Nooa Core Engine: Architectural Grammar Validator

Thiago Butignon

January 2025

Contents

1 Nooa Core Engine: Architectural Grammar Validator and Code Hygiene Guardian **2**
1.1 Executive Summary . . . 2
1.2 1. Introduction . . . 3
1.2.1 1.1 Motivation . . . 3
1.2.2 1.2 Approach . . . 3
1.2.3 1.3 Dogfooding Philosophy . . . 3
1.3 2. Theoretical Foundation . . . 4
1.3.1 2.1 Clean Architecture as Universal Grammar . . . 4
1.3.2 2.2 Clean Architecture BNF Grammar . . . 5
1.3.3 2.3 Grammatical Rules as Code Patterns . . . 6
1.4 3. Nooa Core Engine Architecture . . . 6
1.4.1 3.1 Layer Structure . . . 6
1.4.2 3.2 Dependency Rules . . . 7
1.4.3 3.3 Complete Flow Example . . . 8
1.5 4. Implemented Features . . . 9
1.5.1 4.1 Architectural Validation (v1.0-v1.1) . . . 9
1.5.2 4.2 Code Hygiene (v1.2) . . . 12
1.6 5. Type System . . . 16
1.6.1 5.1 Discriminated Unions . . . 16
1.6.2 5.2 YAML Schema Validation . . . 17
1.7 6. Results and Validation . . . 19
1.7.1 6.1 Self-Validation (Dogfooding) . . . 19
1.7.2 6.2 Project Metrics . . . 19
1.7.3 6.3 Performance . . . 20
1.7.4 6.4 Real-World Use Cases . . . 21
1.8 7. Comparison with Existing Tools . . . 22
1.9 8. Future Work . . . 23
1.9.1 8.1 Short Term (v1.3) . . . 23
1.9.2 8.2 Medium Term (v2.0) . . . 23
1.9.3 8.3 Long Term (v3.0+) . . . 23
1.10 9. Lessons Learned . . . 23

1.10.1 9.1 Dogfooding as Methodology 23
1.10.2 9.2 Constraints Breed Excellence 23
1.10.3 9.3 Type Safety vs. Flexibility 24
1.10.4 9.4 Barrel Export False Positives 24
1.11 10. Conclusion 24
1.11.1 10.1 Main Contributions 24
1.11.2 10.2 Practical Impact 24
1.11.3 10.3 Vision for the Future 25
1.12 References 25
1.13 Appendices 26
1.13.1 Appendix A: Complete Grammar Example 26
1.13.2 Appendix B: Data Structures 28

1 Nooa Core Engine: Architectural Grammar Validator and Code Hygiene Guardian

Technical Whitepaper v1.2

Authors: Thiago Butignon and Nooa AI Team **Date**: January 2025 **Version**: 1.2.0

1.1 Executive Summary

This whitepaper presents the **Nooa Core Engine**, an innovative tool that treats software architecture as a **formal grammar**, enabling automatic validation of architectural compliance in TypeScript projects. Inspired by Robert C. Martin's Clean Architecture principles and Rodrigo Manguinho's dogmatic approach, Nooa transforms subjective architectural rules into verifiable formal specifications.

Key Contributions:

1. **Architecture as Formal Grammar**: Modeling Clean Architecture using BNF (Backus-Naur Form) notation, treating architectural layers as linguistic elements (nouns, verbs, adverbs).
2. **Complete Validation System**: Combination of architectural validation (forbidden dependencies, circular dependencies, required dependencies) with code hygiene (synonym detection, dead code detection).
3. **Dogfooding as Methodology**: Application of the philosophy "use the grammar to build the validator of the grammar," ensuring the tool is a dogmatic example of Clean Architecture.

4. **Optimized Algorithms**: DFS implementation for cycle detection (O(V + E)), Jaro-Winkler for string similarity, and reverse graph analysis for unreferenced code.

Results: Nooa Core Engine v1.2.0 validates itself with **zero architectural errors**, demonstrating that formal rules can be rigorously applied without compromising productivity.

1.2 1. Introduction

1.2.1 1.1 Motivation

Software architecture is often treated as art, not science. Developers learn architectural patterns through examples, but lack a **formal and verifiable** way to ensure compliance. Common problems include:

- **Architectural Erosion**: Over time, architecture degrades as developers violate layer separation principles
- **Wrong Dependencies**: Domain depends on Infrastructure, violating the Dependency Inversion Principle
- **Semantic Duplication**: Multiple developers create classes with different names but identical functionality
- **Dead Code**: Old files remain in the repository for years without being used

Research Question: *Can software architecture be formalized as a grammar, allowing automatic validation analogous to natural language grammar checkers?*

1.2.2 1.2 Approach

We treat **Clean Architecture as a formal language** where:

- **Domain** = NOUN (entities, data)
- **Use Cases** = VERB (actions, behaviors)
- **Infrastructure** = ADVERB (how actions are performed)
- **Presentation** = CONTEXT (where/to whom)
- **Validation** = GRAMMAR CHECKER
- **Main** = SENTENCE COMPOSER

Just as compilers validate code syntax, Nooa validates "architectural syntax."

