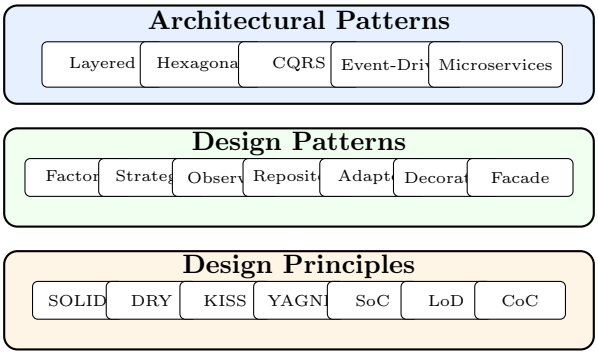
Designer's View

Architecture Viewpoint Specification

Design Patterns, Principles, Decisions & Implementation Guidance



Version:	2.0
Status:	Release
Classification:	ISO/IEC/IEEE 42010 Compliant
Last Updated:	December 12, 2025

Based on the Views and Beyond approach to software architecture documentation

Contents

1 Viewpoint Name

Viewpoint Identification

Name:	Designer's View
Synonyms:	Technical Design View, Implementation View, Developer's View, Software Design View, Detailed Design View
Identifier:	VP-DES-001
Version:	2.0

1.1 Viewpoint Classification

The Designer's View addresses the concerns of software designers and developers who need to understand detailed technical design decisions, patterns, principles, and implementation guidance. While the Development Viewpoint focuses on code organization and build structure, the Designer's View focuses on how to design and implement functionality within that structure.

Table 1: Viewpoint Classification Taxonomy

Attribute	Value
Style Family	Cross-cutting (Design Guidance)
Primary Focus	Design Patterns, Principles, and Implementation
Abstraction Level	Detailed Design / Implementation
Temporal Perspective	Design-time and Implementation-time
Related Styles	Module Views, Component-and-Connector
IEEE 42010 Category	Development Viewpoint (design aspects)
Relationship	Guides implementation within architecture

1.2 Viewpoint Scope

The Designer's View encompasses the following aspects:

- **Architectural Patterns:** High-level structural patterns guiding system organization.
- **Design Patterns:** Reusable solutions to common design problems.
- **Design Principles:** Fundamental principles guiding design decisions.
- **Interface Contracts:** Detailed interface specifications and contracts.
- **Design Decisions:** Key technical decisions with rationale and trade-offs.
- **Implementation Guidance:** Best practices and coding standards.
- **Design Constraints:** Technical constraints affecting design choices.

- **Quality Attribute Tactics:** Specific techniques achieving quality goals.

2 Overview

The Designer's View provides software designers and developers with the detailed guidance needed to implement the architecture correctly and consistently. It bridges the gap between high-level architectural decisions and code-level implementation.

2.1 Purpose and Scope

The primary purpose of this viewpoint is to communicate design intent, document design decisions, provide implementation guidance, and ensure consistent application of patterns and principles across the system.

Viewpoint Definition

The Designer's View documents the technical design decisions, architectural and design patterns, design principles, interface contracts, implementation guidance, and quality attribute tactics that guide system implementation. It provides designers and developers with the knowledge needed to make consistent, high-quality design decisions aligned with architectural intent.

2.2 Key Characteristics

The Designer's View exhibits several distinctive characteristics:

Technical Depth: Provides detailed technical guidance beyond high-level architecture.

Pattern-Oriented: Organizes design knowledge around proven patterns and practices.

Decision-Centric: Documents key design decisions with rationale and alternatives.

Practical Focus: Emphasizes actionable guidance over abstract theory.

Quality-Driven: Connects design choices to quality attribute achievement.

2.3 Relationship to Other Viewpoints

The Designer's View connects to other architectural viewpoints:

Table 2: Relationships to Other Viewpoints

Viewpoint	Relationship
Development	Designer's View guides implementation within module structure. Patterns apply within packages.
Logical	Design patterns realize logical services. Domain patterns implement domain model.
Component-and-Connector	Component patterns define runtime structure. Connector patterns define interactions.
Information	Data patterns guide data access and management. Repository pattern connects to data layer.
Process	Concurrency patterns guide thread design. Async patterns define communication.
Operational	Resilience patterns guide fault tolerance. Observability patterns enable monitoring.

