The Marker-Generic-Concrete (MGC) Pattern

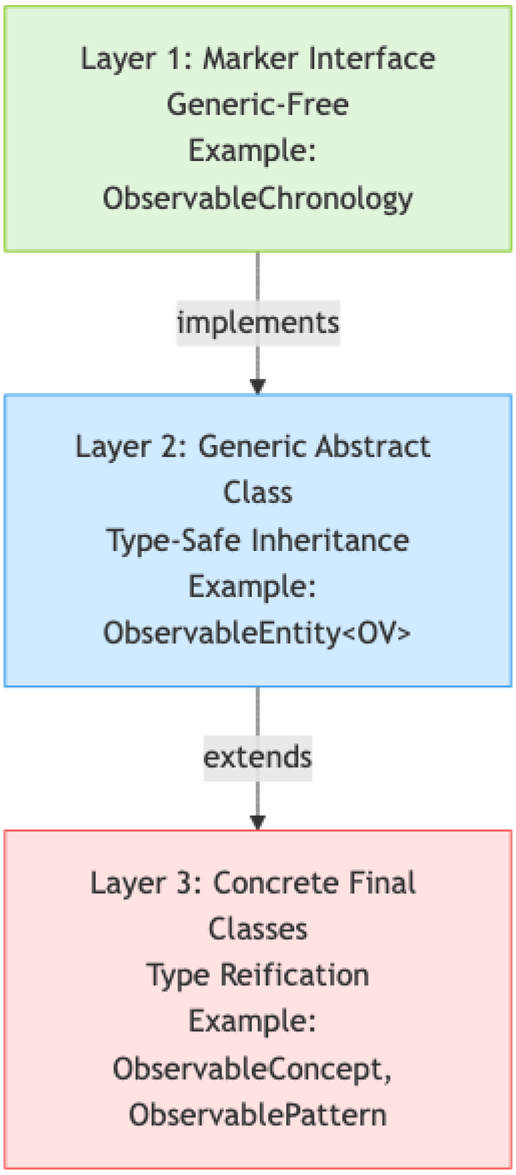
Table of Contents

Overview . . . 1
The Problem Space . . . 2
The Solution: MGC Pattern . . . 4
Implementation Guide . . . 6
Pattern Benefits . . . 8
Relationship to Other Patterns . . . 10
When to Use MGC Pattern . . . 12
Common Pitfalls and Solutions . . . 13
Real-World Example: Observable Entity Hierarchy . . . 14
Comparison with Alternatives . . . 18
Conclusion . . . 19
References . . . 20

Overview

The **Marker-Generic-Concrete (MGC) Pattern** is a three-layer type hierarchy design that balances API simplicity with compile-time type safety in sealed class hierarchies. It eliminates generic complexity from both consumer-facing APIs and concrete implementations while maintaining full type safety in the intermediate inheritance layer.

Pattern Structure



Key Characteristics

- **Generic-free bookends**: Simple interfaces at top and bottom layers
- **Type-safe middle**: Generic parameters constrained to the inheritance layer
- **Sealed hierarchy**: Fixed set of known implementations
- **No runtime casting**: Full compile-time type safety throughout
- **API flexibility**: Clients choose their level of abstraction