1.2.3 1.3 Dogfooding Philosophy

> **"We use the grammar to build the validator of the grammar."**

This is not just a clever idea - it's the **foundational principle** of Nooa. The tool that validates architectural rules must be a **dogmatic perfect example** of the architecture it enforces. This creates a virtuous cycle:

1. We define Clean Architecture rules in `nooa.grammar.yaml`
2. We implement features following those rules
3. We run Nooa against itself to verify compliance
4. Any violation indicates a problem in the rule or code
5. We fix and iterate

Result: `npm start .` must always return **zero violations**.

1.3 2. Theoretical Foundation

1.3.1 2.1 Clean Architecture as Universal Grammar

1.3.1.1 Linguistic Mapping

Architectural Element	Linguistic Role	Example
Domain Model	NOUN	`UserModel`, `ProductModel`
Use Case	TRANSITIVE VERB	`AddAccount`, `Authenticate`
Data Protocol	VERB MODIFIER	`AddAccountRepository`
Implementation	ACTIVE SENTENCE	`DbAddAccount` performs action
Adapter	ADVERB	`BcryptAdapter` (hashing *with bcrypt*)
Controller	CONTEXT/VOICE	`SignUpController` (in HTTP context)
Factory	SENTENCE COMPOSER	`makeDbAddAccount` assembles everything

1.3.1.2 Chomsky Analogy Clean Architecture exhibits properties of Chomsky's **Universal Grammar**:

1. **Poverty of Stimulus**: Developers can generate infinite valid implementations from finite rules
2. **Acquisition Device**: After seeing 2-3 examples, developers "acquire" the pattern
3. **Deep vs. Surface Structure**: Same architectural semantics, different syntax (TypeScript vs. Python)
4. **Recursion**: Controllers compose use cases, which compose protocols, infinitely
5. **Parameters and Constraints**: Inviolable rules (Domain cannot depend on Infrastructure)

1.3.2 2.2 Clean Architecture BNF Grammar

```
<program> ::= <domain> <data> <infrastructure> <presentation> <main>

<domain> ::= <models> <usecases>

<usecases> ::= <usecase>+
<usecase> ::= <usecase-interface> <usecase-namespace>

<usecase-interface> ::= "export interface" <UseCaseName> "{"
                        <verb> ":" "(" <params> ")" "=>" "Promise<" <result>
                        "}"

<data> ::= <protocols> <implementations>

<implementations> ::= <implementation>+
<implementation> ::= "export class" <ImplName> "implements" <UseCaseName> "{"
                     <constructor>
                     <method-implementation>
                     "}"

<constructor> ::= "constructor(" <protocol-dependencies> ")" "{}"
<protocol-dependencies> ::= ("private readonly" <dependency> ":" <ProtocolNam
```

Dependency Rule (Chomsky Hierarchy Level 2 - Context-Free):

```
<dependency-rule> ::= <higher-level> "→" <lower-level>
                    | <higher-level> "□ " <lower-level>  /* forbidden */
```

```
/* Allowed dependencies */
<allowed> ::= Domain □  Data
            | Domain □  Presentation
            | Domain □  Infrastructure  /* FORBIDDEN */
            | Data □  Infrastructure
            | Main → All

/* Direction constraint */
<constraint> ::= □  module M: dependencies(M) ⊆ { inner layers }
```

1.3.3 2.3 Grammatical Rules as Code Patterns

1.3.3.1 Rule DOM-001: Use Case Contract Completeness

```
<complete-usecase> ::= <interface> ∧ <namespace> ∧ <params> ∧ <result>
<violation> ::= <interface> ∧ ¬<namespace>  /* Incomplete sentence */
```

Grammatical Explanation: - Interface = Verb signature (transitive verb requires object) - Params = Direct object (what the verb acts upon) - Result = Predicate/Complement (what the verb produces)

1.3.3.2 Rule DATA-001: Dependency Inversion

```
<valid-implementation> ::= "implements" <DomainInterface> ∧
                           "constructor" "(" <Protocols>+ ")" ∧
                           ¬<ConcreteDependency>
```

Grammatical Explanation: - Verb implementation depends on abstract adverb, not concrete - Like defining “to run” generically, not “to run quickly”

