The Theoretical Foundations and Design of `memimg`

Abstract

`memimg` is a JavaScript library that provides transparent, event-sourced persistence for in-memory object graphs. It is a sophisticated, dynamic re-interpretation of the classical **Memory Image** (or **System Prevalence**) architectural pattern. Its core innovation is the use of **Mutation Sourcing** over traditional Command Sourcing, achieved via JavaScript's metaprogramming (`Proxy`) features. This approach creates a fully transparent "Imperative Shell" over a deterministic "Functional Core," allowing developers to mutate objects naturally while the system automatically logs low-level state changes. This design dramatically simplifies development, enhances flexibility, and provides a more robust solution to schema evolution compared to classical implementations. This document details the theoretical foundations of this approach and maps them to the design decisions in the TypeScript implementation.

Part I: The Theoretical Foundations of `memimg`

This section outlines the core theory, situating `memimg` within established architectural patterns and detailing its innovative departures.

1. The Classical Pattern: Memory Image and Command Sourcing

The architectural pattern known as Memory Image (Martin Fowler) or System Prevalence (Prevayler) provides a high-performance alternative to traditional database-centric architectures. Its principles, synthesized from sources like Fowler, `kmemimg`, and Prevayler, are as follows:

- **In-Memory State**: The entire application's object model (the "prevalent system") is held live in main memory. This eliminates database round-trips and object-relational mapping, leading to significant performance gains and simpler, richer domain models.
- **Durability via Logging**: To survive crashes, every operation that changes the system's state is first recorded in a persistent, append-only log.
- **Command Sourcing**: The unit of logging is a **`Command`** (or `Transaction`). This is a serializable object, explicitly

written by the developer, that encapsulates a single, logical state change (e.g., `new CreateAccount(id, name)`). The log is a history of these domain-specific commands.

- **Recovery via Replay**: On startup, the system state is rebuilt by creating a new, empty object model and re-executing every `Command` from the log in its original order.
- **Snapshots**: To avoid replaying an entire (potentially massive) log, the system periodically saves a complete snapshot of the in-memory state. Recovery then involves loading the latest snapshot and replaying only the commands that occurred since.

This classical pattern is powerful but imposes a significant ceremony on the developer, who must define and manage a rigid hierarchy of `Command` objects.

2. `memimg`'s Core Thesis: A Dynamic Re-interpretation

`memimg` embraces the core benefits of the Memory Image pattern but rejects its traditional implementation. Its central thesis is:

> By logging low-level **mutations** instead of high-level **commands**, and by leveraging dynamic proxies to do so transparently, we can achieve the full benefits of in-memory persistence while freeing the developer from the rigidity of Command Sourcing.

This leads to a system that is more flexible, easier to evolve, and more natural to use within a dynamic language environment like JavaScript.

3. Key Theoretical Pillar: Mutation Sourcing vs. Command Sourcing

This is the most significant departure `memimg` makes from the classical pattern.

- **Classical Command Sourcing**: The developer's *intent* is serialized.
  - **Process**: A developer must define a class like `DepositCommand`, instantiate it with parameters (`new DepositCommand(...)`), and submit it to the persistence engine.
  - **Ceremony**: This requires significant upfront design of a command hierarchy that mirrors the business domain.
- **`memimg`'s Mutation Sourcing**: The *effect* of the developer's action is serialized.
  - **Process**: A developer writes a standard, imperative mutation: `account.balance += 100;`.
  - **Transparency**: The `memimg` proxy shell intercepts this action. It does not understand the "deposit" concept. It only observes that a `SET` operation was applied to the `balance` property of a specific object. It is this generic, low-level mutation event that is generated and logged.

This theoretical shift provides two transformative advantages:

1. **Simplified Development & Exploration**: The cognitive load is near zero. The developer works with plain objects, arrays, and maps. There is no persistence-related API to learn or command hierarchy to maintain. This enables an agile, exploratory programming style where the data model can be changed freely without having to update a parallel command structure.
2. **Simplified Schema Evolution**: This is the crucial long-term benefit. In a Command Sourced system, changing a command's signature (e.g., adding a `currency` parameter to `DepositCommand`) creates a versioning problem. The event log now contains legacy commands, and the system must be explicitly taught how to handle or migrate them. This is a complex and error-prone task.