2.4 Design Architecture Overview

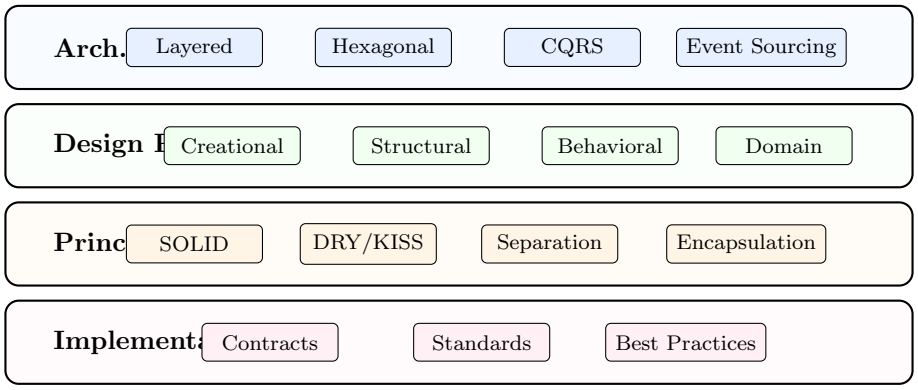


Figure 1: Design Architecture Layers

3 Concerns

This section enumerates the design concerns that the Designer's View is designed to address.

3.1 Primary Concerns

C1: Architectural Pattern Selection

- What architectural patterns structure the system?
- Why were these patterns chosen?

- How do patterns interact?
- What are the pattern boundaries?
- What variations are applied?

C2: Design Pattern Application

- Which design patterns should be used where?
- How are patterns implemented in this context?
- What adaptations are needed?
- How do patterns compose?
- What anti-patterns should be avoided?

C3: Design Principles Adherence

- What principles guide design decisions?
- How are SOLID principles applied?
- Where are exceptions allowed?
- How is principle adherence verified?
- What trade-offs exist between principles?

C4: Interface Design

- How are interfaces defined?
- What contracts must implementations honor?
- How is versioning handled?
- What documentation is required?
- How are breaking changes managed?

C5: Design Decision Documentation

- What key decisions were made?
- What alternatives were considered?
- What is the rationale for each decision?
- What trade-offs were accepted?
- When should decisions be revisited?

C6: Quality Attribute Tactics

- What tactics achieve performance goals?
- How is reliability ensured?
- What security patterns are used?
- How is maintainability supported?
- What testability considerations exist?

C7: Error Handling Strategy

- How are errors represented?
- What exception hierarchy exists?

- How are errors propagated?
- What recovery strategies exist?
- How are errors logged and monitored?

C8: Cross-Cutting Concerns

- How is logging implemented?
- How is authentication/authorization handled?
- How is caching managed?
- How is validation performed?
- How are transactions managed?

C9: Code Organization

- How is code structured within modules?
- What naming conventions apply?
- How are dependencies managed?
- What layering exists within components?
- How is code reuse achieved?

C10: Testing Strategy

- How is testability designed in?
- What testing patterns are used?
- How are dependencies mocked?
- What test coverage is expected?
- How are integration tests structured?

3.2 Concern-Quality Attribute Mapping

Table 3: Concern to Quality Attribute Mapping

Concern	Maintain.	Testabil.	Perform.	Security	Reliabil.	Reusabil.	Flextic.	Underst.
Arch. Patterns	●	○	○	○	○	○	●	●
Design Patterns	●	●	○	○	○	●	●	○
Principles	●	●	–	–	○	●	●	●
Interfaces	●	●	○	○	○	●	●	○
Decisions	○	○	○	○	○	○	○	●
QA Tactics	○	○	●	●	●	–	○	–
Error Handling	○	○	○	○	●	○	○	○
Cross-Cutting	●	○	○	●	○	●	○	○
Code Org.	●	●	–	–	–	●	○	●
Testing	○	●	–	○	●	○	○	○

● = Primary impact, ○ = Secondary impact, – = Minimal impact

4 Anti-Concerns

Understanding what the Designer's View is *not* appropriate for helps stakeholders avoid misapplying this viewpoint.

4.1 Out of Scope Topics

AC1: Business Requirements

- Functional requirements specification
- User stories and acceptance criteria
- Business rules definition
- Use case descriptions
- Feature prioritization

AC2: Deployment and Operations

- Infrastructure configuration
- Container orchestration
- Deployment procedures
- Monitoring setup
- Incident management

AC3: Project Management

- Team organization
- Sprint planning
- Resource allocation
- Timeline management
- Risk management

AC4: High-Level Architecture

- System context
- External integrations
- Deployment topology
- Technology selection rationale
- Capacity planning

AC5: Detailed Algorithm Implementation

- Specific algorithm code
- Mathematical formulas
- Data structure implementations
- Performance optimizations
- Platform-specific code

Common Misapplications

Avoid using the Designer's View for:

- Defining what the system should do (use Requirements)
- High-level system structure (use Logical/C&C Views)
- Deployment and infrastructure (use Deployment View)
- Team and work organization (use Planner's View)
- Detailed code documentation (use code comments/API docs)

5 Typical Stakeholders

The Designer's View serves stakeholders involved in detailed system design and implementation.

