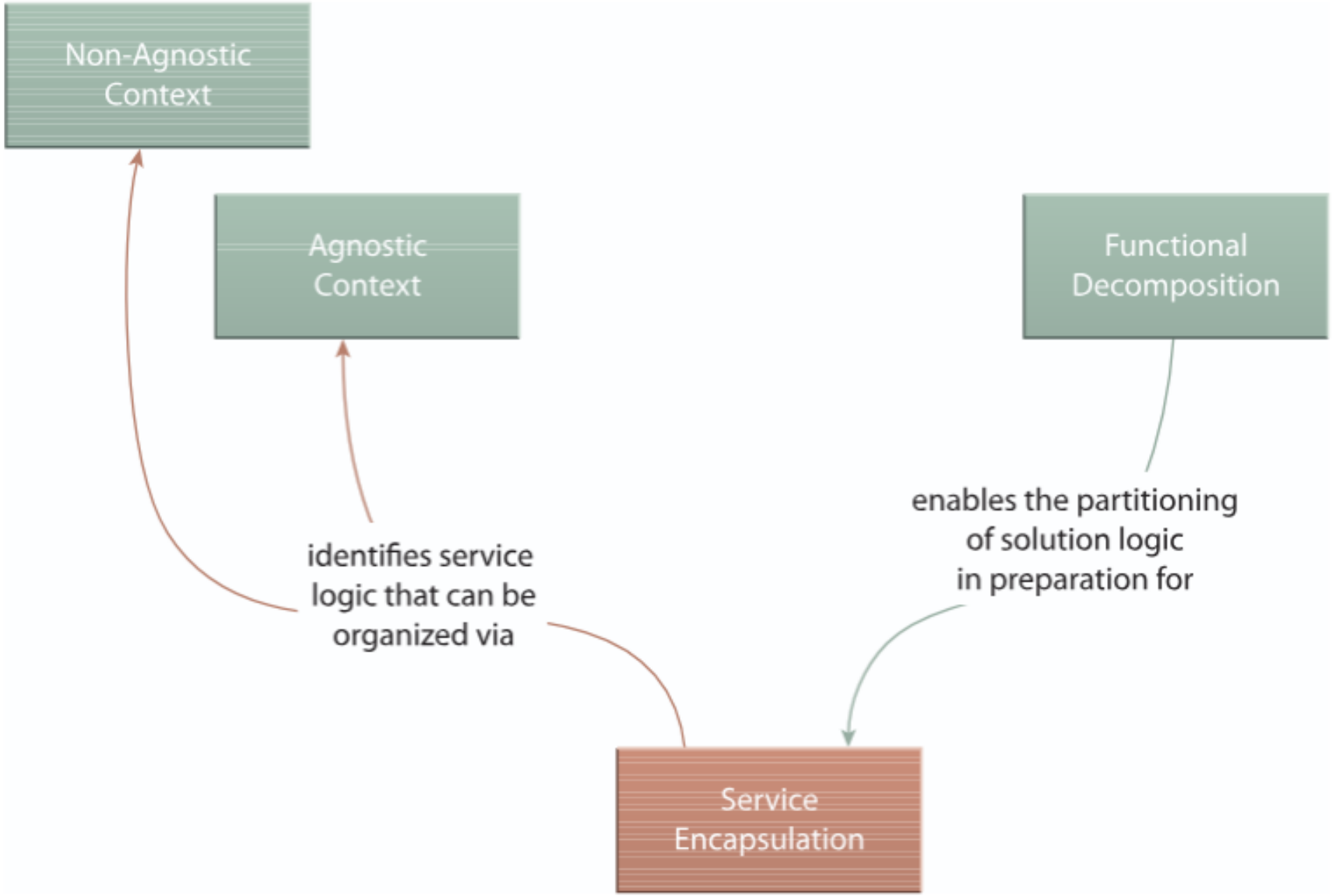


Figure 11.10

Service Encapsulation determines what logic will eventually comprise services.