1.4 3. Nooa Core Engine Architecture

1.4.1 3.1 Layer Structure

```
src/
├── domain/            # Enterprise Business Rules (innermost layer)
|   ├── models/          # Pure entities and types
|   |   ├── architectural-rule.model.ts
|   |   ├── architectural-violation.model.ts
|   |   └── code-symbol.model.ts
|   └── usecases/        # Use case interfaces (contracts)
|       └── analyze-codebase.usecase.ts (interface)
```

```
|
├── data/                # Application Business Rules
|   ├── protocols/      # Interface definitions for infrastructure
|   |   ├── code-parser.protocol.ts
|   |   └── grammar-repository.protocol.ts
|   └── usecases/        # Use case implementations
|       └── analyze-codebase.usecase.ts (implementation)
|
├── infra/               # Frameworks & Drivers (outermost layer)
|   ├── parsers/         # Parsing implementations (ts-morph)
|   |   └── ts-morph-parser.adapter.ts
|   └── repositories/    # Data access (YAML)
|       └── yaml-grammar.repository.ts
|
├── presentation/        # Interface Adapters
|   └── controllers/     # CLI controllers
|       └── cli.controller.ts
|
└── main/                # Composition Root
    ├── factories/       # Dependency injection factories
    |   ├── parser.factory.ts
    |   ├── repository.factory.ts
    |   └── usecase.factory.ts
    └── server.ts        # Application entry point
```

1.4.2 3.2 Dependency Rules

Following Clean Architecture, dependencies point inward:

```
Main (knows all layers)
  ↓
Presentation (depends on Domain interfaces)
  ↓
Infrastructure (implements Data protocols)
  ↓
Data (depends only on Domain)
  ↓
Domain (NO external dependencies)
```

Validation: Nooa itself checks its dependencies with:

```yaml
- name: "Domain-Independence"
```

```
  severity: error
  from:
    role: [NOUN, VERB_CONTRACT]
  to:
    role: [VERB_IMPLEMENTATION, ADVERB_CONCRETE]
  rule: "forbidden"
```

1.4.3 3.3 Complete Flow Example

1. Domain (Contract):

```
// src/domain/usecases/analyze-codebase.ts
export interface IAnalyzeCodebase {
  analyze: (params: IAnalyzeCodebase.Params) => Promise<IAnalyzeCodebase.Resu
}

export namespace IAnalyzeCodebase {
  export type Params = { projectPath: string; grammar: ArchitecturalGrammarMo
  export type Result = ArchitecturalViolationModel[];
}
```

2. Data (Implementation):

```
// src/data/usecases/analyze-codebase.usecase.ts
import { IAnalyzeCodebase } from '@/domain/usecases';
import { ICodeParser } from '../protocols';

export class AnalyzeCodebaseUseCase implements IAnalyzeCodebase {
  constructor(
    private readonly codeParser: ICodeParser,  // □  Protocol, not concrete i
    private readonly grammarRepository: IGrammarRepository
  ) {}

  async analyze(params: IAnalyzeCodebase.Params): Promise<IAnalyzeCodebase.Re
    const symbols = await this.codeParser.parse(params.projectPath);
    return this.validateArchitecture(symbols, params.grammar);
  }
}
```

3. Infrastructure (Adapter):

```
// src/infra/parsers/ts-morph-parser.adapter.ts
import { Project } from 'ts-morph';  // □  External library only in Infrastru
```

```typescript
import { ICodeParser } from '@/data/protocols';

export class TSMorphParserAdapter implements ICodeParser {
  async parse(projectPath: string): Promise<CodeSymbolModel[]> {
    const project = new Project({ tsConfigFilePath: `${projectPath}/tsconfig.
    // ... parsing with ts-morph
  }
}
```

4. Main (Composition):

```typescript
// src/main/factories/usecase.factory.ts
export const makeAnalyzeCodebaseUseCase = (): IAnalyzeCodebase => {
  const parser = makeCodeParser();           // Parser factory
  const repository = makeGrammarRepository(); // Repository factory
  return new AnalyzeCodebaseUseCase(parser, repository);
};
```

1.5 4. Implemented Features

1.5.1 4.1 Architectural Validation (v1.0-v1.1)

1.5.1.1 4.1.1 Forbidden Dependencies Syntax:

```yaml
- name: "Domain-Independence"
  severity: error
  from:
    role: DOMAIN
  to:
    role: INFRASTRUCTURE
  rule: "forbidden"
```

Implementation: Verifies that no symbol with `from.role` depends on symbols with `to.role`.

