Nooa Core Engine: Validador de Gramática Arquitetural

Thiago Butignon

Janeiro 2025

Contents

1 Nooa Core Engine: Validador de Gramática Arquitetural e Guardião de Higiene de Código **2**
1.1 Resumo Executivo . . . 2
1.2 1. Introdução . . . 3
1.2.1 1.1 Motivação . . . 3
1.2.2 1.2 Abordagem . . . 3
1.2.3 1.3 Filosofia Dogfooding . . . 4
1.3 2. Fundamentação Teórica . . . 4
1.3.1 2.1 Clean Architecture como Gramática Universal . . . 4
1.3.2 2.2 Gramática BNF da Clean Architecture . . . 5
1.3.3 2.3 Regras Gramaticais como Padrões de Código . . . 6
1.4 3. Arquitetura do Nooa Core Engine . . . 6
1.4.1 3.1 Estrutura de Camadas . . . 6
1.4.2 3.2 Regras de Dependência . . . 7
1.4.3 3.3 Exemplo de Fluxo Completo . . . 8
1.5 4. Features Implementadas . . . 9
1.5.1 4.1 Validação Arquitetural (v1.0-v1.1) . . . 9
1.5.2 4.2 Higiene de Código (v1.2) . . . 12
1.6 5. Sistema de Tipos . . . 16
1.6.1 5.1 Discriminated Unions . . . 16
1.6.2 5.2 Validação de Schema YAML . . . 17
1.7 6. Resultados e Validação . . . 19
1.7.1 6.1 Auto-Validação (Dogfooding) . . . 19
1.7.2 6.2 Métricas do Projeto . . . 19
1.7.3 6.3 Performance . . . 20
1.7.4 6.4 Casos de Uso Reais . . . 21
1.8 7. Comparação com Ferramentas Existentes . . . 22
1.9 8. Trabalhos Futuros . . . 23
1.9.1 8.1 Curto Prazo (v1.3) . . . 23
1.9.2 8.2 Médio Prazo (v2.0) . . . 23
1.9.3 8.3 Longo Prazo (v3.0+) . . . 23
1.10 9. Lições Aprendidas . . . 23

1.10.1 9.1 Dogfooding como Metodologia 23
1.10.2 9.2 Restrições Geram Excelência 23
1.10.3 9.3 Type Safety vs. Flexibilidade 24
1.10.4 9.4 Falsos Positivos em Barrel Exports 24
1.11 10. Conclusão 24
1.11.1 10.1 Contribuições Principais 24
1.11.2 10.2 Impacto Prático 25
1.11.3 10.3 Visão de Futuro 25
1.12 Referências 25
1.13 Apêndices 26
1.13.1 Apêndice A: Exemplo Completo de Gramática 26
1.13.2 Apêndice B: Estrutura de Dados 28

1 Nooa Core Engine: Validador de Gramática Arquitetural e Guardião de Higiene de Código

Whitepaper Técnico v1.2

Autores: Thiago Butignon e Equipe Nooa AI **Data**: Janeiro 2025 **Versão**: 1.2.0

1.1 Resumo Executivo

Este whitepaper apresenta o **Nooa Core Engine**, uma ferramenta inovadora que trata a arquitetura de software como uma **gramática formal**, permitindo validação automática de conformidade arquitetural em projetos TypeScript. Inspirado nos princípios de Clean Architecture de Robert C. Martin e na abordagem dogmática de Rodrigo Manguinho, o Nooa transforma regras arquiteturais subjetivas em especificações formais verificáveis.

Principais Contribuições:

1. **Arquitetura como Gramática Formal**: Modelagem da Clean Architecture usando notação BNF (Backus-Naur Form), tratando camadas arquiteturais como elementos linguísticos (substantivos, verbos, advérbios).
2. **Sistema de Validação Completo**: Combinação de validação arquitetural (dependências proibidas, dependências cíclicas, dependências obrigatórias) com higiene de código (detecção de sinônimos, detecção de código morto).
3. **Dogfooding como Metodologia**: Aplicação da filosofia "usar a gramática para construir o validador da gramática", garantindo que a ferramenta seja um exemplo dogmático de Clean Architecture.

