Service Locator Technical Manual

1. General Overview

The `ServiceLocator` class in the Maggie AI Assistant framework implements the Service Locator pattern, providing a centralized registry that manages service dependencies throughout the application. As a core architectural component, it functions as a sophisticated dependency injection mechanism that facilitates loose coupling between components that require services and their concrete implementations.

Key Features

- **Centralized Service Registration**: Provides a single point of registration for all services
- **State-Aware Service Access**: Controls service availability based on application state
- **Transition-Aware Service Access**: Supports services that are only available during specific state transitions
- **Type-Safe Service Retrieval**: Offers type-checked access to registered services
- **Lazy Service Creation**: Supports on-demand service instantiation with factory methods
- **Service Availability Queries**: Allows components to check if a service is available

Architectural Benefits

- **Reduced Coupling**: Components depend on the ServiceLocator rather than concrete service implementations
- **Runtime Flexibility**: Services can be registered, replaced, or modified during application execution
- **Enhanced Testability**: Facilitates service mocking and substitution during unit testing
- **State Management Integration**: Seamlessly coordinates with the application's state management system
- **Resource Optimization**: Services can be made available only in states where they're needed

Core Implementation

The `ServiceLocator` is implemented as a class with static methods and class variables, following a singleton-like approach where all service registrations are stored in class-level dictionaries. This design ensures that all components access the same registry instance throughout the application lifecycle.

```
class ServiceLocator:
    _services: Dict[str, Any] = {}
    _state_constraints: Dict[str, Set] = {}
    _transition_constraints: Dict[str, List[Tuple]] = {}
    _current_state: Optional[Any] = None
    _last_transition: Optional[Tuple] = None
    _logger = ComponentLogger('ServiceLocator')
```

2. Implementation Details

2.1 ServiceLocator Class Structure

The `ServiceLocator` is implemented as a static utility class with class methods for registration and retrieval operations. All state is maintained at the class level, making it accessible globally throughout the application.

Key Internal Data Structures:

- `_services`: Dictionary mapping service names to service instances
- `_state_constraints`: Dictionary mapping service names to sets of states in which the service is available
- `_transition_constraints`: Dictionary mapping service names to lists of state transitions during which the service is available
- `_current_state`: Reference to the current application state
- `_last_transition`: Tuple containing the (from_state, to_state) of the last transition

Registration Methods:

```python
@classmethod
def register(cls, name: str, service: Any, available_states: Optional[List] = None) -> None:
    """
    Register a service with optional state constraints.

    Parameters
    ----------
    name : str
        Name of the service to register
    service : Any
        Service instance to register
    available_states : Optional[List]
        List of states in which the service is available, if None, available in all states
    """
    cls._services[name] = service
    if available_states:
        cls._state_constraints[name] = set(available_states)
    cls._logger.info(f"Registered service: {name}")
```

```python
@classmethod
def register_for_transition(cls, name: str, service: Any, transitions: List[Tuple]) -> None:
    """
    Register a service that's only available during specific state transitions.

    Parameters
    ----------
    name : str
        Name of the service to register
    service : Any
        Service instance to register
    transitions : List[Tuple]
        List of (from_state, to_state) tuples representing transitions
        during which the service is available
    """
    cls._services[name] = service
    cls._transition_constraints[name] = transitions
    cls._logger.info(f"Registered transition-specific service: {name}")
```

Retrieval Methods:

```
@classmethod
@with_error_handling(error_category=ErrorCategory.SYSTEM)
def get(cls, name: str) -> Optional[Any]:
    """
    Get a service by name, respecting state and transition constraints.

    Parameters
    ----------
    name : str
        Name of the service to retrieve

    Returns
    -------
    Optional[Any]
        Service instance if found and available, None otherwise
    """
    service = cls._services.get(name)
    if service is None:
        cls._logger.warning(f"Service not found: {name}")
        return None

    # Check state constraints
    if cls._current_state and name in cls._state_constraints:
        from maggie.core.state import State
        allowed_states = cls._state_constraints.get(name, set())
        if cls._current_state not in allowed_states:
            cls._logger.warning(f"Service {name} not available in current state {cls._current_sta
            return None

    # Check transition constraints
    if cls._last_transition and name in cls._transition_constraints:
        allowed_transitions = cls._transition_constraints.get(name, [])
        if cls._last_transition not in allowed_transitions:
            from_state_name = cls._last_transition[0].name if cls._last_transition[0] else 'None'
            to_state_name = cls._last_transition[1].name
            cls._logger.warning(f"Service {name} not available for transition {from_state_name} -
            return None

    return service
```