5.1 Primary Stakeholders

Table 4: Primary Stakeholder Analysis

Stakeholder	Role Description	Primary Interests
Software Designers	Create detailed designs	Pattern selection, design decisions, quality tactics
Senior Developers	Implement complex features	Implementation guidance, patterns, best practices
Tech Leads	Guide team implementation	Standards, consistency, design review criteria
Software Architects	Define design standards	Pattern catalog, principles, decision framework
Code Reviewers	Evaluate implementations	Design compliance, pattern usage, principle adherence
QA Engineers	Design test strategies	Testability patterns, test design, coverage guidance

5.2 Secondary Stakeholders

Table 5: Secondary Stakeholder Analysis

Stakeholder	Role Description	Primary Interests
Junior Developers	Learn and implement	Learning patterns, understanding decisions
DevOps Engineers	Automate and deploy	Build integration, deployment patterns
Security Engineers	Ensure security	Security patterns, secure coding practices
Performance Eng.	Optimize performance	Performance patterns, optimization techniques
Technical Writers	Document system	Design documentation, API specifications
New Team Members	Onboard to project	Understanding design philosophy, conventions

5.3 Stakeholder Concern Matrix

Table 6: Stakeholder-Concern Responsibility Matrix

	Arch Pat.	Design Pat.	Principles	Interfaces	Decisions	QA Tactics	Errors	Cross-Cut	Code Org	Testing
Architect	R	A	R	A	A	R	A	R	A	C
Designer	C	R	R	R	R	R	R	R	R	R
Sr. Developer	I	R	R	R	C	C	R	R	R	R
Tech Lead	C	R	R	R	R	C	R	R	R	R
Reviewer	I	C	C	C	I	I	C	C	C	C
QA Engineer	I	I	I	C	I	C	C	I	I	R

R = Responsible, A = Accountable, C = Consulted, I = Informed

6 Model Types

The Designer's View employs several complementary model types to capture design knowledge.

6.1 Model Type Catalog

MT1: Pattern Catalog

- *Purpose:* Document applicable patterns

- *Primary Elements:* Patterns, contexts, applications
- *Key Relationships:* Applies-to, composes-with
- *Typical Notation:* Pattern templates, class diagrams

MT2: Architecture Decision Records (ADRs)

- *Purpose:* Document significant decisions
- *Primary Elements:* Decisions, context, consequences
- *Key Relationships:* Supersedes, relates-to
- *Typical Notation:* ADR templates (Markdown)

MT3: Interface Specifications

- *Purpose:* Define contracts between components
- *Primary Elements:* Interfaces, methods, types
- *Key Relationships:* Implements, extends
- *Typical Notation:* Interface definitions, OpenAPI

MT4: Class/Structure Diagrams

- *Purpose:* Show detailed class relationships
- *Primary Elements:* Classes, relationships, methods
- *Key Relationships:* Inherits, composes, uses
- *Typical Notation:* UML class diagrams

MT5: Sequence Diagrams

- *Purpose:* Show interaction flows
- *Primary Elements:* Objects, messages, lifelines
- *Key Relationships:* Calls, returns
- *Typical Notation:* UML sequence diagrams

MT6: State Diagrams

- *Purpose:* Model object state behavior
- *Primary Elements:* States, transitions, events
- *Key Relationships:* Transitions-to, triggers
- *Typical Notation:* UML state machine diagrams

MT7: Coding Standards Document

- *Purpose:* Define coding conventions
- *Primary Elements:* Rules, examples, rationale
- *Key Relationships:* Applies-to, overrides
- *Typical Notation:* Style guides, linter configs

MT8: Error Handling Guide

- *Purpose:* Define error handling strategy

- *Primary Elements:* Error types, handlers, policies
- *Key Relationships:* Handles, propagates
- *Typical Notation:* Exception hierarchies, flow diagrams

MT9: Quality Attribute Tactics Catalog

- *Purpose:* Document tactics for quality attributes
- *Primary Elements:* Attributes, tactics, implementations
- *Key Relationships:* Achieves, supports
- *Typical Notation:* Tactic tables, decision trees