The Problem Space

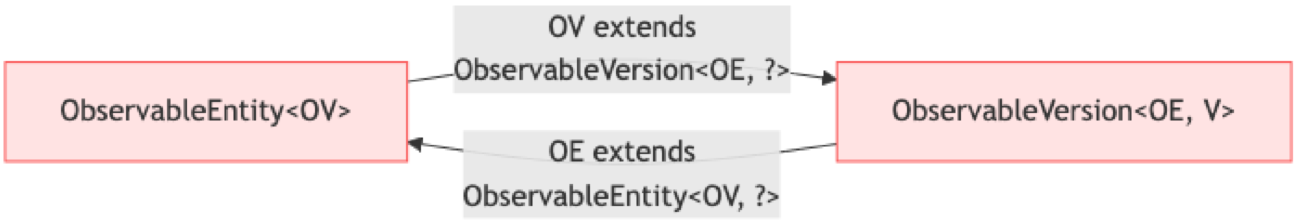
Challenge: Generic Complexity in APIs

Traditional sealed generic hierarchies force complexity onto all consumers:

```java
// Traditional approach - generics everywhere
public sealed class Entity<V extends EntityVersion>
    permits ConceptEntity, SemanticEntity, PatternEntity, StampEntity {
}

// Consumer code becomes verbose
public class MyProcessor {
    // Forced to use generics even when version type doesn't matter
    public void process(Entity<? extends EntityVersion> entity) { }

    // Or lose type information entirely
    public void process(Entity<?> entity) { }

    // Collection declarations become unwieldy
    List<Entity<? extends EntityVersion>> entities = new ArrayList<>();
}
```

Challenge: Circular Generic Constraints

When types reference each other, the complexity multiplies:


ObservableEntity<OV>
OV extends ObservableVersion<OE, ?>
ObservableVersion<OE, V>
OE extends ObservableEntity<OV, ?>

```java
// Without MGC - circular complexity
public abstract class ObservableEntity<OV extends ObservableVersion<OE, ?>>
public abstract class ObservableVersion<OE extends ObservableEntity<OV, ?>, V>

// Downstream code suffers
public interface ComponentArea<
    OE extends ObservableEntity<ObservableVersion<OE, ?>>,
    OV extends ObservableVersion<OE, ?>,
    FX extends Pane> {
    // Three type parameters just to work with entities!
}
```

Challenge: Wildcard Proliferation

Attempting to simplify with wildcards loses type safety:

```java
// Wildcards everywhere - lost type information
public void process(Entity<?> entity) {
    // No type safety - need instanceof checks
    if (entity instanceof ConceptEntity conceptEntity) {
        // Safe operations, but verbose
    }
}
```

The Solution: MGC Pattern

Layer 1: The Marker Interface

Purpose: Provide a generic-free type alias for downstream consumption.

```
/**
 * Marker interface for observable chronologies (versioned entities).
 * Simplifies downstream signatures by eliminating generic parameters.
 */
public sealed interface ObservableChronology
        permits ObservableConcept, ObservablePattern,
                ObservableSemantic, ObservableStamp {
    // No methods required - pure type marker
}
```

Benefits:

- Simple type name for method signatures
- No generic noise in downstream code
- Acts as a "type alias" in the type system
- Enables clean collection declarations

Usage in Downstream Code:

```
// Clean, generic-free API
public void process(ObservableChronology chronology) { }

// Simple collections
List<ObservableChronology> chronologies = new ArrayList<>();

// No wildcards needed
ObservableList<ObservableChronology> selected =
    FXCollections.observableArrayList();
```

Layer 2: The Generic Abstract Class

Purpose: Introduce type parameters for type-safe inheritance.

```
/**
 * Generic abstract class providing type-safe inheritance.
 * Type parameter OV tracks the observable version type.
 */
public abstract sealed class ObservableEntity<OV extends ObservableVersionType>
        implements Entity<OV>, ObservableChronology
        permits ObservableConcept, ObservablePattern,
                ObservableSemantic, ObservableStamp {

    // Type-safe version storage
    private MutableIntObjectMap<OV> versionMap;

    // Type-safe version access
    public ImmutableList<OV> versions() {
        return Lists.immutable.ofAll(versionMap.values());
    }

    // Type-safe abstract method for subclasses
    protected abstract OV wrap(EntityVersion version);
}
```

Benefits:

- Type parameters enable type-safe inheritance
- Subclasses can refine generic types
- Compiler enforces type consistency
- Internal framework code benefits from type safety

