The Memento Pattern

**Intent**
- Capture and externalize an object's internal state, without violating encapsulation, so that the object state can be restored later.
  * Hides state storage implementation details from the object.
  * Useful for persistent storage and transaction rollback operations.

This pattern resolves the following forces:
1. Hide (and therefore allows transparent change of) state storage implementation details.
2. Externalize an objects internal state without breaking encapsulation.

**Structure of the Memento Pattern**

```
Originator
createMemento ()
setMemento (Memento &m)
state
```

```
Memento
setState ()
getState ()
state
```

```
Caretaker
state = m.getState ();
return new Memento (state)
```

**Use of the Memento Pattern**

```cpp
// IterationState is the Memento for our Iterator.
template <class T, class IterationState> class Iterator
public:
  virtual IterationState * first () = 0;
  // Resets the iterator to the beginning.

  virtual int next (IterationState *) = 0;
  // Advance the iterator to the next item.

  virtual int is_done (const IterationState *) const = 0;
  // Returns 1 if we've seen all the items, else 0.

  virtual int current_item (const IterationState *,
    T &item) const = 0;
  // Accesses the current item.
```
Advantages of Memento-based Iterator

- Derived (concrete) Iterators need not store any state.
  - The iteration state is stored in the IterationState class.

- An Iterator does not need to be a friend of its collection. Collection-specific private details are confined the Memento (IterationState class).

- Readily enables support for multiple iterators on one collection.