Type-Safe Retrieval:

```
@classmethod
def get_typed(cls, name: str, service_type: Type[T]) -> Optional[T]:
    """
    Get a service with type checking.

    Parameters
    ----------
    name : str
        Name of the service to retrieve
    service_type : Type[T]
        Expected type of the service

    Returns
    -------
    Optional[T]
```

```
        Service instance if found, available, and of correct type, None otherwise
    """
    service = cls.get(name)
    if service is None:
        return None
    if not isinstance(service, service_type):
        cls._logger.error(f"Service type mismatch: {name} is {type(service).__name__}, expected {
        return None
    return cast(T, service)
```

State Tracking:

```
@classmethod
def update_state(cls, new_state) -> None:
    """
    Update the current application state.

    Parameters
    ----------
    new_state
        New application state
    """
    if cls._current_state != new_state:
        cls._last_transition = (cls._current_state, new_state) if cls._current_state else None
        cls._current_state = new_state
        cls._logger.debug(f"Updated service locator state to {new_state.name}")
```

2.2 State Integration

The `ServiceLocator` integrates with the application's state management system through the `update_state` method, which is called whenever the application state changes. This allows the ServiceLocator to track the current state and enforce state-based service availability constraints.

2.3 Error Handling

The `ServiceLocator` uses a decorator-based error handling system to ensure robust operation. The `with_error_handling` decorator catches and logs exceptions, preventing service retrieval failures from crashing the application.

3. Implementing the ServiceLocator System

3.1 Core Integration

To implement the ServiceLocator in your application:

1. **Initialize Core Services**: Register essential services during application startup
2. **Connect to State Management**: Set up state tracking with the ServiceLocator
3. **Register Component Services**: Register services provided by application components
4. **Retrieve Services**: Use ServiceLocator throughout the application to access services

3.2 Step-by-Step Implementation

Step 1: Register Core Services

During application initialization, register core services that other components will depend on:

```python
def _register_core_services(self) -> bool:
    try:
        from maggie.service.locator import ServiceLocator

        # Register core services
        ServiceLocator.register('event_bus', self.event_bus)
        ServiceLocator.register('state_manager', self.state_manager)
        ServiceLocator.register('maggie_ai', self)
        ServiceLocator.register('config_manager', self.config_manager)

        self.logger.debug('Core services registered with ServiceLocator')
        return True
    except ImportError as e:
        self.logger.error(f"Failed to import ServiceLocator: {e}")
        return False
    except Exception as e:
        self.logger.error(f"Error registering core services: {e}")
        return False
```

Step 2: Connect to State Management

Ensure the ServiceLocator is updated when the application state changes:

```python
def _on_state_changed(self, transition) -> None:
    try:
        # Update ServiceLocator with new state
        from maggie.service.locator import ServiceLocator
        ServiceLocator.update_state(transition.to_state)

        # Handle the state change in this component
        # ...
    except Exception as e:
        self.logger.error(f"Error handling state change: {e}")
```

Step 3: Register Component Services

When initializing components, register their services:

```python
def _initialize_processors(self) -> bool:
    try:
        # Initialize LLM processor
        from maggie.service.llm.processor import LLMProcessor
        from maggie.service.locator import ServiceLocator

        llm_config = self.config.get('llm', {})
        self.llm_processor = LLMProcessor(llm_config)

        # Register with ServiceLocator
        ServiceLocator.register('llm_processor', self.llm_processor)

        self.logger.debug('LLM processor initialized and registered')
        return True
    except Exception as e:
        self.logger.error(f"Error initializing LLM processor: {e}")
        return False
```

Step 4: Retrieve Services in Components

Use ServiceLocator to access services throughout the application:

```
def process_text(self, text: str) -> str:
    try:
        # Get required services
        from maggie.service.locator import ServiceLocator

        llm_processor = ServiceLocator.get('llm_processor')
        if not llm_processor:
            self.logger.error("LLM processor service not available")
            return "Sorry, text processing is not available right now."

        # Process the text
        result = llm_processor.generate_text(text)
        return result
    except Exception as e:
        self.logger.error(f"Error processing text: {e}")
        return "An error occurred during text processing."
```

3.3 Advanced Integration Techniques

State-Aware Service Registration

Register services that should only be available in specific application states:

```
# Register TTS processor for ACTIVE and READY states
from maggie.core.state import State
ServiceLocator.register('tts_processor', self.tts_processor,
                        available_states=[State.ACTIVE, State.READY])
```

Transition-Aware Service Registration

Register services that should only be available during specific state transitions:

```
# Register a service that's only available during the transition from LOADING to ACTIVE
ServiceLocator.register_for_transition('initialization_helper',
                                       initialization_service,
                                       transitions=[(State.LOADING, State.ACTIVE)])
```