   With Mutation Sourcing, this problem is largely eliminated. The event log contains a history of fundamental state changes (`SET`, `DELETE`, `ARRAY_PUSH`), not a history of code execution. You can freely refactor your application's business logic, rename methods, or change behavioral patterns. As long as the final object *shape* remains compatible with the logged mutations, the history remains valid. The persistence layer is resilient to changes in the application layer.

4. Key Architectural Pillar: The Imperative Shell over a Functional Core

The `memimg` architecture is best understood as two distinct layers, as articulated in the `kmemimg` reference.

- **The Functional Core**: This is the pure, conceptual foundation. It holds that `State = fold(events)`. The event log is the single source of truth, and the system state is a deterministic, reproducible projection of that log.
- **The Imperative Shell**: This is the pragmatic, "magic" layer that makes the functional core usable. It allows developers to write simple, imperative code while the shell translates these actions into the events that feed the functional core. This shell is built entirely out of JavaScript's `Proxy` objects, making the underlying functional reality completely transparent.

5. Transactional Integrity via Delta Layering

To support the "draft state" workflows common in interactive applications, `memimg` adds a transactional layer on top of the core engine. This is a classic delta-tracking architecture.

- **Three-Tiered State**:
  1. **Base State**: The last known-good state, loaded from a snapshot and/or the event log. It is considered immutable within the transaction.
  2. **Delta Layer**: An in-memory `Map` of pending changes, where keys are object paths and values are the new data.
  3. **Merged View**: A special "transactional proxy" presents a virtual, unified view of the state by first checking the delta layer for a value and falling back to the base state if not found.
- **`save()` (Commit)**: Applies the changes from the delta to the base state. It is only at this moment that the real, persistent mutation events are generated and logged.

- **`discard()` (Rollback)**: A computationally cheap operation that simply throws the delta away, leaving the base state untouched.

Part II: Design and Implementation Analysis

This section maps the theoretical principles from Part I to the concrete design decisions in the `memimg` TypeScript codebase, focusing on the *why* behind the implementation.

`memimg.ts` *(The Public API & Orchestrator)*

- **Design**: A minimal, clean API surface (`createMemoryImage`, `serialize...`, `replay...`).
- **Decision**: The use of a global `WeakMap` (`memimgRegistry`) to associate root proxies with their internal tracking infrastructure (`ProxyInfrastructure`) is a deliberate design choice. The alternative would be to attach metadata directly to the user's object (e.g., `root.__memimg = ...`), which would pollute the object and violate the principle of transparency. The `WeakMap` provides this association externally and, crucially, prevents memory leaks by not holding a strong reference to the root object.

`proxy.ts` *&* `collection-wrapper.ts` *(The Imperative Shell)*

- **Design**: This is the implementation of the Imperative Shell, where the core "magic" happens.
- **Decision (`wrapIfNeeded`)**: The "wrap-before-recurse" strategy is the chosen solution to the circular reference problem. By immediately creating and registering a proxy for an object *before* traversing its children, the system can handle graphs of any complexity without infinite loops. When a circular reference is encountered, the already-registered proxy is simply returned.
- **Decision (`get` trap)**: The design of the `get` trap with a dual responsibility is a powerful pattern that fully leverages JavaScript's dynamic nature.
  1. **JIT Proxification**: It wraps nested objects and collections "just-in-time" as they are accessed. This is more efficient than a full upfront traversal and makes the system feel instantaneous.
  2. **Method Interception**: It intercepts access to collection methods (`push`, `set`, etc.) and returns a wrapped function that bundles the original behavior with event logging. This elegantly solves the problem of capturing mutations within built-in collection objects.
- **Decision (`collection-wrapper.ts`)**: The logic for wrapping Array, Map, and Set methods was refactored into a single, data-driven function. This is a design choice for maintainability and adherence to the DRY (Don't Repeat Yourself) principle, as proven by its focused unit tests.