Patrones de Definición de Servicios

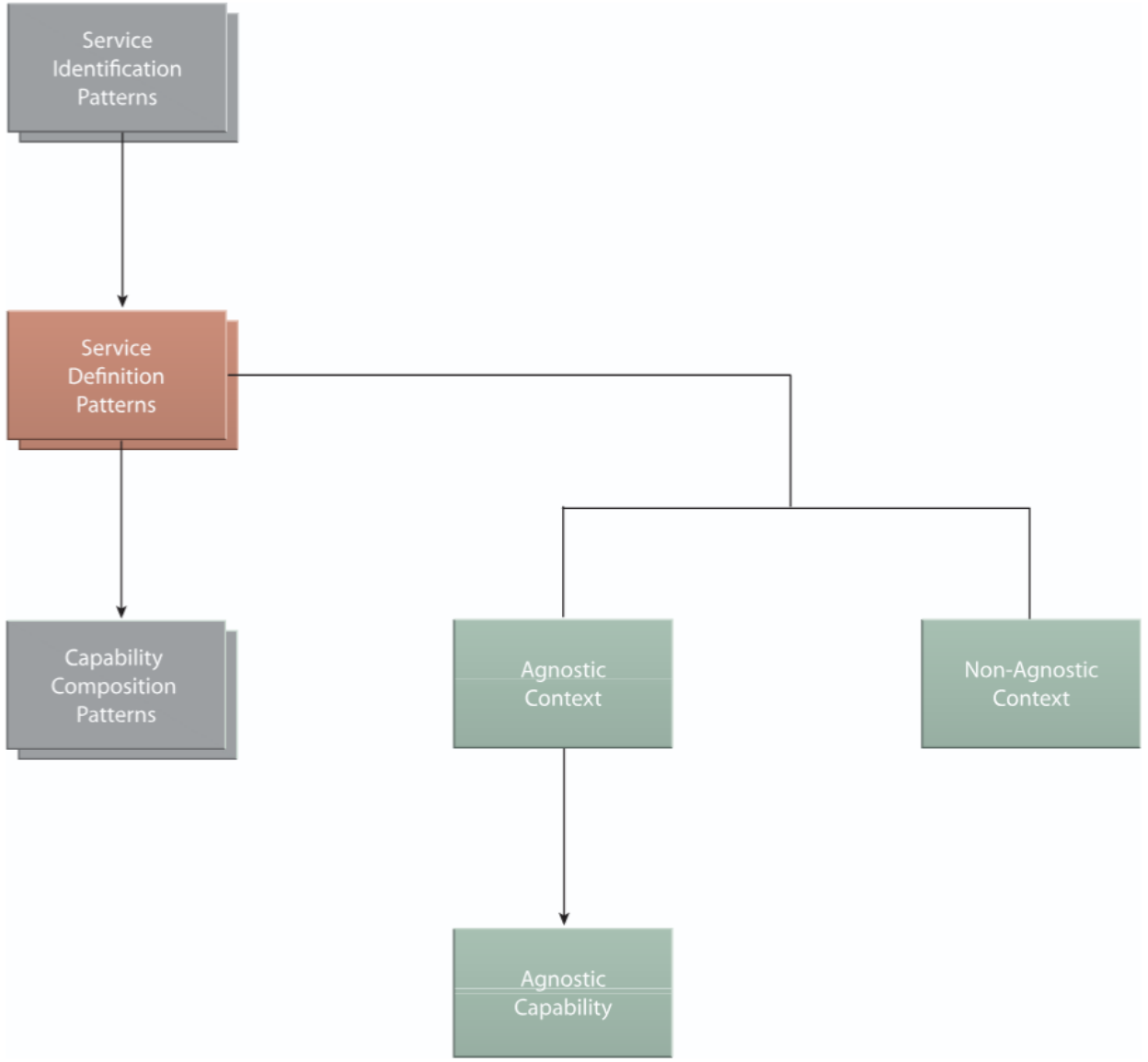


Figure 11.11
Service definition patterns organize service logic into specific contexts, thereby establishing service boundaries.