6.2 Model Type Relationships

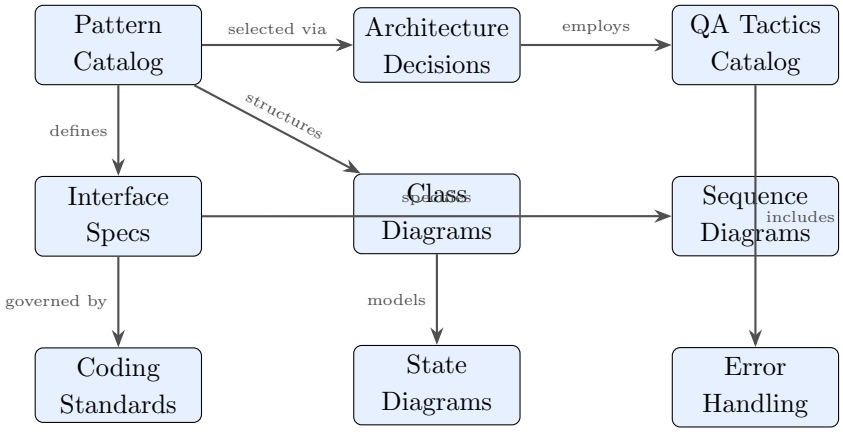


Figure 2: Model Type Dependency Relationships

7 Model Languages

For each model type, specific languages, notations, and techniques are prescribed.

7.1 Design Element Notation

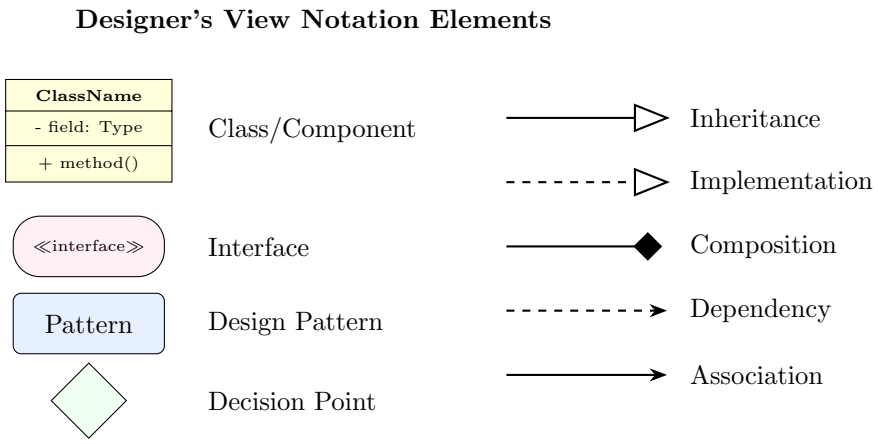


Figure 3: Designer's View Notation Legend

7.2 SOLID Principles

Table 7: SOLID Design Principles

	Principle	Description
S	Single Responsibility	A class should have only one reason to change
O	Open/Closed	Open for extension, closed for modification
L	Liskov Substitution	Subtypes must be substitutable for base types
I	Interface Segregation	Many specific interfaces over one general interface
D	Dependency Inversion	Depend on abstractions, not concretions

7.3 Design Pattern Categories

Table 8: Design Pattern Classification

Category	Purpose	Key Patterns
Creational	Object creation mechanisms	Factory, Abstract Factory, Builder, Singleton, Prototype
Structural	Object composition	Adapter, Bridge, Composite, Decorator, Facade, Proxy
Behavioral	Object interaction	Strategy, Observer, Command, State, Chain of Responsibility
Domain	Domain model patterns	Repository, Aggregate, Entity, Value Object, Domain Event
Architectural	System structure	Layered, Hexagonal, CQRS, Event Sourcing, Microservices
Integration	System integration	API Gateway, Circuit Breaker, Retry, Bulkhead, Saga

7.4 Architecture Decision Record Template

ADR Template

Title: ADR-NNN: [Decision Title]
Status: [Proposed — Accepted — Deprecated — Superseded]
Context: What is the issue that we're seeing that is motivating this decision or change?
Decision: What is the change that we're proposing and/or doing?
Consequences: What becomes easier or more difficult to do because of this change?
Alternatives Considered:

- Alternative 1: Description and why not chosen
- Alternative 2: Description and why not chosen

7.5 Tabular Specifications

7.5.1 Pattern Application Table

Table 9: Example Pattern Application Matrix

Pattern	Category	Applied To	Purpose
Repository	Domain	Data access layer	Abstract persistence operations
Factory	Creational	Service creation	Centralize object instantiation
Strategy	Behavioral	Payment processing	Swap payment providers
Observer	Behavioral	Event system	Decouple event producers/consumers
Decorator	Structural	Logging, caching	Add cross-cutting concerns
Adapter	Structural	External APIs	Normalize external interfaces

7.5.2 Quality Attribute Tactics Table

Table 10: Example Quality Attribute Tactics

Attribute	Tactic	Implementation	Trade-off
Performance	Caching	Redis for session, API responses	Memory cost, staleness
Performance	Connection Pool	HikariCP, 20-50 connections	Resource usage
Reliability	Circuit Breaker	Resilience4j, 50% threshold	Temporary unavailability
Reliability	Retry	Exponential backoff, 3 retries	Increased latency
Security	Input Validation	Schema validation at boundaries	Processing overhead
Testability	Dependency Injection	Constructor injection	Code complexity

8 Viewpoint Metamodels

This section defines the conceptual metamodel underlying the Designer's View.

