Maggie AI Assistant: Abstractions and Adapters System

Table of Contents

General Overview

Purpose and Benefits

The Maggie AI Assistant's abstraction and adapter system provides a flexible, loosely coupled architecture that enhances the maintainability, testability, and extensibility of the application. This system allows core

components to interact through well-defined interfaces rather than concrete implementations, reducing direct dependencies between components.

Key benefits include:

- **Decoupling**: Components depend on abstractions rather than concrete implementations
- **Testability**: Easier to mock dependencies for unit testing
- **Extensibility**: New implementations can be introduced without changing existing code
- **Consistency**: Standardized interfaces for common capabilities
- **Flexibility**: Components can operate without knowing the specifics of their dependencies
- **Modularity**: Components can be developed, tested, and deployed independently

Design Principles

The abstraction and adapter system is built upon several key design principles:

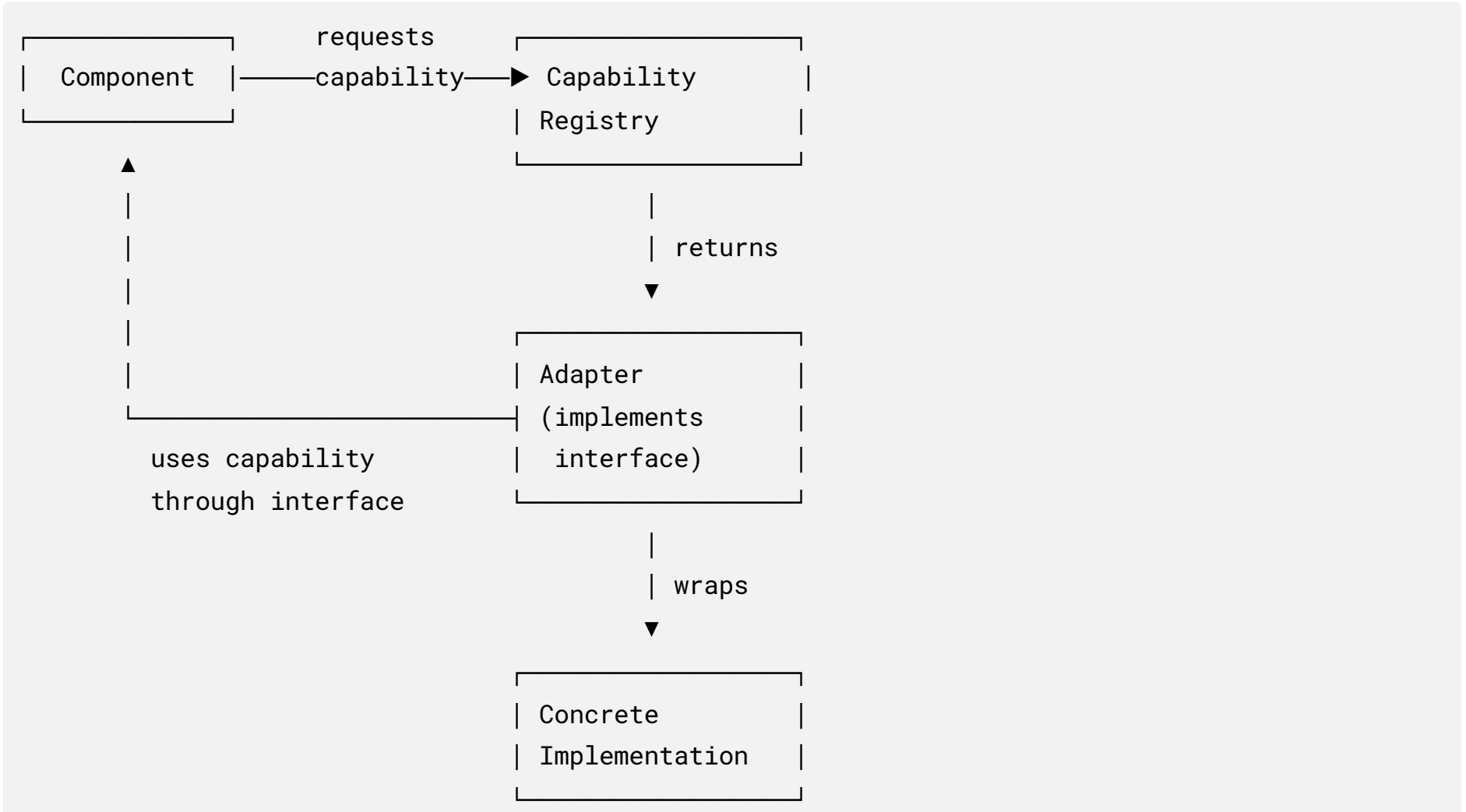
1. **Interface Segregation Principle**: Interfaces are specialized for specific capabilities
2. **Dependency Inversion Principle**: High-level modules depend on abstractions, not details
3. **Adapter Pattern**: Adapters convert existing implementations to expected interfaces
4. **Singleton Pattern**: Registry provides a single access point for capabilities
5. **Service Locator Pattern**: Components retrieve needed capabilities at runtime