Agnostic context

Agnostic context

Este patrón propone **agrupar, refinar y reorganizar la lógica agnóstica** dentro de **contextos especializados**, basados en **modelos de servicio reutilizables**:

- **Abstracción de Entidad (Entity Abstraction)**
- **Abstracción de Utilidad (Utility Abstraction)**

Non-agnostic Context

Non-Agnostic Context

- Este patrón se refiere a contextos funcionales específicos, no reutilizables fuera del dominio inmediato.
- Es decir, servicios hechos a la medida para una funcionalidad concreta de la aplicación, aunque usen lógica agnóstica como base.

Cómo aplicar Non-Agnostic Context

Identifica las partes dependientes del frontend o framework

Estas son las que **no podrían reutilizarse fuera de la app de chat**:

- Controladores Flask (/routes/chat.py)
- Adaptadores de WebSocket (/sockets/chat_socket.py)
- Coordinador de conversación (ChatOrchestratorService)
- Serializadores / Parsers específicos de UI

Actividad U3_2

- Investigar modelos de chat en huggingface u otra pagina
- Seleccionar el que más te convenga para la aplicación.

- **Presentar prueba de ejecución**

¿Donde quedan los servicios de tarea?

Servicios que ejecutan operaciones de negocio concretas

Aunque realizan **operaciones concretas**, su **lógica de negocio debe residir en servicios agnósticos**, no en el transporte.

Los **Task Services** suelen ser **un nivel de abstracción de servicio agnóstico**, que puede combinar varios servicios de entidad o utilidad

- Task Service: "Gestión de conversaciones"
- └─ Llama a ConversationService (guardar, recuperar)
- └─ Llama a MessageService (crear mensaje)
- └─ Llama a UserService (validar usuario)

Service Layers

Service Layers

- Sin capas, cada equipo puede definir servicios arbitrarios.
- Esto genera:
  - **Redundancia funcional**: varias versiones de la misma lógica.
  - **Inconsistencias de diseño**: servicios que hacen lo mismo de forma distinta.
  - **Difícil mantenimiento y evolución** del inventario de servicios.
- **Solución:** organizar los servicios en **capas lógicas** según su propósito funcional.

Definición de capas

Capa 1: Servicios Agnósticos (Lógica de propósito múltiple)

- Contienen **lógica de negocio reusable**.
- Pueden ser invocados desde distintos controladores o composiciones.
- Ejemplos:
  - UserService → CRUD y autenticación de usuarios.
  - MessageService → creación, actualización, validación de mensajes.
  - ConversationService → persistencia y recuperación de conversaciones.
  - AIService → análisis, predicción, NLP, u otras tareas inteligentes.

Características:

- Aceptan y retornan **DTOs canónicos**.
- No dependen de HTTP, WebSocket ni infraestructura.
- Pueden participar en **Capability Composition y Recomposition**.

Definición de capas

Capa 2: Servicios No Agnósticos / de Transporte (Lógica de propósito único)

- Contienen **infraestructura específica**: transporte, serialización, protocolos.
- Ejemplos :
  - APIController → traduce JSON/HTTP ↔ DTO, expone endpoints.
  - ResponseHandler → estandariza respuestas HTTP o WS.
- No implementan reglas de negocio, solo **mapas de datos y entrega de respuesta**.

Características:

- Dependen de la infraestructura (Flask, WebSocket).
- Solo llaman a servicios agnósticos usando DTOs canónicos.
- Sirven como **puente entre cliente y servicios agnósticos**.

Definición de capas

Capa 3: (Opcional) Capas de Composición / Task Services

- Servicios que combinan capacidades agnósticas para resolver un **problema de negocio completo**.
- Ejemplos:
  - MessagingCapability → combina UserService, MessageService y ConversationService.
  - MessagingRecomposition → reutiliza capacidades para múltiples escenarios.
- Mantienen lógica de negocio coordinada, pero no dependen de la infraestructura.
- **Beneficio:** ayuda a organizar **Capability Composition y Recomposition** dentro de una estructura de capas.

Separación de responsabilidades

- Capa agnóstica → lógica de negocio.
- Capa de composición → coordinación de capacidades.
- Capa no agnóstica → transporte y mapeo de datos.