8.1 Core Metamodel

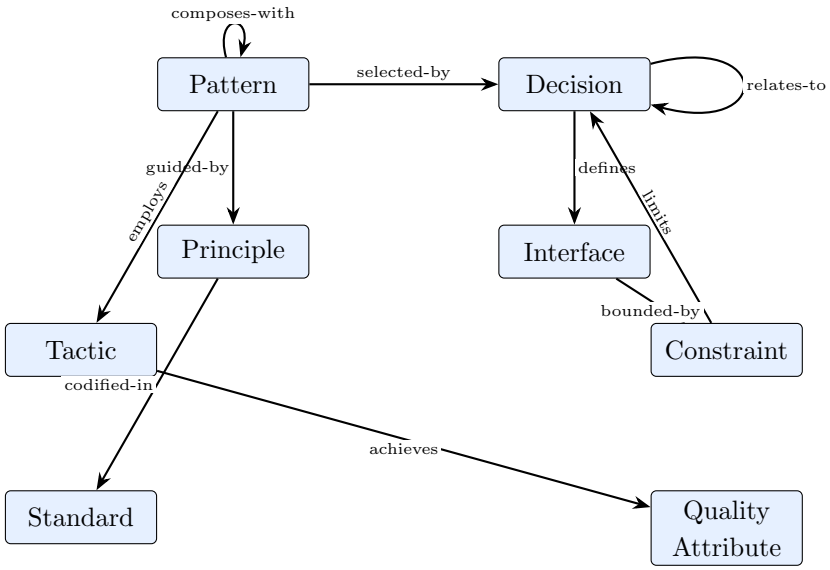


Figure 4: Designer's View Core Metamodel

8.2 Entity Definitions

Entity: Pattern

Definition: A reusable solution to a commonly occurring design problem within a given context.

Attributes:

- `patternId`: Unique identifier
- `name`: Pattern name
- `category`: Pattern category (creational, structural, behavioral, etc.)
- `intent`: What the pattern does
- `problem`: Problem the pattern solves
- `solution`: How the pattern solves the problem
- `structure`: Structural diagram
- `participants`: Key classes/interfaces
- `consequences`: Trade-offs and results
- `applicability`: When to use
- `relatedPatterns`: Related patterns

Constraints:

- Pattern should solve a recurring problem
- Trade-offs should be documented
- Applicability criteria should be clear

Entity: Decision

Definition: A significant technical choice that affects system design with documented rationale and consequences.

Attributes:

- `decisionId`: Unique identifier (ADR number)
- `title`: Decision title
- `status`: Current status (proposed, accepted, deprecated, superseded)
- `context`: Circumstances driving the decision
- `decision`: The choice made
- `rationale`: Why this choice was made
- `alternatives`: Other options considered
- `consequences`: Positive and negative outcomes
- `date`: When decision was made
- `deciders`: Who made the decision
- `supersedes`: Previous decision replaced

Constraints:

- Significant decisions should be documented
- Alternatives should be captured
- Consequences should include trade-offs

Entity: Principle

Definition: A fundamental guideline that directs design decisions and promotes desired qualities.

Attributes:

- `principleId`: Unique identifier
- `name`: Principle name
- `statement`: Concise principle statement
- `rationale`: Why this principle matters
- `implications`: What following this means
- `examples`: Examples of application
- `counterExamples`: Examples of violation
- `exceptions`: When exceptions are allowed
- `qualityAttributes`: Qualities supported

Constraints:

- Principles should be actionable
- Exceptions should be documented
- Conflicts between principles should be resolved

Entity: Interface

Definition: A contract defining the operations, data types, and behaviors that an implementation must provide.

Attributes:

- `interfaceId`: Unique identifier
- `name`: Interface name
- `description`: Interface purpose
- `version`: Interface version
- `operations`: List of operations
- `types`: Data types used
- `preconditions`: Required state before calls
- `postconditions`: Guaranteed state after calls
- `invariants`: Always-true conditions
- `exceptions`: Possible exceptions
- `qualityOfService`: Performance/reliability requirements

Constraints:

- Interfaces should be stable
- Breaking changes require versioning
- Contracts should be complete

Entity: Tactic

Definition: A specific design technique used to achieve or enhance a particular quality attribute.