System Architecture

The abstraction and adapter system consists of three main parts:

1. **Abstract Interfaces**: Define capabilities without specifying implementations
2. **Capability Registry**: Singleton registry that maintains registered implementations
3. **Adapters**: Bridge between concrete implementations and abstract interfaces

Components access capabilities through the registry, which returns the appropriate implementation wrapped in an adapter that conforms to the expected interface.



System Components

Core Abstractions

The system defines several core abstractions in `maggie/utils/abstractions.py`:

`ILoggerProvider`

Defines the interface for logging services:

```python
class ILoggerProvider(ABC):
    @abstractmethod
    def debug(self, message: str, **kwargs) -> None: pass

    @abstractmethod
    def info(self, message: str, **kwargs) -> None: pass

    @abstractmethod
    def warning(self, message: str, **kwargs) -> None: pass

    @abstractmethod
    def error(self, message: str, exception: Optional[Exception] = None, **kwargs) -> None: pass

    @abstractmethod
    def critical(self, message: str, exception: Optional[Exception] = None, **kwargs) -> None: pa
```

`IErrorHandler`

Defines the interface for error handling services:

```python
class IErrorHandler(ABC):
    @abstractmethod
    def record_error(self, message: str, exception: Optional[Exception] = None, **kwargs) -> Any:

    @abstractmethod
    def safe_execute(self, func: Callable, *args, **kwargs) -> Any: pass
```

`IEventPublisher`

Defines the interface for event publishing services:

```python
class IEventPublisher(ABC):
    @abstractmethod
    def publish(self, event_type: str, data: Any = None, **kwargs) -> None: pass
```

`IStateProvider`

Defines the interface for state management services:

```
class IStateProvider(ABC):
    @abstractmethod
    def get_current_state(self) -> Any: pass
```

Additional Enumerations:

The module also defines enumerations for categorizing log levels, error categories, and error severities:

```
class LogLevel(Enum):
    DEBUG = auto()
    INFO = auto()
    WARNING = auto()
    ERROR = auto()
    CRITICAL = auto()

class ErrorCategory(Enum):
    SYSTEM = auto()
    NETWORK = auto()
    RESOURCE = auto()
    PERMISSION = auto()
    CONFIGURATION = auto()
    INPUT = auto()
    PROCESSING = auto()
    MODEL = auto()
    EXTENSION = auto()
    STATE = auto()
    UNKNOWN = auto()

class ErrorSeverity(Enum):
    DEBUG = auto()
    INFO = auto()
    WARNING = auto()
    ERROR = auto()
    CRITICAL = auto()
```

Capability Registry

The `CapabilityRegistry` is a singleton class that manages the registration and retrieval of capability implementations:

```
class CapabilityRegistry:
    _instance = None
    _lock = threading.RLock()

    @classmethod
    def get_instance(cls):
        if cls._instance is None:
            with cls._lock:
                if cls._instance is None:
                    cls._instance = CapabilityRegistry()
        return cls._instance

    def __init__(self):
        self._registry = {}
```

```
    def register(self, capability_type: Type, instance: Any) -> None:
        self._registry[capability_type] = instance

    def get(self, capability_type: Type) -> Optional[Any]:
        return self._registry.get(capability_type)
```

The module also provides convenient functions to retrieve specific capabilities:

```
def get_logger_provider() -> Optional[ILoggerProvider]:
    return CapabilityRegistry.get_instance().get(ILoggerProvider)

def get_error_handler() -> Optional[IErrorHandler]:
    return CapabilityRegistry.get_instance().get(IErrorHandler)

def get_event_publisher() -> Optional[IEventPublisher]:
    return CapabilityRegistry.get_instance().get(IEventPublisher)

def get_state_provider() -> Optional[IStateProvider]:
    return CapabilityRegistry.get_instance().get(IStateProvider)
```