Layer 3: Concrete Final Classes

Purpose: Reify generic types, providing concrete type-safe implementations.

```
/**
 * Concrete final implementation with no generic parameters.
 * Type is fully specified - ObservableConceptVersion is concrete.
 */
public final class ObservableConcept
        extends ObservableEntity<ObservableConceptVersion>
        implements ConceptEntity<ObservableConceptVersion> {

    ObservableConcept(ConceptEntity<ConceptVersionRecord> conceptEntity) {
        super(conceptEntity);
    }

    @Override
    protected ObservableConceptVersion wrap(EntityVersion version) {
        return new ObservableConceptVersion(this,
            (ConceptVersionRecord) version);
    }

    // Concrete, type-safe accessor - no generics!
    public ObservableConceptVersion lastVersion() {
        return versions().getLast();
    }
}
```

Benefits:

- No generic parameters - simple class declaration
- Fully type-safe - no casting required
- Final - prevents further complexity
- Clean API for consumers

Implementation Guide

Step 1: Define the Marker Interface

Create a sealed interface with no methods:

```
public sealed interface [Domain]
        permits [Concrete1], [Concrete2], [Concrete3] {
    // Intentionally empty - pure type marker
}
```

Naming Convention: Use the domain name alone or with `Type` suffix:

- `ObservableChronology` □
- `ObservableVersionType` □
- `EntityType` □

Step 2: Create the Generic Abstract Class

Implement the marker and introduce type parameters:

```
public abstract sealed class [Domain]<T extends [Constraint]>
        implements [Marker]
        permits [Concrete1], [Concrete2], [Concrete3] {

    // Use generic type T in internal structures
    private final T value;

    // Type-safe abstract methods for subclasses
    protected abstract T transform(T input);

    // Type-safe public API
    public T getValue() {
        return value;
    }
}
```

Key Points:

- Must implement the marker interface
- Use `sealed` to control permitted subclasses
- Generic parameters should be bounded
- Provide type-safe abstract methods for subclasses

Step 3: Implement Concrete Final Classes

Extend the abstract class with specific types:

```
public final class [Specific][Domain]
        extends [Domain]<[ConcreteType]> {

    [Specific][Domain]([Parameters]) {
        super(parameters);
    }

    @Override
    protected [ConcreteType] transform([ConcreteType] input) {
        // Type-safe implementation
        return input.process();
    }

    // Add type-specific methods - no generics!
    public [ConcreteType] getSpecific() {
        return getValue();
    }
}
```

Key Points:

- Use `final` to prevent further inheritance
- Fully specify all generic parameters
- No generic parameters in the class declaration
- Provide type-specific convenience methods

Step 4: Provide Type-Safe Accessor

Create helper methods that eliminate runtime casting:

```
public final class ObservableConcept
        extends ObservableEntity<ObservableConceptVersion> {

    // Type-safe accessor - consumers don't need to cast
    public ObservableConcept getObservableConcept() {
        return this;
    }

    // Or provide in the parent class for all types
}

// In abstract class:
public abstract sealed class ObservableVersion<V extends EntityVersion> {

    public final ObservableChronology getObservableEntity() {
        return observableEntity;  // Returns marker interface
    }
}

// In concrete class - override with specific type:
public final class ObservableConceptVersion {

    public ObservableConcept getObservableConcept() {
        return (ObservableConcept) getObservableEntity();
    }
}
```

Pattern Benefits

For API Consumers

Benefit	Description
Simple Signatures	Method parameters use generic-free marker interface
No Wildcard Hell	Collections declared as `List<Marker>` instead of `List<? extends Generic<?>>`
Clear Intent	Type names communicate purpose without generic noise
Easy Migration	Existing code using wildcards can adopt marker gradually
IntelliSense Friendly	IDEs provide cleaner code completion without generic clutter

For Framework Developers

Benefit	Description
Type Safety	Full compile-time checking in inheritance layer
Refactoring Safety	Type errors caught at compile time, not runtime
Sealed Hierarchy	Complete control over permitted implementations
No instanceof	Type system prevents need for runtime type checking
Future-Proof	Adding new concrete types doesn't affect existing code

