Badges

[Java CI - Build with Maven] [Quality Gate Status] [Maven Central]

DDD Building Blocks Best Practices

DDD Building Blocks Best Practices is a base template project for maven.

Quick Start

Add `ddd-building-blocks` dependency in your project.

Example 1. pom.xml

```
<dependency>
    <groupId>io.github.bhuwanupadhyay</groupId>
    <artifactId>ddd-building-blocks</artifactId>
    <version>{revnumber}</version>
</dependency>
```

Article

Domain-driven design (DDD), is an approach used to build systems that have a complex business domain. So you wouldn't apply DDD to, say, infrastructure software or building routers, proxies, or caching layers, but instead to business software that solves real-world business problems. It's a great technique for separating the way the business is modeled from the plumbing code that ties it all together. Separating these two in the software itself makes it easier to design, model, build and evolve an implementation over time.

In the implementation, DDD building blocks play an important role in how business is modeled into the code. In this article, I will take you through the best available options for building blocks in object-oriented principles.

[layer] | *../assets/layer.png*

Value Object

```
import java.util.Objects;

public abstract class ValueObject {

  @Override
  public int hashCode() {
    return this.toHashCode();
  }

  /**
   * This method will determine hash code of your Value Object attributes. {@link
DomainError}, is
   * one of value object created by using {@link ValueObject}.
   *
   * @return hash code of an object
   */
  protected abstract int toHashCode();

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    ValueObject valueObject = (ValueObject) o;
    return Objects.equals(this, valueObject);
  }

  public String getObjectName() {
    return this.getClass().getName();
  }
}
```

Entity

```
import java.util.Objects;

public abstract class Entity<ID extends ValueObject> {

  public static final String ENTITY_ID_IS_REQUIRED = "Entity Id is required.";

  private final ID id;

  public Entity(ID id) {

    DomainAsserts.raiseIfNull(
        id, DomainError.create(getObjectName() + ".id", ENTITY_ID_IS_REQUIRED));

    this.id = id;
  }

  public ID getId() {
    return this.id;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || this.getClass() != o.getClass()) return false;
    Entity<?> entity = (Entity<?>) o;
    return Objects.equals(this.id, entity.id);
  }

  @Override
  public int hashCode() {
    return Objects.hash(this.id);
  }

  public String getObjectName() {
    return this.getClass().getName();
  }
}
```

Domain Error, Exception and Assertion

```
import java.util.Objects;
import java.util.function.Supplier;

public final class DomainError extends ValueObject {

  public static final String DOMAIN_ERROR_CODE_SHOULD_NOT_BE_BLANK =
      "Domain error code should not be blank.";
```

```
  public static final String DOMAIN_ERROR_MESSAGE_SHOULD_NOT_BE_BLANK =
      "Domain error message should not be blank.";

  private final String errorCode;

  private final String errorMessage;

  public DomainError(String errorCode, String errorMessage) {

    DomainAsserts.begin()
        .raiseIfBlank(
            errorCode,
            create(getObjectName() + ".errorCode",
DOMAIN_ERROR_CODE_SHOULD_NOT_BE_BLANK))
        .raiseIfBlank(
            errorMessage,
            create(getObjectName() + ".errorMessage",
DOMAIN_ERROR_MESSAGE_SHOULD_NOT_BE_BLANK))
        .ifHasErrorsThrow();

    this.errorCode = errorCode;
    this.errorMessage = errorMessage;
  }

  public static Supplier<DomainError> create(final String errorCode, final String
errorMessage) {
    return () -> new DomainError(errorCode, errorMessage);
  }

  @Override
  protected int toHashCode() {
    return Objects.hash(errorCode, errorMessage);
  }

  @Override
  public String toString() {
    return "DomainError{"
        + "violator='"
        + errorCode
        + '\''
        + ", message='"
        + errorMessage
        + '\''
        + '}';
  }
}
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public final class DomainValidationException extends RuntimeException {

  private final List<DomainError> domainErrors = new ArrayList<>();

  public DomainValidationException(List<DomainError> domainErrors) {
    super("[" + domainErrors.size() + "] domain errors.");
    this.domainErrors.addAll(domainErrors);
  }

  public List<DomainError> getDomainErrors() {
    return Collections.unmodifiableList(domainErrors);
  }
}
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;
import java.util.function.Supplier;

public final class DomainAsserts {

  private DomainAsserts() {}

  public static DomainAssert begin() {
    return new DomainAssert();
  }

  public static <T> void raiseIfNull(T value, Supplier<DomainError> error) {
    begin().raiseIfNull(value, error).ifHasErrorsThrow();
  }

  public static class DomainAssert {
    private final List<DomainError> domainErrors = new ArrayList<>();

    private DomainAssert() {}

    public <T> DomainAssert raiseIfNull(T value, Supplier<DomainError> error) {
      if (Objects.isNull(value)) {
        this.domainErrors.add(error.get());
      }
      return this;
    }

    public DomainAssert raiseIfBlank(String value, Supplier<DomainError> error) {
      raiseIfNull(value, error);

      if (value != null && value.isBlank()) {
        this.domainErrors.add(error.get());
      }
      return this;
    }

    public void ifHasErrorsThrow() {
      if (this.domainErrors.isEmpty()) {
        return;
      }
      throw new DomainValidationException(this.domainErrors);
    }
  }
}
```

Domain Event

```
import java.util.UUID;

public abstract class DomainEvent {

  private final String eventId = UUID.randomUUID().toString();

  private final String eventClassName = getClass().getName();

  private final String domainEventType;

  protected DomainEvent(DomainEventType domainEventType) {
    DomainAsserts.raiseIfNull(
        domainEventType,
        DomainError.create(objectName() + ".domainEventType", "Domain event type is
required."));
    this.domainEventType = domainEventType.name();
  }

  public String getEventId() {
    return eventId;
  }

  public String getEventClassName() {
    return eventClassName;
  }

  public String getDomainEventType() {
    return domainEventType;
  }

  public String objectName() {
    return getClass().getName();
  }

  public enum DomainEventType {
    /** Represents domain event is inside same bounded context only. */
    INSIDE_CONTEXT,
    /** Represents domain event is only for other bounded context. */
    OUTSIDE_CONTEXT,
    /**
     * Represents domain event is available for both i.e. inside same bounded context
and other
     * bounded context.
     */
    BOTH
  }
}
```

Service

```
/** This this designed to communicate between different aggregates. */
public interface DomainService {}
```

Repositories

```
import java.util.List;
import java.util.Optional;

public abstract class DomainRepository<T extends AggregateRoot<ID>, ID extends
ValueObject> {

  public static final String ENTITY_IS_REQUIRED = "Entity is required.";

  private final DomainEventPublisher publisher;

  protected DomainRepository(DomainEventPublisher publisher) {
    this.publisher = publisher;
  }

  public abstract Optional<T> findOne(ID id);

  public T save(T entity) {
    DomainAsserts.raiseIfNull(
        entity, DomainError.create(objectName() + ".entity", ENTITY_IS_REQUIRED));
    T persisted = this.persist(entity);
    entity.getDomainEvents().forEach(publisher::publish);
    entity.clearDomainEvents();
    return persisted;
  }

  protected abstract T persist(T entity);

  public abstract List<T> findAll();

  public String objectName() {
    return getClass().getName();
  }
}
```

Factories

Example Scenario

[rtms] | *../assets/rtms.png*