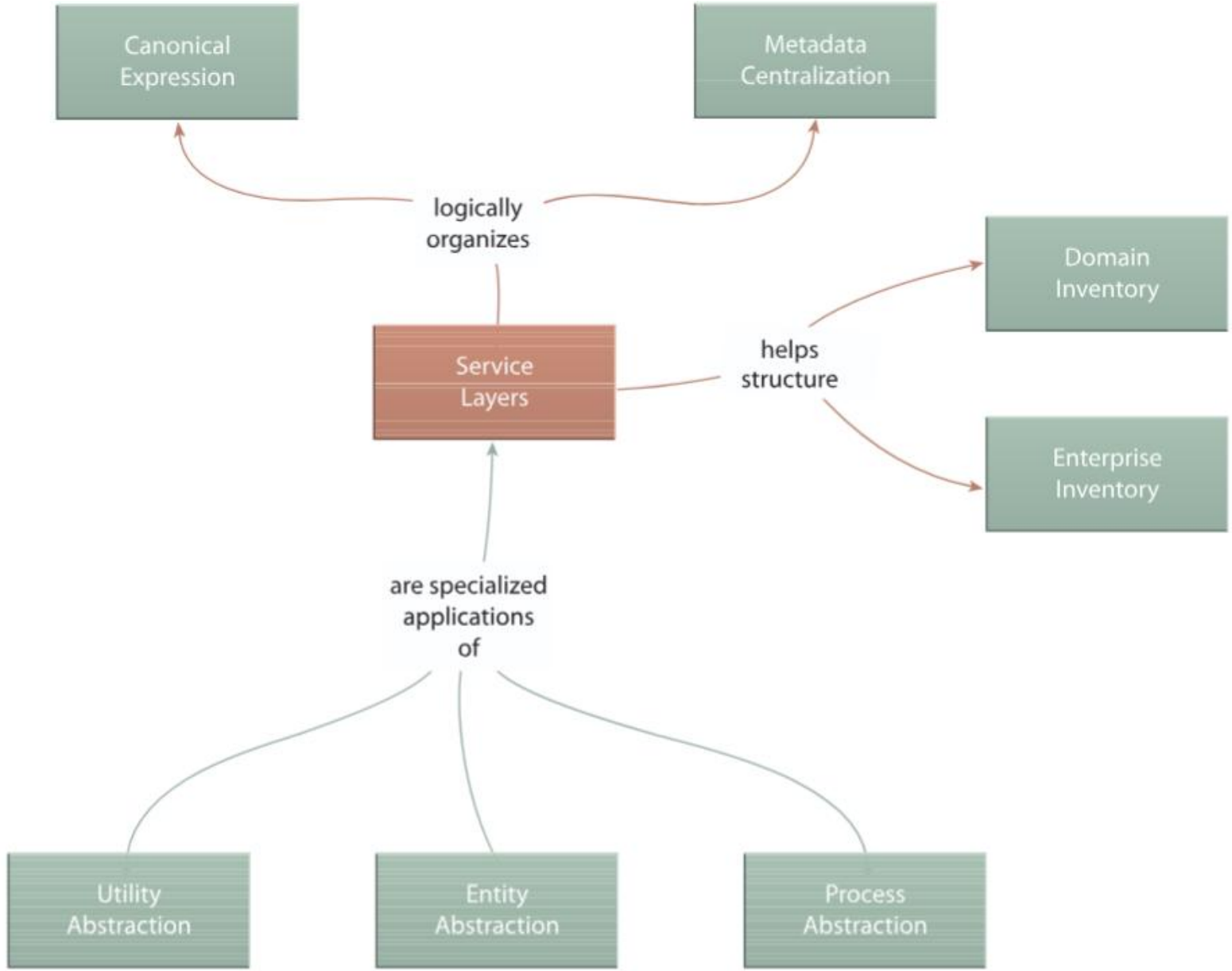


Figure 6.24

Service Layers relates to preceding and upcoming patterns by adding a logical structure to the service inventory.

Canonical Expression

Canonical Expression

Problema
Los contratos de servicio pueden expresar capacidades similares de diferentes maneras, lo que conduce a inconsistencias y aumenta el riesgo de mala interpretación.

Solución
Los contratos de servicio se estandarizan utilizando **convenciones de nomenclatura**.