1.5.1.2 4.1.2 Circular Dependency Detection Algorithm: DFS with 3 states (WHITE/GRAY/BLACK)

```typescript
private detectCycles(graph: Map<string, string[]>): string[][] {
  const state = new Map<string, 'WHITE' | 'GRAY' | 'BLACK'>();
  const cycles: string[][] = [];
```

```
  function dfs(node: string, path: string[]): void {
    state.set(node, 'GRAY');  // Currently visiting
    path.push(node);

    for (const neighbor of graph.get(node) || []) {
      if (state.get(neighbor) === 'GRAY') {
        // Found cycle!
        const cycleStart = path.indexOf(neighbor);
        cycles.push([...path.slice(cycleStart), neighbor]);
      } else if (state.get(neighbor) === 'WHITE') {
        dfs(neighbor, [...path]);
      }
    }

    state.set(node, 'BLACK');  // Fully visited
  }

  // Visit all nodes
  for (const node of graph.keys()) {
    if (state.get(node) === 'WHITE') {
      dfs(node, []);
    }
  }

  return cycles;
}
```

Complexity: O(V + E)

Syntax:

```
- name: "No-Circular-Dependencies"
  severity: error
  from:
    role: ALL
  to:
    circular: true
  rule: "forbidden"
```

1.5.1.3 4.1.3 Required Dependencies

Validation: Ensures specific architectural connections exist.

```yaml
- name: "Use-Case-Must-Implement-Contract"
  severity: error
  from:
    role: VERB_IMPLEMENTATION
  to:
    role: VERB_CONTRACT
  rule: "required"
```

Implementation:

```typescript
private validateRequiredDependencies(
  symbols: CodeSymbolModel[],
  rule: DependencyRule
): ArchitecturalViolationModel[] {
  const violations: ArchitecturalViolationModel[] = [];

  for (const symbol of symbols) {
    if (this.roleMatches(symbol.role, rule.from.role)) {
      const hasRequiredDep = symbol.dependencies.some((dep) =>
        this.roleMatches(this.getRoleForPath(dep), rule.to.role)
      );

      if (!hasRequiredDep) {
        violations.push({
          rule: rule.name,
          severity: rule.severity,
          file: symbol.path,
          message: `${symbol.path} must depend on role ${rule.to.role}`,
        });
      }
    }
  }

  return violations;
}
```

1.5.1.4 4.1.4 Naming Pattern Validation **Syntax**:

```yaml
- name: "UseCase-Files-Follow-Convention"
  severity: warning
  for:
    role: VERB_IMPLEMENTATION
```

```
  pattern: "(\\.usecase\\.ts|/index\\.ts)$"
  rule: "naming_pattern"
```

Implementation: Regex validation at parse time with discriminated unions.

1.5.2 4.2 Code Hygiene (v1.2)

1.5.2.1 4.2.1 Synonym Detection

Problem: Multiple developers create classes with different names but identical functionality.

Solution: Jaro-Winkler Algorithm + Thesaurus-based Normalization

Syntax:

```
- name: "Detect-Duplicate-Use-Cases"
  severity: warning
  for:
    role: VERB_IMPLEMENTATION
  options:
    similarity_threshold: 0.85
    thesaurus:
      - [Analyze, Validate, Check, Verify]
      - [Create, Generate, Build, Make]
  rule: "find_synonyms"
```

Algorithm:

```
private validateSynonyms(
  symbols: CodeSymbolModel[],
  rule: SynonymDetectionRule
): ArchitecturalViolationModel[] {
  const violations: ArchitecturalViolationModel[] = [];
  const roleSymbols = symbols.filter((s) => this.roleMatches(s.role, rule.for

  // Pairwise comparison
  for (let i = 0; i < roleSymbols.length; i++) {
    for (let j = i + 1; j < roleSymbols.length; j++) {
      const name1 = this.normalizeName(
        this.extractFileName(roleSymbols[i].path),
        rule.options.thesaurus
      );
      const name2 = this.normalizeName(
        this.extractFileName(roleSymbols[j].path),
        rule.options.thesaurus
```

```
      );

      const similarity = this.calculateJaroWinkler(name1, name2);

      if (similarity >= rule.options.similarity_threshold) {
        violations.push({
          rule: rule.name,
          severity: rule.severity,
          file: roleSymbols[i].path,
          message: `${roleSymbols[i].path} and ${roleSymbols[j].path} are ${M
        });
      }
    }
  }

  return violations;
}

private calculateJaroWinkler(s1: string, s2: string): number {
  // Jaro-Winkler algorithm implementation
  // https://en.wikipedia.org/wiki/Jaro–Winkler_distance

  const jaroSimilarity = this.jaroSimilarity(s1, s2);

  // Extra weight for common prefixes (up to 4 characters)
  let prefixLength = 0;
  for (let i = 0; i < Math.min(4, Math.min(s1.length, s2.length)); i++) {
    if (s1[i] === s2[i]) {
      prefixLength++;
    } else {
      break;
    }
  }

  const p = 0.1;  // Default scaling factor
  return jaroSimilarity + (prefixLength * p * (1 - jaroSimilarity));
}
```