Adapter Implementations

The adapter implementations in `maggie/utils/adapters.py` bridge the gap between concrete implementations and the abstract interfaces:

`LoggingManagerAdapter`

Adapts the `LoggingManager` to the `ILoggerProvider` interface:

```
class LoggingManagerAdapter(ILoggerProvider):
    def __init__(self, logging_manager):
        self.logging_manager = logging_manager
        registry = CapabilityRegistry.get_instance()
        registry.register(ILoggerProvider, self)

    def debug(self, message: str, **kwargs) -> None:
        from maggie.utils.logging import LogLevel
        self.logging_manager.log(LogLevel.DEBUG, message, **kwargs)

    # ... implementations for other methods ...
```

`ErrorHandlerAdapter`

Adapts error handling functions to the `IErrorHandler` interface:

```
class ErrorHandlerAdapter(IErrorHandler):
    def __init__(self):
        registry = CapabilityRegistry.get_instance()
        registry.register(IErrorHandler, self)

    def record_error(self, message: str, exception: Optional[Exception] = None, **kwargs) -> Any:
```

```
        from maggie.utils.error_handling import record_error as do_record_error, ErrorCategory, E
        # ... implementation ...

    def safe_execute(self, func: Callable, *args, **kwargs) -> Any:
        from maggie.utils.error_handling import safe_execute as do_safe_execute, ErrorCategory, E
        # ... implementation ...
```

`EventBusAdapter`

Adapts the `EventBus` to the `IEventPublisher` interface:

```
class EventBusAdapter(IEventPublisher):
    def __init__(self, event_bus):
        self.event_bus = event_bus
        registry = CapabilityRegistry.get_instance()
        registry.register(IEventPublisher, self)

    def publish(self, event_type: str, data: Any = None, **kwargs) -> None:
        priority = kwargs.pop('priority', None)
        if priority is not None:
            self.event_bus.publish(event_type, data, priority)
        else:
            self.event_bus.publish(event_type, data)
```

`StateManagerAdapter`

Adapts the `StateManager` to the `IStateProvider` interface:

```
class StateManagerAdapter(IStateProvider):
    def __init__(self, state_manager):
        self.state_manager = state_manager
        registry = CapabilityRegistry.get_instance()
        registry.register(IStateProvider, self)

    def get_current_state(self) -> Any:
        return self.state_manager.get_current_state()
```

Implementation Guide

Setting Up the Capability Registry

The capability registry is automatically initialized when first accessed. Typically, this initialization happens during application startup in `main.py` through the `initialize_components` function:

```
def initialize_components(config: Dict[str, Any], debug: bool = False) -> Dict[str, Any]:
    components = {}

    # Initialize registry
    registry = CapabilityRegistry.get_instance()
    components['registry'] = registry
```

```
# Initialize logging manager
from maggie.utils.logging import LoggingManager
logging_mgr = LoggingManager.initialize(config)
components['logging_manager'] = logging_mgr

# Create and register adapters
error_handler = ErrorHandlerAdapter()
components['error_handler'] = error_handler

from maggie.core.event import EventBus
event_bus = EventBus()
components['event_bus'] = event_bus

from maggie.core.state import StateManager, State
state_manager = StateManager(State.INIT, event_bus)
components['state_manager'] = state_manager

# Create and register adapters
logging_adapter = LoggingManagerAdapter(logging_mgr)
components['logging_adapter'] = logging_adapter

event_bus_adapter = EventBusAdapter(event_bus)
components['event_bus_adapter'] = event_bus_adapter

state_manager_adapter = StateManagerAdapter(state_manager)
components['state_manager_adapter'] = state_manager_adapter

# Enhance logging manager with event publishing and state information
logging_mgr.enhance_with_event_publisher(event_bus_adapter)
logging_mgr.enhance_with_state_provider(state_manager_adapter)

# Return initialized components
return components
```

Implementing Concrete Providers

To implement a concrete provider for one of the abstract interfaces:

1. Create a class that implements the required interface methods
2. Instantiate the class
3. Create an appropriate adapter for the class
4. Register the adapter with the registry