Code Comparison

Without MGC Pattern:

```
// Consumer code - verbose and complex
public class Processor {
    public void process(
        ObservableEntity<? extends ObservableVersion<?, ?>> entity,
        ObservableList<? extends ObservableVersion<?, ?>> versions) {

        // Wildcards everywhere, limited type safety
        for (ObservableVersion<?, ?> version : versions) {
            // Need instanceof checks
            if (version instanceof ObservableConceptVersion cv) {
                processConceptVersion(cv);
            }
        }
    }
}
```

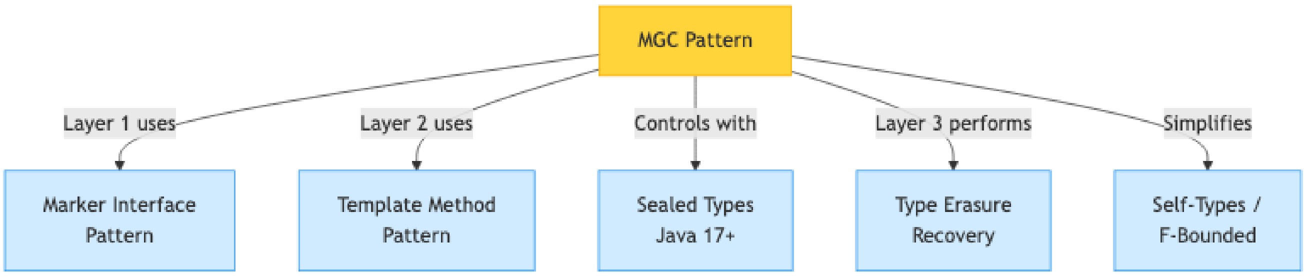
With MGC Pattern:

```
// Consumer code - clean and simple
public class Processor {
    public void process(
        ObservableChronology chronology,
        ObservableList<? extends ObservableVersionType> versions) {

        // Pattern matching on concrete types
        for (ObservableVersionType version : versions) {
            switch (version) {
                case ObservableConceptVersion cv ->
                    processConceptVersion(cv);
                case ObservableSemanticVersion sv ->
                    processSemanticVersion(sv);
                case ObservablePatternVersion pv ->
```

```
                    processPatternVersion(pv);
                case ObservableStampVersion stv ->
                    processStampVersion(stv);
            }
        }
    }
}
```

Relationship to Other Patterns

Pattern Relationships Overview



Marker Interface Pattern

Relationship: The MGC pattern’s Layer 1 is a pure Marker Interface.

Difference: Traditional marker interfaces (like `Serializable`) have no hierarchy. MGC uses the marker as part of a three-layer structure.

```
// Traditional Marker Interface
public interface Serializable { }

// MGC Marker Interface
public sealed interface ObservableChronology
    permits /* concrete types */ { }
```

Template Method Pattern

Relationship: Layer 2 (generic abstract class) often uses Template Method.

Synergy: Abstract methods in Layer 2 define the template, concrete classes in Layer 3 provide implementations.

```
// Layer 2 - Template
public abstract sealed class ObservableEntity<OV> {
    protected abstract OV wrap(EntityVersion version);  // Template method
}
```

```
// Layer 3 - Implementation
public final class ObservableConcept extends ObservableEntity<...> {
    @Override
    protected ObservableConceptVersion wrap(EntityVersion version) {
        return new ObservableConceptVersion(this, version);
    }
}
```

Type Erasure Recovery

Relationship: Layer 3 "recovers" the generic types through reification.

Innovation: MGC goes beyond traditional type erasure recovery by eliminating generics entirely in Layer 3, not just capturing them.

```
// Traditional - still generic
public class ArrayList<E> extends AbstractList<E> {
    // Generic parameter preserved
}

// MGC - no generics at leaf
public final class ObservableConcept
    extends ObservableEntity<ObservableConceptVersion> {
    // Generic parameter gone from declaration
}
```

Sealed Type Hierarchy (Java 17+)

Relationship: MGC leverages sealed types to control the hierarchy.