Attributes:

- `tacticId`: Unique identifier
- `name`: Tactic name
- `qualityAttribute`: Target quality attribute
- `description`: How the tactic works
- `implementation`: How to implement
- `tradeoffs`: Quality trade-offs
- `applicability`: When to use
- `examples`: Implementation examples
- `relatedPatterns`: Patterns that employ this tactic

Constraints:

- Tactics should target specific qualities
- Trade-offs should be understood
- Implementation should be practical

Entity: Constraint

Definition: A limitation or restriction that bounds design choices.

Attributes:

- `constraintId`: Unique identifier
- `name`: Constraint name
- `description`: Constraint description
- `type`: Constraint type (technical, business, regulatory)
- `source`: Where constraint comes from
- `scope`: What the constraint applies to
- `impact`: How it affects design
- `flexibility`: How negotiable it is
- `verification`: How compliance is verified

Constraints:

- Constraints should be justified
- Impact should be understood
- Verification should be possible

Entity: Standard

Definition: A codified rule or convention that ensures consistency in implementation.

Attributes:

- `standardId`: Unique identifier
- `name`: Standard name
- `category`: Standard category (naming, formatting, architecture)
- `rule`: The specific rule
- `rationale`: Why this standard exists
- `examples`: Good and bad examples
- `enforcement`: How it's enforced (manual, automated)
- `exceptions`: Allowed exceptions
- `scope`: Where it applies

Constraints:

- Standards should be enforceable
- Exceptions should be documented
- Automated enforcement preferred

Entity: Quality Attribute

Definition: A measurable or testable property that indicates how well the system satisfies stakeholders.

Attributes:

- `attributeId`: Unique identifier
- `name`: Attribute name
- `definition`: What the attribute means
- `scenarios`: Quality attribute scenarios
- `measures`: How it's measured
- `targets`: Target values
- `tactics`: Tactics that support it
- `tradeoffs`: Conflicts with other attributes
- `priority`: Relative importance

Constraints:

- Attributes should be measurable
- Scenarios should be testable
- Trade-offs should be explicit

8.3 Relationship Definitions

Table 11: Metamodel Relationship Definitions

Relationship	Source	Target	Description
selected-by	Pattern	Decision	Pattern is chosen through decision
guided-by	Pattern	Principle	Pattern application follows principle
defines	Decision	Interface	Decision specifies interface design
codified-in	Principle	Standard	Principle is formalized as standard
achieves	Tactic	QA	Tactic improves quality attribute
limits	Constraint	Decision	Constraint restricts decision options
employs	Pattern	Tactic	Pattern uses tactic for quality
bounded-by	Interface	Constraint	Interface is limited by constraint
composes	Pattern	Pattern	Patterns combine together
relates-to	Decision	Decision	Decisions are interconnected

9 Conforming Notations

Several existing notations and methods support Designer's View modeling.

9.1 UML Diagrams

UML provides comprehensive notation for detailed design:

Class Diagrams: Classes, interfaces, relationships, attributes, methods.

Sequence Diagrams: Object interactions over time.

State Machine Diagrams: Object lifecycle and state transitions.

Conformance Level: High for structural and behavioral design.

9.2 Architecture Decision Records

ADRs document significant design decisions:

Elements: Context, decision, consequences, alternatives.

Format: Lightweight Markdown format (Michael Nygard style).

Conformance Level: High for decision documentation.

9.3 Pattern Languages

Pattern documentation follows standard templates:

Elements: Name, context, problem, solution, consequences.

Sources: GoF, POSA, EAA, DDD patterns.

Conformance Level: High for pattern documentation.

9.4 Notation Comparison

Table 12: Design Notation Comparison

Feature	UML	ADR	Pattern	OpenAPI	C4	Code
Class structure	●	–	○	–	○	●
Interactions	●	–	○	○	–	○
Decisions	–	●	○	–	–	–
Patterns	○	○	●	–	–	○
Interfaces	○	–	–	●	–	●
State behavior	●	–	○	–	–	○
Standards	–	–	–	○	–	●

● = Strong support, ○ = Limited support, – = Not applicable

10 Model Correspondence Rules

Model correspondence rules define how design elements relate to elements in other views.

10.1 Development View Correspondence

Correspondence Rule CR-01: Pattern to Module Mapping

Rule: Design patterns should be implemented within appropriate modules.

Formal Expression:

$$\forall p \in Patterns : \exists M \subseteq Modules : implements(M, p)$$

Rationale: Ensures patterns are realized in code structure.

Verification: Code review against pattern specifications.