For example, to implement a custom logger provider:

```
from maggie.utils.abstractions import ILoggerProvider

class CustomLoggerProvider:
    def __init__(self, config):
        self.config = config
        # Initialize the logger with configuration

    def log(self, level, message, *args, **kwargs):
        # Implement logging functionality
        pass
```

```
# Then create an adapter for this custom implementation
class CustomLoggerAdapter(ILoggerProvider):
    def __init__(self, custom_logger):
        self.logger = custom_logger
        registry = CapabilityRegistry.get_instance()
        registry.register(ILoggerProvider, self)

    def debug(self, message: str, **kwargs) -> None:
        self.logger.log('DEBUG', message, **kwargs)

    def info(self, message: str, **kwargs) -> None:
        self.logger.log('INFO', message, **kwargs)

    # ... other method implementations ...

# Usage:
custom_logger = CustomLoggerProvider(config)
custom_adapter = CustomLoggerAdapter(custom_logger)
# Now the custom logger is available through the registry
```

Adapting Services with Adapters

Adapters bridge the gap between concrete implementations and the abstract interfaces expected by the system. An adapter:

1. Implements the required abstract interface
2. Delegates calls to the concrete implementation
3. Translates between the interface expected by the abstract system and the interface provided by the concrete implementation
4. Registers itself with the capability registry

When creating an adapter:

```
class MyServiceAdapter(IRequiredInterface):
    def __init__(self, concrete_service):
        self.service = concrete_service
        # Register with the registry
        registry = CapabilityRegistry.get_instance()
        registry.register(IRequiredInterface, self)

    # Implement the methods required by the interface
    def required_method(self, *args, **kwargs):
        # Translate to the concrete implementation's method
        return self.service.concrete_method(*args, **kwargs)
```

Accessing Capabilities Through Abstractions

Components can access capabilities through the helper functions or directly through the registry:

```
from maggie.utils.abstractions import get_logger_provider, get_error_handler

class MyComponent:
    def __init__(self):
```

```
    # Get capabilities through helper functions
    self.logger_provider = get_logger_provider()
    self.error_handler = get_error_handler()

    # Or directly through the registry
    from maggie.utils.abstractions import CapabilityRegistry, IEventPublisher
    registry = CapabilityRegistry.get_instance()
    self.event_publisher = registry.get(IEventPublisher)

def do_something(self):
    # Use the capabilities
    self.logger_provider.info("Doing something")

    def risky_operation():
        # Implementation that might raise exceptions
        pass

    result = self.error_handler.safe_execute(
        risky_operation,
        error_code="OPERATION_FAILED",
        default_return=None
    )

    # Publish an event
    self.event_publisher.publish("operation_completed", {"result": result})
```

The system is designed to be robust in the face of missing capabilities. Always check for `None` when retrieving capabilities that might not be available:

```
logger = get_logger_provider()
if logger:
    logger.info("System is running")
else:
    print("Logger not available, using fallback")
```

Implementation Examples

Example 1: Basic Registration and Retrieval

This example demonstrates basic registration and retrieval of capabilities:

```
from maggie.utils.abstractions import (
    ILoggerProvider,
    CapabilityRegistry,
    get_logger_provider
)

# Create a simple logger implementation
class SimpleLogger(ILoggerProvider):
    def debug(self, message: str, **kwargs) -> None:
        print(f"DEBUG: {message}")

    def info(self, message: str, **kwargs) -> None:
```

```
        print(f"INFO: {message}")

    def warning(self, message: str, **kwargs) -> None:
        print(f"WARNING: {message}")

    def error(self, message: str, exception: Optional[Exception] = None, **kwargs) -> None:
        error_msg = f"ERROR: {message}"
        if exception:
            error_msg += f" - Exception: {str(exception)}"
        print(error_msg)

    def critical(self, message: str, exception: Optional[Exception] = None, **kwargs) -> None:
        error_msg = f"CRITICAL: {message}"
        if exception:
            error_msg += f" - Exception: {str(exception)}"
        print(error_msg)

# Register the implementation
registry = CapabilityRegistry.get_instance()
simple_logger = SimpleLogger()
registry.register(ILoggerProvider, simple_logger)

# Retrieve and use the logger
logger = get_logger_provider()
if logger:
    logger.info("System initialized")
    logger.debug("Debug information")

    try:
        1 / 0
    except Exception as e:
        logger.error("An error occurred", exception=e)
```