Requirement: The pattern works best with sealed types but can be adapted without them using package-private constructors.

```
// With sealed types (Java 17+)
public sealed interface ObservableChronology
    permits ObservableConcept, ObservablePattern { }

// Without sealed types (pre-Java 17)
public interface ObservableChronology {
    // Use package-private constructors in implementations
}
```

Self-Types / F-Bounded Polymorphism

Relationship: The generic layer sometimes uses self-referential bounds, but simpler than full F-bounded polymorphism.

Difference: F-bounded polymorphism keeps the self-reference through all layers. MGC eliminates it in Layer 3.

```
// F-bounded polymorphism
public interface Comparable<T extends Comparable<T>> { }
public class String implements Comparable<String> { }  // Still generic

// MGC approach
public sealed interface ObservableVersionType { }  // Layer 1
public abstract class ObservableVersion<V> { }      // Layer 2
public final class ObservableConceptVersion { }     // Layer 3 - no generic!
```

Strategy Pattern

Relationship: The concrete implementations in Layer 3 can be viewed as different "strategies" for handling specific types.

Difference: Strategy is typically about behavior, MGC is about structure and type safety.

Adapter Pattern

Relationship: The marker interface (Layer 1) acts as an adapter, providing a unified interface for different concrete types.

Difference: Traditional Adapter wraps unrelated classes. MGC is part of the inheritance structure from the start.

When to Use MGC Pattern

Ideal Scenarios □

Scenario	Rationale
Sealed Type Hierarchies	You have a fixed set of known implementations
Consumer-Facing APIs	Downstream code shouldn’t need generic complexity
Type-Safe Frameworks	Internal code needs compile-time type safety
JavaFX/UI Integration	Properties and collections benefit from simpler types
High-Level Abstractions	Business logic works at marker interface level
Plugin Architectures	Plugins implement concrete types, core uses marker

Poor Fit Scenarios □

Scenario	Why It’s a Poor Fit
Open Extension	External code needs to create new subtypes
Single Implementation	Pattern adds unnecessary complexity for one type
Generic Algorithms	Code that genuinely needs to work with any `T`
Simple DTOs	Data Transfer Objects don’t benefit from hierarchy
Performance Critical	Extra indirection may impact hot paths (profile first)

Common Pitfalls and Solutions

Pitfall 1: Forgetting to Implement Marker in Abstract Class

Problem:

```
public sealed interface ObservableChronology { }

// Missing implements ObservableChronology!
public abstract sealed class ObservableEntity<OV> { }

public final class ObservableConcept extends ObservableEntity<...> { }
// Compiler error: ObservableConcept doesn't implement ObservableChronology
```

Solution:

```
public abstract sealed class ObservableEntity<OV>
        implements ObservableChronology {  // ← Add this
    // ...
}
```

Pitfall 2: Incorrect Permits Clause

Problem:

```
// Wrong - lists abstract class
public sealed interface ObservableChronology
        permits ObservableEntity { }  // □ Wrong!
```

Solution:

```
// Correct - lists final concrete classes
public sealed interface ObservableChronology
        permits ObservableConcept, ObservablePattern,
                ObservableSemantic, ObservableStamp { }  // ✓
```

Pitfall 3: Exposing Generic Abstract Class in API

Problem:

```
// Bad - forces consumers to deal with generics
public void process(ObservableEntity<? extends ObservableVersionType> entity) { }
```

Solution:

```
// Good - use marker interface
public void process(ObservableChronology chronology) { }
```

Pitfall 4: Not Making Leaf Classes Final

Problem:

```
// Missing final - allows uncontrolled extension
public class ObservableConcept extends ObservableEntity<...> { }
```