Lazy Service Creation

Use the `get_or_create` method to create services on demand:

```python
def get_visualization_service(self) -> VisualizationService:
    from maggie.service.locator import ServiceLocator

    # This will create the service only if it doesn't exist yet
    return ServiceLocator.get_or_create(
        'visualization_service',
        factory=lambda: VisualizationService(self.config.get('visualization', {})),
        available_states=[State.ACTIVE, State.BUSY]
    )
```

4. Usage Examples

4.1 Basic Service Registration and Retrieval

```python
# --- Registration ---
from maggie.service.locator import ServiceLocator

# Register a simple service
config_manager = ConfigManager('config.yaml')
ServiceLocator.register('config_manager', config_manager)

# --- Retrieval ---
# Get the service later in another component
config_manager = ServiceLocator.get('config_manager')
if config_manager:
    # Use the config manager
    app_config = config_manager.config
    log_level = app_config.get('logging', {}).get('level', 'INFO')
else:
    # Handle missing service
    log_level = 'INFO'  # default fallback
```

4.2 State-Aware Service Usage

```python
# --- Registration ---
from maggie.core.state import State
from maggie.service.locator import ServiceLocator

# Register LLM processor with state constraints
llm_processor = LLMProcessor(config.get('llm', {}))
ServiceLocator.register('llm_processor', llm_processor,
                        available_states=[State.READY, State.ACTIVE, State.BUSY])

# --- State Update ---
# When state changes (e.g., in state manager)
def transition_to(self, new_state: State, trigger: str) -> bool:
    # State transition logic...

    # Update ServiceLocator state
    ServiceLocator.update_state(new_state)
    # More transition logic...

# --- Retrieval ---
# Get the LLM processor in a component
llm_processor = ServiceLocator.get('llm_processor')
if llm_processor:
    # LLM processor is available in current state
    result = llm_processor.generate_text(prompt)
else:
    # LLM processor not available in current state
    result = "Service not available in current state"
```

4.3 Type-Safe Service Retrieval

```python
from maggie.service.locator import ServiceLocator
from maggie.service.tts.processor import TTSProcessor

# Get TTS processor with type checking
tts_processor = ServiceLocator.get_typed('tts_processor', TTSProcessor)
if tts_processor:
    # We have a properly typed TTSProcessor instance
    tts_processor.speak("Hello, world!")
else:
    # Service not found or not of correct type
    print("TTS service not available or not of correct type")
```

4.4 Lazy Service Creation

```python
from maggie.service.locator import ServiceLocator
from maggie.utils.visualization import VisualizationService

# Get or create visualization service
viz_service = ServiceLocator.get_or_create(
    'visualization_service',
    factory=lambda: VisualizationService(config.get('visualization', {})),
    available_states=[State.ACTIVE, State.BUSY]
)

# Use the service
viz_service.create_chart(data)
```

4.5 Checking Service Availability

```python
from maggie.service.locator import ServiceLocator
from maggie.core.state import State

# Check what services are available in ACTIVE state
available_services = ServiceLocator.get_available_services(State.ACTIVE)
print(f"Services available in ACTIVE state: {available_services}")
```

```
# Check if a specific service is available
if ServiceLocator.has_service('llm_processor'):
    print("LLM processor service is registered")

    # Check if it's available in current state
    llm_processor = ServiceLocator.get('llm_processor')
    if llm_processor:
        print("LLM processor is available in current state")
    else:
        print("LLM processor is registered but not available in current state")
else:
    print("LLM processor service is not registered")
```

5. Advanced Topics

5.1 Service Lifecycle Management

The ServiceLocator pattern in the Maggie AI framework does not directly manage service lifecycles (creation and destruction). However, you can implement lifecycle management using the following strategies:

- **Registration Timing**: Register services only when they are needed and ready to use
- **Service Factories**: Use factory functions to create services on demand
- **State-Based Availability**: Make services available only in appropriate states
- **Manual Cleanup**: Clear the registry when services are no longer needed

Example of manual service registry cleanup:

```
def cleanup_services(self):
    """
    Clear all registered services during application shutdown.
    """
    from maggie.service.locator import ServiceLocator
    ServiceLocator.clear()
    self.logger.info("Service registry cleared")
```