Complexity: O(N² × L²) where N = symbols, L = average string length

Normalization:

```
private normalizeName(name: string, thesaurus?: string[][]): string {
  let normalized = name.toLowerCase();

  // Remove common suffixes
  const suffixes = ['usecase', 'impl', 'adapter', 'repository', 'controller',
  for (const suffix of suffixes) {
    normalized = normalized.replace(new RegExp(`-?${suffix}$`), '');
  }

  // Apply thesaurus (replace synonyms with canonical term)
  if (thesaurus) {
    for (const group of thesaurus) {
      const canonical = group[0].toLowerCase();
      for (let i = 1; i < group.length; i++) {
        const regex = new RegExp(`\\b${group[i].toLowerCase()}\\b`, 'g');
        normalized = normalized.replace(regex, canonical);
      }
    }
  }

  return normalized;
}
```

1.5.2.2 4.2.2 Unreferenced Code Detection (Zombie Code)

Problem: Old files that are no longer used but were never deleted.

Solution: Reverse dependency graph analysis

Syntax:

```
- name: "Detect-Zombie-Files"
  severity: info
  for:
    role: ALL
  options:
    ignore_patterns:
      - "^src/main/server\\.ts$"  # Entry point
      - "/index\\.ts$"             # Barrel exports
  rule: "detect_unreferenced"
```

Algorithm:

```
private detectUnreferencedCode(
```

```
  symbols: CodeSymbolModel[],
  rule: UnreferencedCodeRule
): ArchitecturalViolationModel[] {
  const violations: ArchitecturalViolationModel[] = [];

  // Step 1: Build incoming references map
  const incomingReferences = new Map<string, number>();

  // Initialize all references to 0
  for (const symbol of symbols) {
    incomingReferences.set(symbol.path, 0);
  }

  // Count incoming references
  for (const symbol of symbols) {
    for (const dep of symbol.dependencies) {
      const current = incomingReferences.get(dep) || 0;
      incomingReferences.set(dep, current + 1);
    }
  }

  // Step 2: Find files with zero references
  for (const symbol of symbols) {
    const refCount = incomingReferences.get(symbol.path) || 0;

    // Step 3: Filter by ignore patterns
    const shouldIgnore = rule.options?.ignore_patterns?.some((pattern) =>
      new RegExp(pattern).test(symbol.path)
    );

    if (refCount === 0 && !shouldIgnore) {
      violations.push({
        rule: rule.name,
        severity: rule.severity,
        file: symbol.path,
        message: `${symbol.path} is not imported by any file (potential zombi
      });
    }
  }
```

```
  return violations;
}
```

Complexity: O(N) where N = total files

1.6 5. Type System

1.6.1 5.1 Discriminated Unions

For compile-time type safety, we use discriminated unions:

```
type DependencyRule = BaseRule & {
  from: RuleFrom;
  to: RuleTo;
  rule: 'allowed' | 'forbidden' | 'required';
};

type NamingPatternRule = BaseRule & {
  for: RuleFor;
  pattern: string;
  rule: 'naming_pattern';
};

type SynonymDetectionRule = BaseRule & {
  for: RuleFor;
  rule: 'find_synonyms';
  options: SynonymDetectionOptions;
};

type UnreferencedCodeRule = BaseRule & {
  for: RuleFor;
  rule: 'detect_unreferenced';
  options?: UnreferencedCodeOptions;
};

type ArchitecturalRuleModel =
  | DependencyRule
  | NamingPatternRule
  | SynonymDetectionRule
  | UnreferencedCodeRule;
```

Benefits:

- **Compile-time Safety**: TypeScript prevents accessing wrong properties
- **Type Guards**: Proper filtering and narrowing
- **Runtime Validation**: YAML parser validates structure

Type Guard Example:

```
const circularRules = grammar.rules.filter(
  (rule): rule is DependencyRule =>
    rule.rule !== 'naming_pattern' &&
    rule.rule !== 'find_synonyms' &&
    rule.rule !== 'detect_unreferenced' &&
    'to' in rule &&
    'circular' in rule.to &&
    rule.to.circular === true
);
```