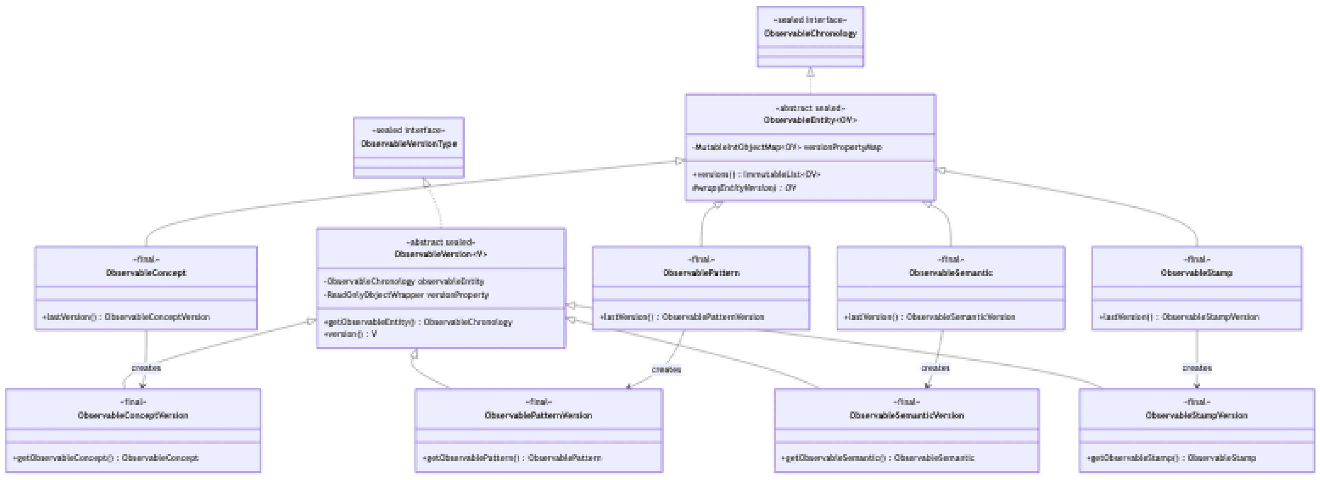
Solution:

```
// Correct - final prevents further extension
public final class ObservableConcept extends ObservableEntity<...> { }
```

Real-World Example: Observable Entity Hierarchy

This example from the Komet framework demonstrates MGC in practice:

Complete Class Hierarchy


«sealed interface»
ObservableChronology
«abstract sealed»
ObservableEntity<OV>
-MutableIntObjectMap<OV> versionPropertyMap
+versions() : ImmutableList<OV>
#wrap(EntityVersion) : OV
«sealed interface»
ObservableVersionType
«final»
ObservableConcept
+lastVersion() : ObservableConceptVersion
«abstract sealed»
ObservableVersion<V>
-ObservableChronology observableEntity
-ReadOnlyObjectWrapper versionProperty
+getObservableEntity() : ObservableChronology
+version() : V
«final»
ObservablePattern
+lastVersion() : ObservablePatternVersion
«final»
ObservableSemantic
+lastVersion() : ObservableSemanticVersion
«final»
ObservableStamp
+lastVersion() : ObservableStampVersion
creates
creates
creates
creates
«final»
ObservableConceptVersion
+getObservableConcept() : ObservableConcept
«final»
ObservablePatternVersion
+getObservablePattern() : ObservablePattern
«final»
ObservableSemanticVersion
+getObservableSemantic() : ObservableSemantic
«final»
ObservableStampVersion
+getObservableStamp() : ObservableStamp

Layer 1: Marker Interfaces

```
/**
 * Marker for observable chronologies (versioned entities).
 * Simplifies downstream signatures by eliminating generic parameters.
 */
public sealed interface ObservableChronology
        permits ObservableConcept, ObservablePattern,
                ObservableSemantic, ObservableStamp {
}

/**
 * Marker for observable versions.
 * Enables generic-free version handling.
 */
public sealed interface ObservableVersionType
        permits ObservableConceptVersion, ObservablePatternVersion,
                ObservableSemanticVersion, ObservableStampVersion {
}
```