Example 2: Component Using Multiple Capabilities

This example shows a component that uses multiple capabilities through abstractions:

```
from maggie.utils.abstractions import (
    get_logger_provider,
    get_error_handler,
    get_event_publisher,
    get_state_provider
)

class DataProcessor:
    """A component that uses multiple system capabilities through abstractions."""

    def __init__(self):
        # Get required capabilities
        self.logger = get_logger_provider()
        self.error_handler = get_error_handler()
        self.event_publisher = get_event_publisher()
        self.state_provider = get_state_provider()

        if not all([self.logger, self.error_handler, self.event_publisher, self.state_provider]):
            raise RuntimeError("Required capabilities not available")
```

```python
    def process_data(self, data):
        """Process data with error handling and event publishing."""
        # Log the operation
        self.logger.info(f"Processing data: {len(data)} items")

        # Check if we're in the right state
        current_state = self.state_provider.get_current_state()
        from maggie.core.state import State
        if current_state not in [State.ACTIVE, State.BUSY]:
            self.logger.warning(f"Processing data in inappropriate state: {current_state.name}")
            self.event_publisher.publish("invalid_operation", {
                "operation": "process_data",
                "state": current_state.name,
                "required_states": ["ACTIVE", "BUSY"]
            })
            return False

        # Process data with error handling
        def _process_items():
            results = []
            for item in data:
                # Processing logic
                processed = self._process_item(item)
                results.append(processed)
            return results

        results = self.error_handler.safe_execute(
            _process_items,
            error_code="DATA_PROCESSING_ERROR",
            default_return=[],
            error_details={"data_size": len(data)}
        )

        # Publish completion event
        self.event_publisher.publish("data_processed", {
            "input_size": len(data),
            "output_size": len(results),
            "success": len(results) > 0
        })

        return results

    def _process_item(self, item):
        # Implementation of item processing
        return {"processed": item}
```

Example 3: Creating Custom Adapters

This example demonstrates creating a custom adapter for a third-party service:

```python
from maggie.utils.abstractions import ILoggerProvider, CapabilityRegistry
import logging

class ThirdPartyLoggerAdapter(ILoggerProvider):
    """Adapter for a third-party logging library to conform to Maggie's ILoggerProvider interface

    def __init__(self, logger_name="maggie", log_level=logging.INFO):
```

```
        """
        Initialize the adapter with a standard Python logger.

        Parameters
        ----------
        logger_name : str
            Name for the logger
        log_level : int
            Logging level (from Python's logging module)
        """
        # Create a standard Python logger
        self.logger = logging.getLogger(logger_name)
        self.logger.setLevel(log_level)

        # Add console handler if none exists
        if not self.logger.handlers:
            handler = logging.StreamHandler()
            formatter = logging.Formatter(
                '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
            )
            handler.setFormatter(formatter)
            self.logger.addHandler(handler)

        # Register with the registry
        registry = CapabilityRegistry.get_instance()
        registry.register(ILoggerProvider, self)

    def debug(self, message: str, **kwargs) -> None:
        """Log a debug message."""
        extra = self._prepare_extra(kwargs)
        self.logger.debug(message, extra=extra)

    def info(self, message: str, **kwargs) -> None:
        """Log an info message."""
        extra = self._prepare_extra(kwargs)
        self.logger.info(message, extra=extra)

    def warning(self, message: str, **kwargs) -> None:
        """Log a warning message."""
        extra = self._prepare_extra(kwargs)
        self.logger.warning(message, extra=extra)

    def error(self, message: str, exception: Optional[Exception] = None, **kwargs) -> None:
        """Log an error message."""
        extra = self._prepare_extra(kwargs)
        self.logger.error(message, exc_info=exception, extra=extra)

    def critical(self, message: str, exception: Optional[Exception] = None, **kwargs) -> None:
        """Log a critical message."""
        extra = self._prepare_extra(kwargs)
        self.logger.critical(message, exc_info=exception, extra=extra)

    def _prepare_extra(self, kwargs):
        """Prepare extra context for the logger."""
        # Filter out keys that might conflict with logging's built-in keys
        safe_keys = {k: v for k, v in kwargs.items()
                     if k not in ('exc_info', 'stack_info', 'stacklevel')}
        return {'extra_context': safe_keys} if safe_keys else None
```

```
# Usage example:
adapter = ThirdPartyLoggerAdapter(logger_name="custom_logger", log_level=logging.DEBUG)

# Now the standard Python logger is available through Maggie's abstraction
from maggie.utils.abstractions import get_logger_provider
logger = get_logger_provider()
logger.info("Message using the abstraction", user_id=123, source="login_service")
```