Correspondence Rule CR-02: Standard to Code Mapping

Rule: Coding standards should be enforced in all code.

Formal Expression:

$$\forall c \in Code : \forall s \in Standards : complies(c, s)$$

Rationale: Ensures consistent code quality.

Verification: Automated linting and code review.

10.2 Component-and-Connector View Correspondence

Correspondence Rule CR-03: Interface to Component Mapping

Rule: Design interfaces should be realized as component interfaces.

Formal Expression:

$$\forall i \in DesignInterfaces : \exists c \in Components : realizes(c, i)$$

Rationale: Ensures design contracts are implemented at runtime.

Verification: Interface compliance testing.

11 Operations on Views

This section defines methods for creating, interpreting, analyzing, and maintaining design views.

11.1 Creation Methods

11.1.1 View Development Process

Step 1: Identify Applicable Patterns

1. Analyze quality attribute requirements
2. Review problem contexts
3. Select candidate architectural patterns
4. Select candidate design patterns
5. Document pattern rationale

Step 2: Document Design Decisions

1. Identify significant decisions
2. Document context and drivers
3. Evaluate alternatives
4. Record chosen option with rationale
5. Document consequences and trade-offs

Step 3: Define Interfaces

1. Identify interface boundaries
2. Define operations and signatures
3. Specify contracts (pre/post conditions)
4. Document data types
5. Define error handling

Step 4: Specify Quality Tactics

1. Map quality requirements to attributes
2. Select appropriate tactics
3. Document implementation approach
4. Identify trade-offs
5. Plan verification approach

Step 5: Create Design Models

1. Create class diagrams for key structures
2. Create sequence diagrams for key flows
3. Create state diagrams for stateful objects
4. Document pattern applications
5. Review for consistency

Step 6: Define Standards

1. Define naming conventions
2. Define formatting rules
3. Define architectural constraints
4. Configure automated enforcement
5. Document exceptions process

Step 7: Validate Design

1. Review against requirements
2. Verify pattern applicability
3. Check principle adherence
4. Validate interface completeness
5. Assess quality attribute support

11.1.2 Pattern Selection Process

Pattern Selection Criteria

Context Fit:

- Does the pattern address our specific problem?
- Does our context match the pattern's applicability?
- Are the pattern's assumptions valid in our context?

Quality Attribute Support:

- Does it support our quality goals?
- Are the trade-offs acceptable?
- How does it interact with other patterns?

Implementation Feasibility:

- Can our team implement it correctly?
- Does it fit our technology stack?
- Is the complexity justified?

11.2 Analysis Methods

11.2.1 Design Review Checklist

Design Review Criteria

Pattern Application:

- Is the pattern implemented correctly?
- Are pattern participants properly defined?
- Are pattern constraints respected?

Principle Adherence:

- Does each class have single responsibility?
- Are abstractions used appropriately?
- Is coupling minimized?

Interface Quality:

- Are contracts complete and clear?
- Is the interface minimal but sufficient?
- Are error cases handled?

12 Examples

12.1 Example 1: Repository Pattern Implementation

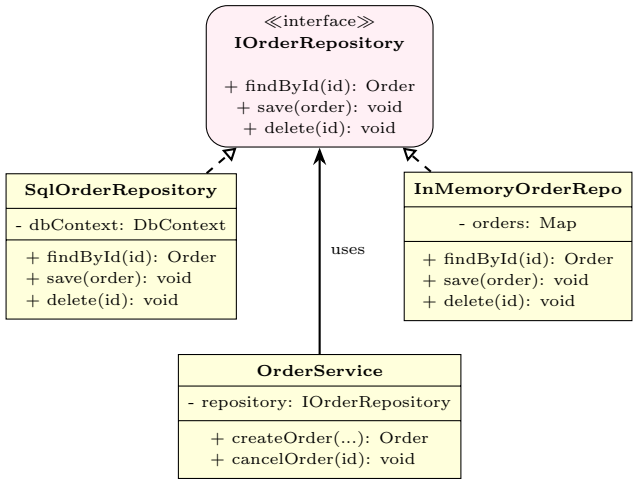


Figure 5: Repository Pattern Implementation

Description: This class diagram shows the Repository pattern applied to Order persistence. The IOrderRepository interface defines the contract. SqlOrderRepository implements it for production with database persistence. InMemoryOrderRepository provides a test double. OrderService depends only on the interface, enabling easy testing and implementation swapping.

12.2 Example 2: Architecture Decision Record

ADR-005: Use Event Sourcing for Order History

Status: Accepted

Context: We need to maintain complete order history for audit, customer service, and analytics. Orders go through many state changes and we need to know exactly what happened and when.