Aplicación
Las convenciones de nomenclatura se aplican a los contratos de servicio como parte de los procesos formales de análisis y diseño.

Impactos
El uso de convenciones de nomenclatura globales introduce estándares a nivel empresarial que deben ser utilizados y aplicados de manera consistente.

Principios
Contrato de Servicio Estandarizado (Standardized Service Contract), Capacidad de Descubrimiento del Servicio (Service Discoverability)

1 Identificar los servicios de dominio vs servicios de transporte

- **Canonical Expression** se aplica principalmente a los servicios **agnósticos**, que representan la **lógica de negocio y entidades**, porque aquí es donde tiene sentido estandarizar **nombres y operaciones**.

Ya están clasificados los servicios:

- Agnóstico
- No Agnóstico

2 Definir convenciones de nomenclatura

- Usar **verbos estándar CRUD** para operaciones sobre entidades:
  - createUser, readUser, updateUser, deleteUser
  - saveConversation, getConversation, updateConversation, deleteConversation
  - sendMessage, getMessage, updateMessage, deleteMessage
- Para servicios que no siguen CRUD exacto, definir verbos claros y consistentes:
  - queryAIModel, analyzeWithAI, predictIntent
- Esto garantiza que cualquier consumidor del servicio (otro módulo, API, o script) tenga **expectativas claras sobre las operaciones**.

2 Definir convenciones de nomenclatura

AIService (Consultar IA, Agnóstica)

- Nombres de métodos claros y consistentes según la acción:
- queryAIModel(modelName, inputData) → consulta genérica al modelo.
- analyzeWithAI(inputData) → análisis de texto, imagen o señal.
- predictIntent(inputText) → retorna intención detectada por el modelo.

Evitamos verbos CRUD porque no estamos manipulando una entidad interna, sino que estamos llamando a un servicio externo.

2 Definir convenciones de nomenclatura

ConversationService (Guardar conversación, Agnóstica)

Aquí podemos usar CRUD :

- createConversation(conversationData) → guarda nueva conversación.
- getConversation(conversationId) → obtiene conversación por ID.
- updateConversation(conversationId, updateData) → actualiza registro.
- deleteConversation(conversationId) → elimina conversación.

Si se abstrae la persistencia con un repositorio (ConversationRepository), los métodos pueden ser idénticos.

2 Definir convenciones de nomenclatura

UserService (Entidad: Usuario, Agnóstica)

Verbos CRUD claros:

- createUser(userData) → registra un nuevo usuario.
- getUser(userId) → obtiene información del usuario.
- updateUser(userId, updateData) → actualiza datos del usuario.
- deleteUser(userId) → elimina usuario.

3 Aplicar expresiones canónicas

Cada servicio expone un contrato claro (métodos, parámetros y tipos de datos) que sea independiente del transporte:

- ConversationService.saveConversation(conversation: ConversationDTO) -> bool
- MessageService.sendMessage(message: MessageDTO) -> ResponseDTO
- UserService.authenticate(credentials: CredentialsDTO) -> UserDTO

Usar DTOs canónicos para intercambio de datos:

- Esto evita que el formato dependa de Flask, WebSocket, JSON específico o base de datos.
- Ejemplo: ConversationDTO incluye id, userId, timestamp, text de forma estándar.

¿Qué es un DTO?

- Un **DTO (Data Transfer Object)** es un **objeto plano** que sirve para **transportar datos entre capas o servicios**, **sin incluir lógica de negocio**.
- Cuando aplicas el patrón **Canonical Expression**, necesitas que **todas las capas compartan una expresión común de los datos** (un *idioma compartido*).
  Los DTOs son **la materialización concreta de ese idioma canónico**.

Funciones de un DTO

1. Normalizar la estructura de los datos (que todos los servicios los entiendan igual).
2. Proteger las entidades internas de ser expuestas directamente al cliente.
3. Aislar cambios entre capas (si cambia la base de datos, no rompes la API).
4. Facilitar interoperabilidad entre servicios (cada uno sabe cómo recibir y enviar datos).