1.6.2 5.2 YAML Schema Validation

```
export class YAMLGrammarRepository implements IGrammarRepository {
  async load(path: string): Promise<ArchitecturalGrammarModel> {
    const content = YAML.parse(fileContent);

    // Schema validation
    if (!content.version || !content.language || !content.roles || !content.r
      throw new Error('Invalid grammar file structure');
    }

    return {
      version: content.version,
      language: content.language,
      roles: content.roles.map((role: any) => ({
        name: role.name,
        path: role.path,
        description: role.description,
      })),
      rules: content.rules.map((rule: any) => {
        const baseRule = {
          name: rule.name,
          severity: rule.severity,
          comment: rule.comment,
```

```
        };

        // Discriminate by rule type
        if (rule.rule === 'find_synonyms') {
          return {
            ...baseRule,
            for: { role: rule.for.role },
            options: {
              similarity_threshold: rule.options.similarity_threshold,
              thesaurus: rule.options.thesaurus,
            },
            rule: 'find_synonyms' as const,
          };
        } else if (rule.rule === 'detect_unreferenced') {
          return {
            ...baseRule,
            for: { role: rule.for.role },
            options: { ignore_patterns: rule.options?.ignore_patterns },
            rule: 'detect_unreferenced' as const,
          };
        } else if (rule.rule === 'naming_pattern') {
          return {
            ...baseRule,
            for: { role: rule.for.role },
            pattern: rule.pattern,
            rule: 'naming_pattern' as const,
          };
        } else {
          // DependencyRule
          return {
            ...baseRule,
            from: { role: rule.from.role },
            to: { role: rule.to.role, circular: rule.to.circular },
            rule: rule.rule as 'allowed' | 'forbidden' | 'required',
          };
        }
      }),
    };
  }
}
```

1.7 6. Results and Validation

1.7.1 6.1 Self-Validation (Dogfooding)

Nooa validates itself using `nooa.grammar.yaml`:

```
$ npm start .

  Nooa Core Engine - Architectural Analysis
==================================================
  Analyzing project: .

  Valid architecture! Zero violations found.

  Performance Metrics
--------------------------------------------------
   Analysis Time: 416ms
  Rules Triggered: 16
  Total Violations: 0 (11 info - expected false positives)
--------------------------------------------------
```

Interpretation:

- **0 errors** - Perfect architectural compliance
- **0 warnings** - No duplicate synonyms detected
- **11 info** - Expected false positives from barrel exports

1.7.2 6.2 Project Metrics

```
Language: TypeScript
Architecture: Clean Architecture (Rodrigo Manguinho approach)
Layers: 5 (Domain, Data, Infrastructure, Presentation, Main)

Files by Layer:
- Domain:         8 files (pure business models)
- Data:           3 files (use case implementations)
- Infrastructure: 4 files (ts-morph, YAML parsers)
- Presentation:   1 file (CLI controller)
- Main:           2 files (factories, entry point)
- Docs:          10 files (comprehensive documentation)
```

```
Total TypeScript: ~1200 lines of production code
Total Documentation: ~3500 lines of markdown
```

1.7.3 6.3 Performance

Empirical Self-Validation Benchmarks:

We executed 5 independent tests to measure the real performance of Nooa validating itself:

```
Project Configuration:
- TypeScript Files: 22 files
- Lines of Code: 1920 lines
- Defined Roles: 10 roles
- Grammar Rules: 15 rules
  - 5 forbidden dependency rules
  - 3 required dependency rules
  - 3 naming pattern rules
  - 1 circular detection rule
  - 2 hygiene rules (synonyms + zombies)
  - 1 dependency inversion rule

Benchmark Results (5 executions):
```

Run	Analysis	Total Time	Memory (MB)
1	454ms	810ms	220
2	583ms	900ms	194
3	403ms	650ms	220
4	474ms	720ms	216
5	398ms	660ms	221
Average	462ms	748ms	214
StdDev	±68ms	±94ms	±11

```
Performance Details:
- Average Analysis Time: 462ms (±14.7%)
- Average Total Time: 748ms (includes initialization)
- Average Memory Usage: 214 MB
- Peak Memory: 221 MB
- Throughput: ~47 files/second
```

```
- Latency: ~21ms per file

Validation Results:
- ✅ 0 architectural errors
- ⚠️  0 warnings
- ℹ️  11 info (expected false positives from barrel exports)
```

Algorithm Complexity:

Feature	Algorithm	Complexity	Measured
Circular Detection	DFS with 3 states	O(V + E)	~45ms
Required Dependencies	Graph traversal	O(N × M)	~120ms
Forbidden Dependencies	Graph traversal	O(N × M × R)	~180ms
Synonym Detection	Jaro-Winkler	$O(N^2 \times L^2)$	~85ms
Unreferenced Code	Reverse graph	O(N)	~32ms

Performance Analysis:

- **Scalability**: Analysis time grows linearly with O(V + E) for dependency graphs
- **Memory**: Conservative usage of ~10MB per analyzed TypeScript file
- **CPU**: Single-threaded but highly optimized with early returns
- **I/O**: Lazy file reading, in-memory caching during analysis
- **Bottleneck**: Parsing with ts-morph (~60% of time), rule validation (~40%)

1.7.4 6.4 Real-World Use Cases

1.7.4.1 Example 1: Preventing Architectural Violations Before Nooa:

```
// src/domain/models/user.ts
import { Database } from '../../infra/database'; // ❌ Domain depends on Infr
```

After Nooa:

```
❌ ERROR: Domain-Independence
domain/models/user.ts cannot depend on infra/database.ts
```

1.7.4.2 Example 2: Detecting Duplicate Logic Before Nooa:

- `CreateUserUseCase` (2023)
- `UserCreatorService` (2024) ← Duplicate!