Decision: We will use Event Sourcing for the Order aggregate, storing all state changes as a sequence of domain events.

Consequences:

- **Positive:** Complete audit trail, temporal queries, replay capability
- **Negative:** Increased complexity, eventual consistency for read models, storage growth

Alternatives Considered:

- **Audit logging:** Would duplicate data, harder to replay
- **Soft deletes with history table:** Complex triggers, incomplete history

12.3 Example 3: Quality Tactic Implementation

Table 13: Circuit Breaker Tactic Configuration

Parameter	Value	Rationale
Failure Threshold	50%	Trip when half of calls fail
Minimum Calls	10	Need sample size before tripping
Wait Duration	30 seconds	Time before attempting recovery
Permitted in Half-Open	3 calls	Test calls before fully closing
Sliding Window Size	100 calls	Rolling window for statistics
Fallback Behavior	Return cached data	Graceful degradation

13 Notes

13.1 Design Principle Guidelines

Design Principle Application

- **Balance Principles:** Principles can conflict; find appropriate balance
- **Context Matters:** Apply principles considering specific context
- **Avoid Dogmatism:** Don't follow principles blindly
- **Document Exceptions:** When violating principles, document why
- **Iterate:** Refactor toward principles as understanding grows
- **Team Agreement:** Ensure team understands and agrees on principles

13.2 Pattern Application Guidelines

Effective Pattern Usage

- **Understand Intent:** Know why the pattern exists before using
- **Adapt Appropriately:** Patterns are templates, not rigid rules
- **Avoid Over-Engineering:** Don't use patterns where simple code suffices
- **Combine Carefully:** Understand how patterns interact
- **Name Clearly:** Use pattern names in code for clarity
- **Document Usage:** Explain how pattern is applied in context

13.3 Common Pitfalls

Design Anti-Patterns to Avoid

1. **God Class:** Class doing too much, knows too much
2. **Spaghetti Code:** Unstructured, tangled control flow
3. **Golden Hammer:** Using favorite pattern/tool everywhere
4. **Copy-Paste Programming:** Duplicating instead of abstracting
5. **Premature Optimization:** Optimizing before measuring
6. **Interface Bloat:** Interfaces with too many methods
7. **Leaky Abstraction:** Implementation details escaping interface
8. **Circular Dependencies:** Modules depending on each other

14 Sources

14.1 Primary References

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

2. Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.

3. Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
4. Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.
5. Bass, L., Clements, P., & Kazman, R. (2021). *Software Architecture in Practice* (4th ed.). Addison-Wesley.

14.2 Supplementary References

6. Freeman, E., & Robson, E. (2020). *Head First Design Patterns* (2nd ed.). O'Reilly.
7. Nygard, M. (2018). *Release It!* (2nd ed.). Pragmatic Bookshelf.
8. Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley.
9. Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.
10. Buschmann, F., et al. (1996). *Pattern-Oriented Software Architecture Volume 1*. Wiley.

14.3 Online Resources

- Refactoring Guru: `https://refactoring.guru/design-patterns`
- Martin Fowler's Patterns: `https://martinfowler.com/`
- ADR GitHub Organization: `https://adr.github.io/`
- Microsoft Design Patterns: `https://docs.microsoft.com/azure/architecture/patterns/`

A Designer's View Checklist

Item	Complete?
Patterns	
Architectural patterns selected and documented	☐
Design patterns identified for key problems	☐
Pattern rationale documented	☐
Pattern interactions understood	☐
Decisions	
Significant decisions documented as ADRs	☐
Alternatives captured	☐
Consequences documented	☐
Decision log maintained	☐
Interfaces	
Key interfaces defined	☐
Contracts specified (pre/post conditions)	☐
Versioning strategy defined	☐
Error handling documented	☐
Quality	
Quality tactics selected	☐
Tactics implementation documented	☐
Trade-offs understood	☐
Verification approach defined	☐
Standards	
Coding standards defined	☐
Naming conventions documented	☐
Automated enforcement configured	☐
Team trained on standards	☐

B Glossary

ADR Architecture Decision Record; documents significant decisions.

Anti-Pattern A common solution that is counterproductive.

Contract The specification of obligations and guarantees of an interface.

Design Pattern A reusable solution to a common design problem.

Invariant A condition that must always be true.

Postcondition A condition guaranteed after an operation completes.

Precondition A condition required before an operation can execute.

Principle A fundamental guideline directing design decisions.

Quality Attribute
A measurable property indicating system quality.

Refactoring Improving code structure without changing behavior.

SOLID Five design principles for maintainable OO code.

Tactic A specific technique for achieving a quality attribute.

Technical Debt
Cost of rework caused by expedient solutions.

Trade-off A balance between competing concerns.