3 Aplicar expresiones canónicas

- **Definir DTOs canónicos**
- Estos objetos representan los datos que se intercambian entre servicios, sin depender de la forma en que se envían por HTTP o WS.

```python
from dataclasses import dataclass
from datetime import datetime
from typing import List, Optional

# DTO para Usuario
@dataclass
class CredentialsDTO:
    username: str
    password: str

@dataclass
class UserDTO:
    id: str
    username: str
    email: str
    is_active: bool

# DTO para Mensaje
@dataclass
class MessageDTO:
    id: str
    conversation_id: str
    sender_id: str
    content: str
    timestamp: datetime
```

3 Aplicar expresiones canónicas

- **Definir los servicios con contratos claros**
- Los servicios solo manejan DTOs y no dependen de Flask ni WebSocket.

```python
class MessageService:
    def sendMessage(self, message: MessageDTO) -> ResponseDTO:
        #Envía un mensaje.
        # Lógica de persistencia o envío
        print(f"Enviando mensaje {message.id} en {message.conversation_id}")
        return ResponseDTO(success=True, message="Mensaje enviado",
                           data={"message_id": message.id})

    def getMessage(self, message_id: str) -> Optional[MessageDTO]:
        #Recupera un mensaje por ID.
        return None
```

4 Separación de responsabilidades

- Los servicios **no agnósticos** (APIController, ResponseHandler) **solo traducen y transportan datos**, pero no deben implementar lógica de negocio.
- Los servicios **agnósticos** aplican **Canonical Expression,** de modo que cualquier controlador REST o WebSocket pueda llamarlos sin conocer detalles internos.

4 Separación de responsabilidades

- **Regla clave:** los servicios no agnósticos nunca implementan lógica de negocio; solo transforman datos y llaman a los servicios agnósticos usando **DTOs canónicos**.

Estructura ejemplo

```
/services
    conversation_service.py  # Agnóstico
    message_service.py       # Agnóstico
    user_service.py          # Agnóstico

/controllers
    api_controller.py        # No agnóstico
    response_handler.py      # No agnóstico

/dtos
    user_dto.py
    message_dto.py
    conversation_dto.py
```

Ejemplo de servicios agnósticos

```python
# message_service.py
from dtos.message_dto import MessageDTO, ResponseDTO

class MessageService:
    def sendMessage(self, message: MessageDTO) -> ResponseDTO:
        print(f"[MessageService] Enviando mensaje {message.id}")
        return ResponseDTO(success=True, message="Mensaje enviado",
                           data={"message_id": message.id})
```

Ejemplo de servicios no agnósticos

```python
# response_handler.py
class ResponseHandler:
    @staticmethod
    def send_success(data):
        return {"success": True, "data": data}

    @staticmethod
    def send_error(message):
        return {"success": False, "error": message}
```

4 Separación de responsabilidades

Contexto	Qué hace	Ejemplo
Agnóstico	Lógica de negocio, persiste entidades, valida reglas, coordina Task Services	UserService, ConversationService.saveConversation
Non-Agnóstico	Traduce JSON/HTTP/WS ↔ DTOs, envía respuestas	APIController, ResponseHandler
DTOs Canónicos	Normalizan datos entre todos los servicios	UserDTO, MessageDTO, ConversationDTO

5 Flujo de implementación

- Definir los DTOs canónicos para todas las entidades (UserDTO, MessageDTO, ConversationDTO).
- Estandarizar métodos de los servicios con verbos CRUD u otros verbos canónicos.
- Actualizar los servicios agnósticos para aceptar y retornar únicamente DTOs canónicos.
- Actualizar los controladores y endpoints para mapear desde el protocolo (HTTP/JSON, WebSocket) a los DTOs canónicos.

5 Flujo de implementación

- Los controladores solo **traducen el protocolo al DTO** y **llaman al servicio agnóstico**, luego traducen la respuesta de nuevo al protocolo.
- Flask solo maneja la infraestructura y el transporte.
- La lógica de negocio permanece en los servicios agnósticos.
- DTOs canónicos permiten que otro controlador (WebSocket, CLI) pueda usar los mismos servicios sin cambiar nada.