`event-handlers.ts` (The Event Registry)

- **Design**: Implements the **Registry** and **Strategy** design patterns for handling the 18 different mutation types.
- **Decision**: This design explicitly avoids a monolithic `switch` statement for event creation and application. By giving each event type its own handler class (`SetEventHandler`, `ArrayPushHandler`, etc.), the logic for each mutation is cleanly decoupled. This makes the system easily extensible (adding a new event type just requires a new handler) and independently testable, as demonstrated by the exhaustive tests in `event-handlers.test.ts`.

`serialize.ts` & `deserialize.ts` (State Marshalling)

- **Design**: Implements the logic for creating snapshots and handling object identity across the serialization boundary.
- **Decision (`CycleTracker` strategy)**: The use of a strategy pattern with two implementations (`SnapshotCycleTracker`, `EventCycleTracker`) is a deliberate design choice. It allows the same core serialization engine to be used for two different contexts—a full snapshot versus a single event value—without duplicating the complex type-handling logic. The former tracks all objects seen during the serialization, while the latter intelligently creates references only for objects that already exist *outside* the value being serialized.
- **Decision (Two-Pass Deserialization)**: The design of deserialization—first building the object structure with placeholders for references, then resolving them in a second pass—is a standard and robust computer science solution to the problem of handling forward references in a serialized graph.
- **Decision (Date Object Handling)**: The specific logic to serialize and deserialize `Date` objects while preserving any custom properties attached to them is a direct result of iterative design and testing. The `date-object-properties.test.ts` file was created specifically to prove the existence of a bug in the initial, simpler implementation and to validate its fix, demonstrating the design's evolution in response to identified limitations.

`transaction.ts`, `delta-manager.ts`, etc. (The Transactional Layer)

- **Design**: A layered architecture implementing the delta-tracking pattern.
- **Decision**: The choice of delta tracking over a full copy-on-write for transactions is a **performance-oriented decision**. It is optimized for the common case in interactive applications: many small, localized mutations. A full copy would be prohibitively expensive.
- **Decision**: The decomposition of the transaction logic into multiple files (`delta-manager.ts`, `proxy-unwrapper.ts`, `transaction-proxy.ts`) is a design choice for **separation of concerns**. This allows each component—delta management, proxy unwrapping, and the transactional proxy's traps—to be reasoned about and tested in isolation, as shown by the focused unit tests for each module.

`event-log.ts` *(Pluggable Persistence)*

- **Design**: An interface-based, pluggable architecture.
- **Decision**: This design completely **decouples the core engine from any specific storage technology**. By defining a simple `EventLog` interface (`append`, `getAll`), `memimg` can be used in Node.js (file system), browsers (IndexedDB, LocalStorage), or with a custom remote backend without changing the core logic. The use of **NDJSON** (Newline Delimited JSON) for the file-based log is a specific choice favoring streamability and append-only resilience.

The Test Suite (`test/memimg/`)

- **Design**: A comprehensive, multi-tiered testing strategy (unit, integration, browser).
- **Decision**: The test suite's structure is a reflection of the system's design.
  - **Unit Tests**: Each module's design decisions (like the `EventHandlerRegistry` or the `DeltaManager`) are validated in isolation.
  - **Integration Tests**: These tests validate the *theoretical concepts*. For example, `circular-references.test.ts` confirms the cycle-handling theory, `persistence.test.ts` confirms the snapshot-and-replay theory, and `transaction.test.ts` confirms the transactional isolation theory.
  - **Browser Tests**: The existence of a separate Playwright suite (`event-log-browser.spec.ts`) is a design choice to guarantee that the `EventLog` interface holds true across different JavaScript runtime environments.

This holistic view of the code and tests reveals a system where every implementation detail is a deliberate and faithful embodiment of its foundational theory.