Layer 2: Generic Abstract Classes

```
/**
 * Generic abstract class for type-safe entity inheritance.
 */
public abstract sealed class ObservableEntity<OV extends ObservableVersionType>
        implements Entity<OV>, ObservableChronology
        permits ObservableConcept, ObservablePattern,
                ObservableSemantic, ObservableStamp {

    private MutableIntObjectMap<OV> versionPropertyMap = new IntObjectHashMap<>();

    // Type-safe version storage and access
    @Override
    public ImmutableList<OV> versions() {
        return Lists.immutable.ofAll(versionPropertyMap.values());
    }

    // Abstract method for subclasses - type-safe
    protected abstract OV wrap(EntityVersion version);
}

/**
 * Generic abstract class for type-safe version inheritance.
 */
public abstract sealed class ObservableVersion<V extends EntityVersion>
        implements EntityVersion, ObservableVersionType
        permits ObservableConceptVersion, ObservablePatternVersion,
                ObservableSemanticVersion, ObservableStampVersion {

    private final ObservableChronology observableEntity;
    private final ReadOnlyObjectWrapper<EntityVersion> versionProperty;

    ObservableVersion(ObservableChronology observableEntity, V entityVersion) {
        this.observableEntity = observableEntity;
        this.versionProperty = new ReadOnlyObjectWrapper<>(entityVersion);
    }

    public final ObservableChronology getObservableEntity() {
        return observableEntity;
    }

    public V version() {
        return (V) versionProperty.getValue();
    }
}
```

Layer 3: Concrete Final Classes

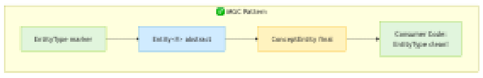
```
/**
 * Concrete observable concept - no generics!
 */
public final class ObservableConcept
        extends ObservableEntity<ObservableConceptVersion>
        implements ConceptEntity<ObservableConceptVersion> {

    ObservableConcept(ConceptEntity<ConceptVersionRecord> conceptEntity) {
        super(conceptEntity);
    }

    @Override
    protected ObservableConceptVersion wrap(EntityVersion version) {
        return new ObservableConceptVersion(this,
            (ConceptVersionRecord) version);
    }

    // Type-safe, generic-free methods
    public ObservableConceptVersion lastVersion() {
        return versions().getLast();
    }
}

/**
 * Concrete observable concept version - no generics!
 */
public final class ObservableConceptVersion
        extends ObservableVersion<ConceptVersionRecord>
        implements ConceptEntityVersion {

    ObservableConceptVersion(ObservableConcept observableConcept,
                            ConceptVersionRecord conceptVersionRecord) {
        super(observableConcept, conceptVersionRecord);
    }

    // Type-safe accessor - returns concrete type
    public ObservableConcept getObservableConcept() {
        return (ObservableConcept) getObservableEntity();
    }

    @Override
    public ConceptVersionRecord getVersionRecord() {
        return version();
    }
}
```

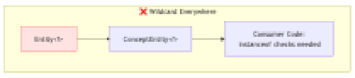
Consumer Code Benefits

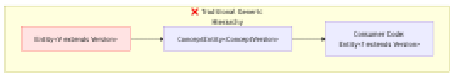
```
/**
 * Knowledge Layout component using MGC pattern types.
 * Notice: No generic parameters needed!
 */
public sealed interface KlChronologyArea<FX extends Pane>
        extends KlArea<FX> {

    // Simple, generic-free property
    ObjectProperty<ObservableChronology> chronologyProperty();

    default ObservableChronology observableChronology() {
        return chronologyProperty().get();
    }

    // Versions use marker interface - no wildcards!
    ObservableList<? extends ObservableVersionType> selectedVersions();
}

/**
 * Type-specific implementation provides concrete types.
 */
public non-sealed interface KlConceptArea<FX extends Pane>
        extends KlChronologyArea<FX> {

    // Override with specific type
    @Override
    ObjectProperty<ObservableConcept> chronologyProperty();

    // Type-safe accessor
    default ObservableList<ObservableConceptVersion> selectedConceptVersions() {
        return (ObservableList<ObservableConceptVersion>) selectedVersions();
    }
}
```