Example 4: Complete Integration Example

This example shows a more complete integration of the abstraction and adapter system:

```
from maggie.utils.abstractions import (
    ILoggerProvider,
    IErrorHandler,
    IEventPublisher,
    IStateProvider,
    CapabilityRegistry,
    get_logger_provider,
    get_error_handler,
    get_event_publisher,
    get_state_provider
)
from maggie.utils.adapters import (
    LoggingManagerAdapter,
    ErrorHandlerAdapter,
    EventBusAdapter,
    StateManagerAdapter
)
from maggie.utils.logging import LoggingManager
from maggie.core.event import EventBus
from maggie.core.state import StateManager, State

def initialize_system(config):
    """Initialize the system with all adapters and capabilities."""

    # Initialize the core components
    logging_manager = LoggingManager.initialize(config)
    event_bus = EventBus()
    state_manager = StateManager(State.INIT, event_bus)

    # Create and register adapters
    logging_adapter = LoggingManagerAdapter(logging_manager)
    error_handler = ErrorHandlerAdapter()
    event_bus_adapter = EventBusAdapter(event_bus)
    state_manager_adapter = StateManagerAdapter(state_manager)

    # Enhance components with cross-cutting capabilities
    logging_manager.enhance_with_event_publisher(event_bus_adapter)
    logging_manager.enhance_with_state_provider(state_manager_adapter)

    # Start the event bus
    event_bus.start()

    # Return initialized components
    return {
        'logging_manager': logging_manager,
```

```
        'event_bus': event_bus,
        'state_manager': state_manager
    }

class ServiceComponent:
    """Example component that uses system capabilities."""

    def __init__(self):
        """Initialize the component with required capabilities."""
        self.logger = get_logger_provider()
        self.error_handler = get_error_handler()
        self.event_publisher = get_event_publisher()
        self.state_provider = get_state_provider()

        if not all([self.logger, self.error_handler, self.event_publisher, self.state_provider]):
            if self.logger:
                self.logger.critical("Required capabilities not available")
            raise RuntimeError("Required capabilities not available")

        self.logger.info("ServiceComponent initialized")

    def start_service(self):
        """Start the service with state transitions."""
        current_state = self.state_provider.get_current_state()

        # Use the logger
        self.logger.info(f"Starting service in state: {current_state.name}")

        # Publish an event
        self.event_publisher.publish("service_starting", {
            "timestamp": time.time(),
            "initial_state": current_state.name
        })

        # Execute an operation with error handling
        def setup_operation():
            # Setup operations
            self.logger.debug("Performing setup operations")
            # Simulate work
            time.sleep(1)
            return True

        success = self.error_handler.safe_execute(
            setup_operation,
            error_code="SERVICE_SETUP_ERROR",
            default_return=False
        )

        if success:
            self.logger.info("Service started successfully")
            self.event_publisher.publish("service_started")
        else:
            self.logger.error("Failed to start service")
            self.event_publisher.publish("service_start_failed")

        return success

# Usage in main application:
if __name__ == "__main__":
```

```
import time

# Configuration for the system
config = {
    'logging': {
        'path': 'logs',
        'console_level': 'INFO',
        'file_level': 'DEBUG'
    }
}

# Initialize the system
components = initialize_system(config)

# Get the state manager from the returned components
state_manager = components['state_manager']

# Start system
state_manager.transition_to(State.STARTUP, "application_start")
```

```
# Create and use a component
try:
    service = ServiceComponent()
    state_manager.transition_to(State.READY, "components_initialized")

    if service.start_service():
        state_manager.transition_to(State.ACTIVE, "service_running")
    else:
        state_manager.transition_to(State.CLEANUP, "service_start_failed")
except Exception as e:
    logger = get_logger_provider()
    if logger:
        logger.critical("Failed to initialize or start service", exception=e)
    state_manager.transition_to(State.CLEANUP, "error_in_startup")

# Cleanup and shutdown
state_manager.transition_to(State.SHUTDOWN, "application_exit")

# Stop the event bus
components['event_bus'].stop()
```

Best Practices

Dependency Management

1. **Favor Composition Over Inheritance**: Use the capability registry to compose functionality rather than inheriting from multiple base classes.
2. **Late Binding**: Components should retrieve capabilities when needed, not just at initialization.
3. **Graceful Degradation**: Always check if a capability is available before using it, and provide fallback behavior if possible.
4. **Single Responsibility**: Each adapter should adapt exactly one implementation to one interface.

5. **Dependency Documentation**: Document which capabilities a component requires to function correctly.