4. **Algoritmos Otimizados**: Implementação de DFS para detecção de ciclos (O(V + E)), Jaro-Winkler para similaridade de strings e análise de grafo reverso para código não referenciado.

Resultados: O Nooa Core Engine v1.2.0 valida-se a si mesmo com **zero erros arquiteturais**, demonstrando que regras formais podem ser aplicadas rigorosamente sem comprometer a produtividade.

1.2 1. Introdução

1.2.1 1.1 Motivação

Arquitetura de software é frequentemente tratada como arte, não ciência. Desenvolvedores aprendem padrões arquiteturais através de exemplos, mas falta uma forma **formal e verificável** de garantir conformidade. Problemas comuns incluem:

- **Erosão Arquitetural**: Com o tempo, a arquitetura se degrada à medida que desenvolvedores violam princípios de separação de camadas
- **Dependências Erradas**: Domain depende de Infrastructure, violando o Princípio da Inversão de Dependência
- **Duplicação Semântica**: Múltiplos desenvolvedores criam classes com nomes diferentes mas funcionalidades idênticas
- **Código Morto**: Arquivos antigos permanecem no repositório por anos sem serem usados

Pergunta de Pesquisa: *Pode a arquitetura de software ser formalizada como uma gramática, permitindo validação automática análoga a verificadores gramaticais de linguagem natural?*

1.2.2 1.2 Abordagem

Tratamos **Clean Architecture como uma linguagem formal** onde:

- **Domain** = SUBSTANTIVO (entidades, dados)
- **Use Cases** = VERBO (ações, comportamentos)
- **Infrastructure** = ADVÉRBIO (como as ações são executadas)
- **Presentation** = CONTEXTO (onde/para quem)
- **Validation** = CORRETOR GRAMATICAL
- **Main** = COMPOSITOR DE SENTENÇAS

Assim como compiladores validam sintaxe de código, o Nooa valida “sintaxe arquitetural”.

1.2.3 1.3 Filosofia Dogfooding

> **"Vamos usar a gramática para construir o validador da gramática."**

Esta não é apenas uma ideia inteligente - é o **princípio fundamental** do Nooa. A ferramenta que valida regras arquiteturais deve ser um **exemplo dogmático perfeito** da arquitetura que ela impõe. Isso cria um ciclo virtuoso:

1. Definimos regras de Clean Architecture em `nooa.grammar.yaml`
2. Implementamos features seguindo essas regras
3. Executamos Nooa contra si mesmo para verificar conformidade
4. Qualquer violação indica problema na regra ou no código
5. Corrigimos e iteramos

Resultado: `npm start .` deve sempre retornar **zero violações**.

1.3 2. Fundamentação Teórica

1.3.1 2.1 Clean Architecture como Gramática Universal

1.3.1.1 Mapeamento Linguístico

Elemento Arquitetural	Papel Linguístico	Exemplo
Domain Model	SUBSTANTIVO	`UserModel`, `Product-Model`
Use Case	VERBO TRANSITIVO	`AddAccount`, `Authenti-cate`
Data Protocol	MODIFICADOR DE VERBO	`AddAccountRepository`
Implementation	SENTENÇA ATIVA	`DbAddAccount` executa ação
Adapter	ADVÉRBIO	`BcryptAdapter` (hasheando *com bcrypt*)
Controller	CONTEXTO/VOZ	`SignUpController` (no contexto HTTP)
Factory	COMPOSITOR DE SENTENÇAS	`makeDbAddAccount` monta tudo

1.3.1.2 Analogia com Chomsky A Clean Architecture exibe propriedades da **Gramática Universal** de Chomsky:

1. **Pobreza de Estímulo**: Desenvolvedores podem gerar infinitas implementações válidas a partir de regras finitas
2. **Dispositivo de Aquisição**: Após ver 2-3 exemplos, desenvolvedores "adquirem" o padrão
3. **Estrutura Profunda vs. Superficial**: Mesma semântica arquitetural, sintaxe diferente (TypeScript vs. Python)
4. **Recursão**: Controllers compõem use cases, que compõem protocolos, infinitamente
5. **Parâmetros e Restrições**: Regras invioláveis (Domain não pode depender de Infrastructure)

1.3.2 2.2 Gramática BNF da Clean Architecture

```
<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
```