After Nooa:

```
⚠️  WARNING: Detect-Duplicate-Use-Cases
```

```
89% similarity between CreateUserUseCase and UserCreatorService
Consider consolidating.
```

1.7.4.3 Example 3: Finding Dead Code Before Nooa:

- Old files remain in repository for years
- Unused dependencies accumulate

After Nooa:

```
   INFO: Detect-Zombie-Files
old-payment-adapter.ts is not imported anywhere
```

1.8 7. Comparison with Existing Tools

Tool	Architecture	Hygiene	Grammar	Dogfooding
Nooa	Complete	Synonyms + Zombies	Formal YAML	Self-validates
dependency-cruiser	Dependencies	No	Complex JSON	No
eslint-plugin-boundaries	Basic	No	ESLint config	No
ArchUnit (Java)	Complete	No	Java code	No
NDepend (.NET)	Complete	Basic	CQL	No

Nooa's Differentiators:

1. **Grammar as First-Class**: Declarative YAML, not complex configuration
2. **Natural Language**: Grammatical explanations of violations
3. **Code Hygiene**: Beyond architecture, detects semantic problems
4. **Dogfooding**: Proves rules work through self-application

---

1.9 8. Future Work

1.9.1 8.1 Short Term (v1.3)

- **Multi-Language Support**: JavaScript, Python, Java
- **HTML/JSON Reports**: Rich visualizations of violations
- **VS Code Extension**: Real-time validation during development
- **Configuration Presets**: Templates for React, Node.js, etc.

1.9.2 8.2 Medium Term (v2.0)

- **Machine Learning**: Duplicate detection using semantic embeddings
- **Complexity Metrics**: Cyclomatic complexity, cognitive complexity
- **Evolution Tracking**: Track architectural degradation over time
- **Fitness Functions**: Continuous validation of architectural objectives

1.9.3 8.3 Long Term (v3.0+)

- **Microservice Validation**: Dependencies between services
- **Module Boundaries**: Bounded context enforcement (DDD)
- **Plugin System**: Custom validators
- **ArchUnit-like DSL**: Fluent API for programmatic rules

1.10 9. Lessons Learned

1.10.1 9.1 Dogfooding as Methodology

Key Lesson: Self-application forces architectural excellence.

During development, dogfooding revealed:

- **Edge cases in grammar syntax**: Ambiguities discovered when writing rules for Nooa itself
- **Performance bottlenecks**: Analysis of own code showed inefficiencies
- **Missing features**: Need for `_ACTUAL` roles to exclude barrel exports

Result: The tool improved the tool.

1.10.2 9.2 Constraints Breed Excellence

Quote: *"The grammar constraints forced better design decisions."*

When Nooa reports:

❌ src/domain/models cannot depend on src/infra

We don't disable the rule - we **refactor the code** to be cleaner.

1.10.3 9.3 Type Safety vs. Flexibility

Challenge: TypeScript discriminated unions vs. YAML flexibility

Solution: Parser validates at runtime, TypeScript guarantees at compile-time

```
// Parse YAML (runtime)
const rule = parseYAML(content);

// Type guard (compile-time safety)
if (rule.rule === 'find_synonyms') {
  // TypeScript knows rule.options.similarity_threshold exists
  const threshold = rule.options.similarity_threshold;
}
```

1.10.4 9.4 Barrel Export False Positives

Problem: Barrel exports (`index.ts`) cause false positives in unreferenced code

Solution: Creation of specialized _ACTUAL roles:

```
- name: VERB_IMPLEMENTATION_ACTUAL
  path: "^src/data/usecases/.*\\.usecase\\.ts$"  # Excludes index.ts
```

1.11 10. Conclusion

1.11.1 10.1 Main Contributions

This work demonstrates that:

1. **Architecture can be formalized**: Clean Architecture exhibits formal grammar properties
2. **Automatic validation is viable**: Architectural rules can be verified automatically
3. **Dogfooding works**: Self-application ensures quality and credibility
4. **Code hygiene matters**: Beyond structure, semantic problems (duplication, dead code) degrade quality

1.11.2 10.2 Practical Impact

Nooa Core Engine enables teams to:

- **Prevent architectural erosion**: Detect violations before merge
- **Maintain consistency**: Ensure everyone follows the same patterns
- **Reduce technical debt**: Eliminate duplicate and dead code
- **Accelerate onboarding**: New developers learn architecture through rules

1.11.3 10.3 Vision for the Future

Objective: Make architectural validation as common as code linting.