Comparison with Alternatives

Visual Comparison of Approaches







❌ Traditional Generic Hierarchy
Entity<V extends Version>
ConceptEntity<ConceptVersion>
Consumer Code: Entity<? extends Version>

Alternative 1: Keep Generics Throughout

```
// No marker interface - generics everywhere
public abstract class ObservableEntity<OV extends ObservableVersion<OE, ?>,
                                       OE extends ObservableEntity<OV, OE>> { }

// Consumer suffers
public interface ComponentArea<
    OE extends ObservableEntity<OV, OE>,
    OV extends ObservableVersion<OE, ?>,
    FX extends Pane> {
    // Complex signatures everywhere
}
```

Verdict: □ Too complex for consumers, verbose signatures

Alternative 2: Use Wildcards Everywhere

```
// Wildcard approach
public void process(ObservableEntity<?, ?> entity) {
```

```java
    // Lost type information
}

public void process(List<? extends ObservableVersion<?, ?>> versions) {
    // Wildcard hell
}
```

Verdict: ▯ Loses type safety, wildcard complexity

Alternative 3: Non-Generic Base with Getters

```java
// Non-generic base
public abstract class ObservableEntity {
    public abstract List<? extends ObservableVersion> versions();
}

// Generic subclass
public abstract class TypedObservableEntity<OV> extends ObservableEntity {
    @Override
    public List<OV> versions() { }
}
```

Verdict: ▯▯ Works but less elegant, can't constrain versions in base

Alternative 4: MGC Pattern

```java
// Layer 1: Simple marker
public sealed interface ObservableChronology { }

// Layer 2: Generic abstract
public abstract sealed class ObservableEntity<OV>
    implements ObservableChronology { }

// Layer 3: Concrete final
public final class ObservableConcept
    extends ObservableEntity<ObservableConceptVersion> { }

// Consumer - clean and simple
public void process(ObservableChronology chronology) { }
```

Verdict: ▯ Best balance of simplicity and type safety

Conclusion

The Marker-Generic-Concrete (MGC) pattern represents a thoughtful solution to a common challenge in Java: balancing API simplicity with compile-time type safety. By using a three-layer architecture—a generic-free marker interface, a generic abstract class, and concrete final implementations—the pattern enables:

- **Simple, generic-free APIs** for consumers
- **Type-safe inheritance** for framework developers
- **Compile-time guarantees** without runtime casting
- **Clean, maintainable code** at all levels

The pattern is particularly valuable in:

- Large frameworks with sealed type hierarchies
- JavaFX applications requiring property bindings
- Plugin architectures with fixed core types
- APIs exposed to external consumers

While not suitable for every situation, the MGC pattern shines when you need to provide both flexibility (work with any type via the marker) and specificity (work with concrete types) while maintaining complete type safety throughout.

References

Related Patterns

- **Marker Interface Pattern** - Pure marker without hierarchy
- **Template Method Pattern** - Abstract methods in generic layer
- **Sealed Types (Java 17+)** - Controlling permitted subclasses
- **Self-Types / F-Bounded Polymorphism** - Self-referential generic bounds
- **Type Erasure Recovery** - Capturing generic types in concrete classes

Further Reading

- *Effective Java* (3rd Edition) by Joshua Bloch - Item 29: Favor generic types
- *Java Generics and Collections* by Maurice Naftalin and Philip Wadler
- JEP 409: Sealed Classes (Java 17) - https://openjdk.org/jeps/409
- *Design Patterns: Elements of Reusable Object-Oriented Software* by Gang of Four

---

Document Version: 1.0
Last Updated: 2025-11-07 + // ← Should be today's date *Pattern Origin: Komet Framework Development*