5 Flujo de implementación

Flujo de ejemplo

- Cliente envía JSON a /send_message.
- APIController transforma JSON a **MessageDTO**.
- Llama a **MessageService.sendMessage(message_dto).**
- **MessageService** aplica la lógica de negocio, devuelve **ResponseDTO**.
- APIController convierte **ResponseDTO** a JSON y lo envía al cliente.

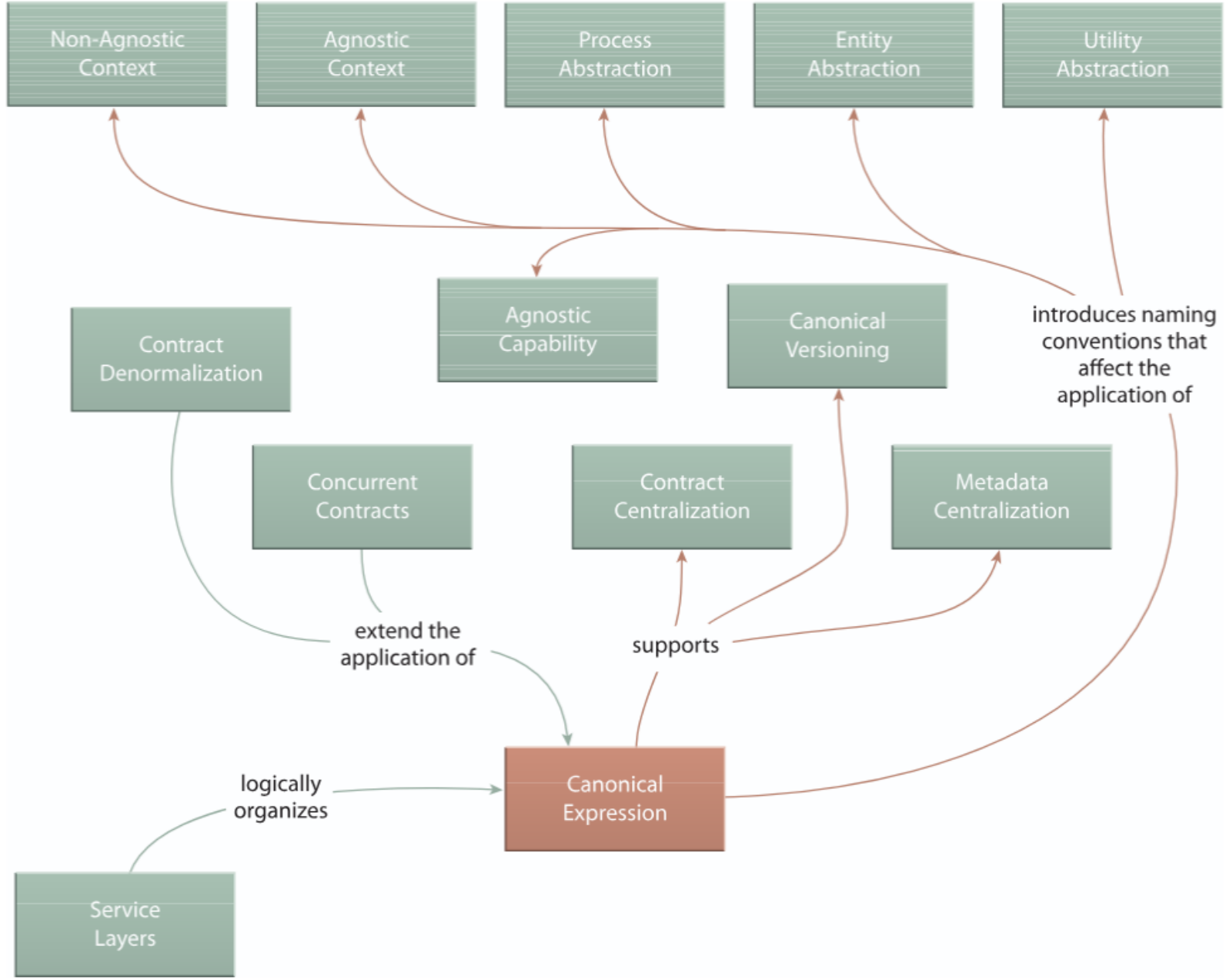