Just as no one commits without `npm run lint`, the goal is that no one commits without `npm run arch-validate`.

Call to Action:

```
npm install -g nooa-core-engine
nooa .
```

If it returns **zero violations**, your architecture is healthy. If not, Nooa shows exactly what to fix.

1.12 References

1. **Martin, Robert C.** (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.
2. **Manguinho, Rodrigo**. *Clean Architecture in TypeScript*. Available at: https://www.youtube.com/@RodrigoManguinho
3. **Chomsky, Noam** (1965). *Aspects of the Theory of Syntax*. MIT Press.
4. **Winkler, William E.** (1990). *String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage*. Proceedings of the Section on Survey Research Methods, American Statistical Association.
5. **Fowler, Martin**. *Technical Debt Quadrant*. Available at: https://martinfowler.com/bliki/Technica
6. **Evans, Eric** (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.
7. **Ford, Neal et al.** (2017). *Building Evolutionary Architectures*. O'Reilly Media.

1.13 Appendices

1.13.1 Appendix A: Complete Grammar Example

```yaml
version: 1.2
language: typescript

roles:
  - name: NOUN
    path: "^src/domain/models"
    description: "Domain entities and types"

  - name: VERB_CONTRACT
    path: "^src/domain/usecases"
    description: "Use case interfaces (contracts)"

  - name: VERB_IMPLEMENTATION
    path: "^src/data/usecases/.*\\.usecase\\.ts$"
    description: "Use case implementations"

  - name: ADVERB_ABSTRACT
    path: "^src/data/protocols"
    description: "Abstract protocols for infrastructure"

  - name: ADVERB_CONCRETE
    path: "^src/infra"
    description: "Concrete infrastructure adapters"

  - name: CONTEXT
    path: "^src/presentation"
    description: "Controllers and presenters"

rules:
  # 1. Circular Dependencies
  - name: "No-Circular-Dependencies"
    severity: error
    from:
      role: ALL
    to:
      circular: true
    rule: "forbidden"
```

```yaml
# 2. Domain Independence
- name: "Domain-Independence"
  severity: error
  from:
    role: [NOUN, VERB_CONTRACT]
  to:
    role: [VERB_IMPLEMENTATION, ADVERB_CONCRETE, CONTEXT]
  rule: "forbidden"

# 3. Required Dependencies
- name: "Use-Cases-Implement-Contracts"
  severity: error
  from:
    role: VERB_IMPLEMENTATION
  to:
    role: VERB_CONTRACT
  rule: "required"

# 4. Naming Patterns
- name: "UseCase-Naming-Convention"
  severity: warning
  for:
    role: VERB_IMPLEMENTATION
  pattern: "\\.usecase\\.ts$"
  rule: "naming_pattern"

# 5. Synonym Detection
- name: "Detect-Duplicate-Use-Cases"
  severity: warning
  for:
    role: VERB_IMPLEMENTATION
  options:
    similarity_threshold: 0.85
    thesaurus:
      - [Analyze, Validate, Check, Verify]
      - [Create, Generate, Build, Make]
  rule: "find_synonyms"

# 6. Zombie Code Detection
```

```yaml
  - name: "Detect-Zombie-Files"
    severity: info
    for:
      role: ALL
    options:
      ignore_patterns:
        - "^src/main/server\\.ts$"
        - "/index\\.ts$"
    rule: "detect_unreferenced"
```

1.13.2 Appendix B: Data Structures

```typescript
// Domain Models
export type ArchitecturalGrammarModel = {
  version: string;
  language: string;
  roles: RoleDefinitionModel[];
  rules: ArchitecturalRuleModel[];
};

export type RoleDefinitionModel = {
  name: string;
  path: string;
  description?: string;
};

export type CodeSymbolModel = {
  name: string;
  path: string;
  role: string;
  dependencies: string[];
};

export type ArchitecturalViolationModel = {
  rule: string;
  severity: RuleSeverity;
  file: string;
  message: string;
  details?: string;
};
```

```typescript
export type RuleSeverity = 'error' | 'warning' | 'info';
```

Document Generated with Clean Architecture Principles License: MIT **Repository**: https://github.com/nooa-ai/nooa-core-engine

For more information: - README.md - Main documentation - docs/DOGFOODING_PHILOSOPHY.md - Self-validation philosophy - docs/HYGIENE_RULES.md - Complete code hygiene guide - docs/CLEAN_ARCHITECTURE_GRAMMAR_ANALYSIS.md - Deep linguistic analysis