Example:

```
class RobustComponent:
    """
    A component that gracefully handles missing capabilities.

    Required Capabilities:
    - ILoggerProvider: For logging operations
    - IErrorHandler: For handling errors

    Optional Capabilities:
    - IEventPublisher: For publishing events (falls back to direct method calls)
    - IStateProvider: For checking application state (ignores state if unavailable)
    """

    def __init__(self):
        # Get required capabilities
        self.logger = get_logger_provider()
        self.error_handler = get_error_handler()

        # These are optional
        self.event_publisher = get_event_publisher()
        self.state_provider = get_state_provider()

        # Fallbacks for missing capabilities
        if not self.logger:
            print("WARNING: Logger not available, using print statements")

        if not self.error_handler:
            print("WARNING: Error handler not available, using basic exception handling")

    def log_message(self, level, message):
        """Log a message with fallback to print statements."""
        if self.logger:
            if level == "debug":
                self.logger.debug(message)
            elif level == "info":
                self.logger.info(message)
            # ... and so on
        else:
            print(f"{level.upper()}: {message}")

    def publish_event(self, event_type, data=None):
        """Publish an event with fallback to direct notification."""
        if self.event_publisher:
            self.event_publisher.publish(event_type, data)
        else:
            # Fallback behavior
            self.log_message("info", f"Would publish event: {event_type}")
            # Maybe call a direct method on a dependent component
```

Testing with Abstractions

The abstraction system makes testing much easier by allowing dependencies to be mocked or stubbed:

```
import unittest
from unittest.mock import MagicMock

from maggie.utils.abstractions import (
    ILoggerProvider,
    IErrorHandler,
    CapabilityRegistry
)

class TestComponent:
    def __init__(self):
        # Get capabilities from registry
        registry = CapabilityRegistry.get_instance()
        self.logger = registry.get(ILoggerProvider)
        self.error_handler = registry.get(IErrorHandler)

    def do_operation(self, value):
        if not value:
            self.logger.error("Value cannot be empty")
            return False

        try:
            result = self._process(value)
            self.logger.info(f"Processed value: {result}")
            return result
        except Exception as e:
            self.logger.error("Processing failed", exception=e)
            return False

    def _process(self, value):
        # Some processing that might fail
        return value * 2

class TestComponentTests(unittest.TestCase):
    def setUp(self):
        # Create mock implementations of the required capabilities
        self.mock_logger = MagicMock(spec=ILoggerProvider)
        self.mock_error_handler = MagicMock(spec=IErrorHandler)

        # Register the mocks with the registry
        registry = CapabilityRegistry.get_instance()
        registry.register(ILoggerProvider, self.mock_logger)
        registry.register(IErrorHandler, self.mock_error_handler)

        # Create the component under test
        self.component = TestComponent()

    def test_do_operation_with_valid_value(self):
        # Test with a valid value
        result = self.component.do_operation(5)

        # Verify the result
        self.assertEqual(result, 10)

        # Verify the logger was called with the expected message
        self.mock_logger.info.assert_called_once()
        self.assertIn("Processed value: 10", self.mock_logger.info.call_args[0][0])

    def test_do_operation_with_empty_value(self):
```

```
        # Test with an empty value
        result = self.component.do_operation(0)

        # Verify the result
        self.assertFalse(result)

        # Verify the logger was called with the expected error message
        self.mock_logger.error.assert_called_once_with("Value cannot be empty")

    def tearDown(self):
        # Clean up the registry
        registry = CapabilityRegistry.get_instance()
        registry._registry.clear()
```

Error Handling

When working with abstractions, always consider how to handle missing or misbehaving capabilities:

1. **Check for None**: Always check if a capability is None before using it.
2. **Use Safe Execution**: Wrap calls to capabilities in safe_execute when appropriate.
3. **Plan for Failures**: Design your component to work (perhaps in a degraded mode) when capabilities are missing.
4. **Log Issues**: Log when capabilities are missing or errors occur using them.