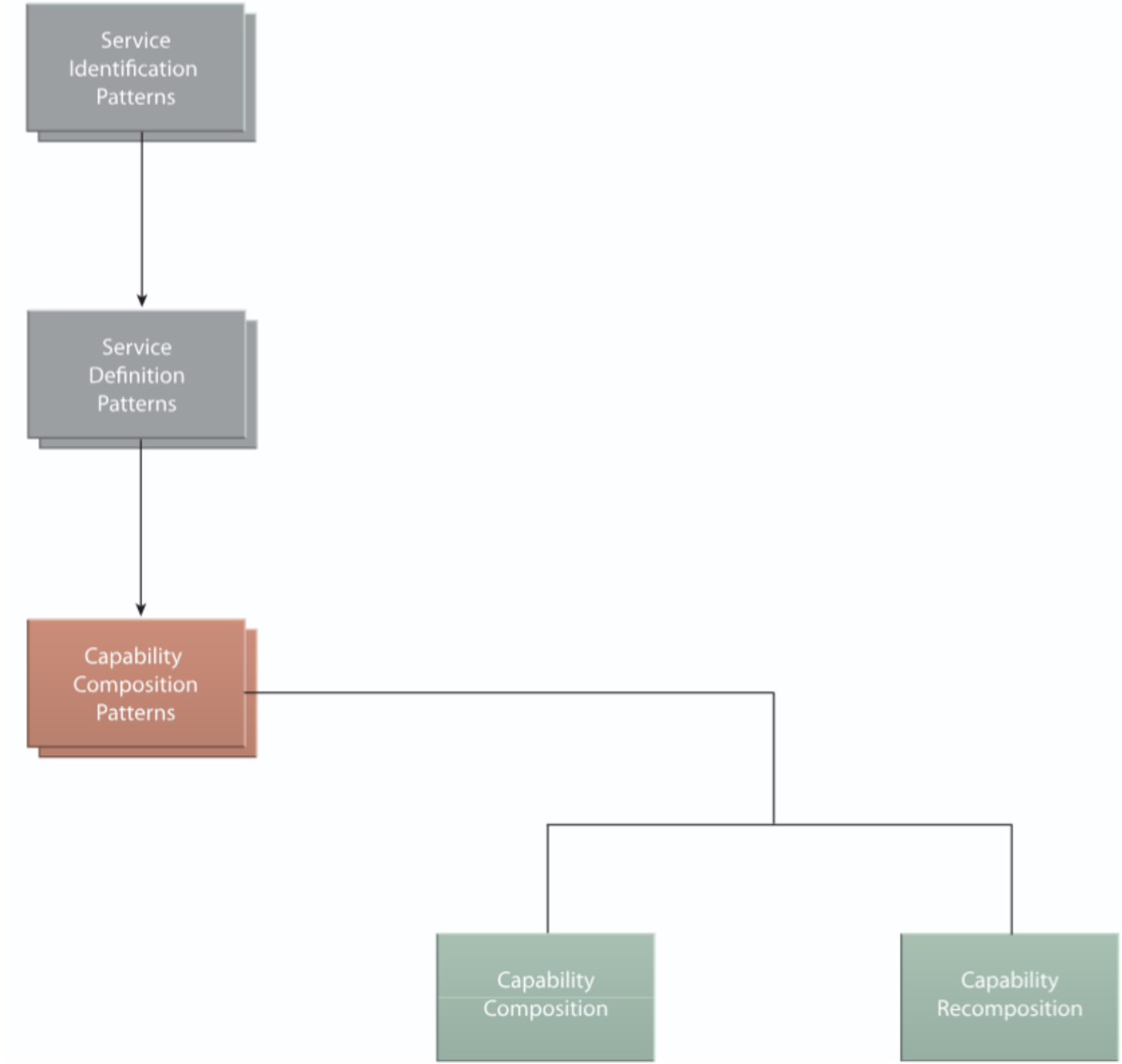


Figure 10.3

Canonical Expression keeps the external expression of service contracts consistent, thereby affecting contract and context-related patterns.


Service Identification Patterns
Service Definition Patterns
Capability Composition Patterns
Capability Composition
Capability Recomposition