5.2 Service Dependencies

When services depend on other services, you can use the ServiceLocator to resolve dependencies:

```
class LLMProcessor:
    def __init__(self, config: Dict[str, Any]):
        self.config = config

        # Resolve dependencies through ServiceLocator
        self.state_manager = ServiceLocator.get('state_manager')
        if self.state_manager:
            StateAwareComponent.__init__(self, self.state_manager)

        self.event_bus = ServiceLocator.get('event_bus')
        if self.event_bus:
            EventListener.__init__(self, self.event_bus)
    # More initialization...
```

5.3 Testing with ServiceLocator

The ServiceLocator pattern can make unit testing challenging due to global state. Here are some approaches to testing with ServiceLocator:

Clear Registry Between Tests

```
def tearDown(self):
    # Clear ServiceLocator registry after each test
    from maggie.service.locator import ServiceLocator
    ServiceLocator.clear()
```

Register Mock Services

```
def setUp(self):
    # Set up mock services for testing
    from maggie.service.locator import ServiceLocator
    from unittest.mock import MagicMock

    # Create and register mock services
    self.mock_llm = MagicMock()
    self.mock_llm.generate_text.return_value = "Mock response"
    ServiceLocator.register('llm_processor', self.mock_llm)

    # Set up state for state-aware services
    from maggie.core.state import State
    ServiceLocator.update_state(State.ACTIVE)
```

Test Component with Mocked ServiceLocator

```
@patch('maggie.service.locator.ServiceLocator.get')
def test_process_command(self, mock_get):
    # Set up mock return values for ServiceLocator.get
    mock_llm = MagicMock()
    mock_llm.generate_text.return_value = "Mock response"

    # Configure mock to return our mock service
    mock_get.side_effect = lambda name: mock_llm if name == 'llm_processor' else None

    # Test the component that uses ServiceLocator
    result = self.component_under_test.process_command("Test command")

    # Verify expectations
    self.assertEqual(result, "Mock response")
    mock_llm.generate_text.assert_called_once()
```

6. Best Practices and Guidelines

6.1 When to Use ServiceLocator

The ServiceLocator pattern is most appropriate when:

- The application needs runtime flexibility in service configurations
- Components need to be decoupled from their dependencies
- Services have different lifecycles or availability requirements
- State-dependent service availability is required
- You need centralized service management

6.2 ServiceLocator Best Practices

When implementing and using the ServiceLocator pattern:

1. **Register Early**: Register core services during application initialization
2. **Be Explicit**: Clearly document which services a component needs
3. **Handle Missing Services**: Always check if a retrieved service is `None`
4. **Use Type Safety**: Use `get_typed` to ensure type compatibility
5. **Maintain Single Responsibility**: The ServiceLocator should only locate services, not create them
6. **Minimize Global State**: Keep service registration focused on true cross-cutting concerns
7. **Document Service Contracts**: Clearly define what each registered service provides

6.3 Avoiding Common Pitfalls

Hidden Dependencies

Problem: Dependencies are hidden inside components, making it hard to understand requirements.

Solution: Document required services in component docstrings.

```python
class STTProcessor:
    """
    Speech-to-Text processor component.

    Required services:
    - 'state_manager': For state-aware behavior
    - 'event_bus': For publishing transcription events
    - 'resource_manager': For managing GPU resources (optional)
    """
```

Service Availability Assumptions

Problem: Assuming services are always available can lead to runtime errors.

Solution: Always check service availability and handle missing services gracefully.

```python
def process_audio(self, audio_data: bytes) -> str:
    # Get required services
    resource_manager = ServiceLocator.get('resource_manager')
    stt_processor = ServiceLocator.get('stt_processor')

    # Check service availability
    if not stt_processor:
        self.logger.error("STT processor not available")
        return "Speech recognition not available"

    # Use optional services with fallback behavior
    if resource_manager:
        resource_manager.optimize_for_audio_processing()

    # Process the audio
    return stt_processor.recognize_speech(audio_data)
```

Circular Dependencies

Problem: Services depending on each other can create initialization loops.

Solution: Use lazy initialization or dependency injection for circular dependencies.

```python
def initialize(self):
    # Resolve circular dependency by fetching service on demand
    # rather than storing it as an instance variable in __init__
    def get_llm():
        return ServiceLocator.get('llm_processor')

    self._get_llm = get_llm
```

7. Conclusion

The ServiceLocator pattern as implemented in the Maggie AI framework provides a flexible, state-aware approach to service management. By centralizing service registration and retrieval, it facilitates component decoupling while ensuring services are only available in appropriate application states.

The state-aware ServiceLocator implementation goes beyond the basic pattern by integrating with the application's state management system, enabling fine-grained control over service availability based on application state.

When implementing your own components within the Maggie AI framework, following the established patterns for service registration and retrieval will ensure seamless integration with the existing architecture.