Example of robust error handling:

```
from maggie.utils.abstractions import get_logger_provider, get_error_handler

class RobustComponent:
    def __init__(self):
        self.logger = get_logger_provider()
        self.error_handler = get_error_handler()

    def perform_operation(self, data):
        # Log operation start
        if self.logger:
            self.logger.info(f"Starting operation with {len(data)} items")

        # Use error handler if available, otherwise use basic try/except
        if self.error_handler:
            result = self.error_handler.safe_execute(
                self._internal_operation,
                data,
                error_code="OPERATION_FAILED",
                default_return=None
            )
        else:
            try:
                result = self._internal_operation(data)
            except Exception as e:
                if self.logger:
                    self.logger.error("Operation failed", exception=e)
                result = None
```

```
        # Log completion
        if self.logger:
            if result:
                self.logger.info("Operation completed successfully")
            else:
                self.logger.warning("Operation failed or returned no result")

        return result

    def _internal_operation(self, data):
        # Implementation that might raise exceptions
        return [item * 2 for item in data]
```

Performance Considerations

While abstractions provide flexibility, they can introduce overhead. Here are some tips for maintaining performance:

1. **Avoid Frequent Registry Lookups**: Get capabilities once and store references rather than looking them up repeatedly.

2. **Balance Abstraction and Performance**: Consider direct dependencies for performance-critical paths where the flexibility of abstractions is less important.

3. **Batch Operations**: When using logging or event capabilities, batch operations where possible to reduce overhead.

4. **Conditional Logging**: Use conditional checks before constructing expensive log messages.

Example of performance-optimized code:

```
from maggie.utils.abstractions import get_logger_provider

class PerformanceAwareComponent:
    def __init__(self):
        # Get the logger once
        self.logger = get_logger_provider()
        self.is_debug_enabled = self.logger and hasattr(self.logger, 'is_debug_enabled') and self

    def process_batch(self, items):
        batch_size = len(items)

        # Only log if necessary
        if self.logger:
            self.logger.info(f"Processing batch of {batch_size} items")

        results = []
        for i, item in enumerate(items):
            result = self.process_item(item)
            results.append(result)

            # Log progress sparingly
            if self.is_debug_enabled and (i + 1) % 100 == 0:
                self.logger.debug(f"Processed {i + 1}/{batch_size} items")
```

```
        # Log completion
        if self.logger:
            self.logger.info(f"Completed processing {batch_size} items")

        return results

    def process_item(self, item):
        # Processing logic
        return item * 2
```

Reference Materials

API Reference

Abstractions Module

Interface Classes:

- `ILoggerProvider`: Interface for logging services
- `IErrorHandler`: Interface for error handling services
- `IEventPublisher`: Interface for event publishing services
- `IStateProvider`: Interface for state management services

Enumerations:

- `LogLevel`: Enumeration of log levels (DEBUG, INFO, WARNING, ERROR, CRITICAL)
- `ErrorCategory`: Enumeration of error categories (SYSTEM, NETWORK, etc.)
- `ErrorSeverity`: Enumeration of error severities (DEBUG, INFO, etc.)

Capability Registry:

- `CapabilityRegistry`: Singleton registry for capability implementations
  - `get_instance()`: Get the singleton instance
  - `register(capability_type, instance)`: Register an implementation
  - `get(capability_type)`: Get an implementation

Helper Functions:

- `get_logger_provider()`: Get the registered logger provider
- `get_error_handler()`: Get the registered error handler
- `get_event_publisher()`: Get the registered event publisher
- `get_state_provider()`: Get the registered state provider

Adapters Module

Adapter Classes:

- `LoggingManagerAdapter`: Adapts LoggingManager to ILoggerProvider
- `ErrorHandlerAdapter`: Adapts error handling functions to IErrorHandler
- `EventBusAdapter`: Adapts EventBus to IEventPublisher
- `StateManagerAdapter`: Adapts StateManager to IStateProvider

Related Design Patterns

The abstraction and adapter system implements several design patterns:

1. **Adapter Pattern**: Converts the interface of a class into another interface clients expect.
2. **Singleton Pattern**: Ensures a class has only one instance and provides a global point of access to it.
3. **Service Locator Pattern**: Encapsulates the processes involved in obtaining a service with a strong abstraction layer.
4. **Dependency Injection**: A technique in which an object receives other objects it depends on.
5. **Interface Segregation Principle**: No client should be forced to depend on methods it does not use.
6. **Dependency Inversion Principle**: High-level modules should not depend on low-level modules. Both should depend on abstractions.

For more information on these patterns, refer to:

- “Design Patterns: Elements of Reusable Object-Oriented Software” by Gamma, Helm, Johnson, and Vlissides
- “Clean Architecture: A Craftsman’s Guide to Software Structure and Design” by Robert C. Martin
- “Patterns of Enterprise Application Architecture” by Martin Fowler