Regra de Dependência (Hierarquia de Chomsky Nível 2 - Context-Free):

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

```
/* Dependências permitidas */
<allowed> ::= Domain □ Data
            | Domain □ Presentation
            | Domain □ Infrastructure  /* PROIBIDO */
            | Data □ Infrastructure
            | Main → All

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

1.3.3 2.3 Regras Gramaticais como Padrões de Código

1.3.3.1 Regra DOM-001: Completude do Contrato de Use Case

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

Explicação Gramatical: - Interface = Assinatura do verbo (verbo transitivo requer objeto) - Params = Objeto direto (sobre o que o verbo atua) - Result = Predicado/Complemento (o que o verbo produz)

1.3.3.2 Regra DATA-001: Inversão de Dependência

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

Explicação Gramatical: - Implementação de verbo depende de advérbio abstrato, não concreto - Como definir “correr” genericamente, não “correr rapidamente”

1.4 3. Arquitetura do Nooa Core Engine

1.4.1 3.1 Estrutura de Camadas

```
src/
├── domain/          # Regras de Negócio Empresarial (camada interna)
|   ├── models/          # Entidades e tipos puros
|   |   ├── architectural-rule.model.ts
|   |   ├── architectural-violation.model.ts
|   |   └── code-symbol.model.ts
|   └── usecases/        # Interfaces de use cases (contratos)
```

```
|       └── analyze-codebase.usecase.ts (interface)
|
├── data/                # Regras de Negócio da Aplicação
|   ├── protocols/     # Definições de interfaces para infraestrutura
|   |   ├── code-parser.protocol.ts
|   |   └── grammar-repository.protocol.ts
|   └── usecases/        # Implementações de use cases
|       └── analyze-codebase.usecase.ts (implementação)
|
├── infra/               # Frameworks & Drivers (camada externa)
|   ├── parsers/         # Implementações de parsing (ts-morph)
|   |   └── ts-morph-parser.adapter.ts
|   └── repositories/    # Acesso a dados (YAML)
|       └── yaml-grammar.repository.ts
|
├── presentation/        # Adaptadores de Interface
|   └── controllers/     # Controladores CLI
|       └── cli.controller.ts
|
└── main/                # Raiz de Composição
    ├── factories/       # Factories de injeção de dependência
    |   ├── parser.factory.ts
    |   ├── repository.factory.ts
    |   └── usecase.factory.ts
    └── server.ts        # Ponto de entrada da aplicação
```

1.4.2 3.2 Regras de Dependência

Seguindo Clean Architecture, dependências apontam para dentro:

```
Main (conhece todas as camadas)
  ↓
Presentation (depende de Domain interfaces)
  ↓
Infrastructure (implementa Data protocols)
  ↓
Data (depende apenas de Domain)
  ↓
Domain (SEM dependências externas)
```

Validação: O próprio Nooa verifica suas dependências com:

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

1.4.3 3.3 Exemplo de Fluxo Completo

1. Domain (Contrato):

```typescript
// 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 (Implementação):

```typescript
// 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,  // □  Protocolo, não implement
    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 (Adaptador):

```typescript
// src/infra/parsers/ts-morph-parser.adapter.ts
```

```typescript
import { Project } from 'ts-morph';  // □  Biblioteca externa apenas na camad
import { ICodeParser } from '@/data/protocols';

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

4. Main (Composição):

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

1.5 4. Features Implementadas

1.5.1 4.1 Validação Arquitetural (v1.0-v1.1)

1.5.1.1 4.1.1 Dependências Proibidas **Sintaxe**:

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

Implementação: Verifica que nenhum símbolo com `from.role` depende de símbolos com `to.role`.

1.5.1.2 4.1.2 Detecção de Dependências Cíclicas **Algoritmo**: DFS com 3 estados (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');  // Visitando agora
    path.push(node);

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

    state.set(node, 'BLACK');  // Visitado completamente
  }

  // Visita todos os nós
  for (const node of graph.keys()) {
    if (state.get(node) === 'WHITE') {
      dfs(node, []);
    }
  }

  return cycles;
}
```

Complexidade: O(V + E)

Sintaxe:

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

1.5.1.3 4.1.3 Dependências Obrigatórias

Validação: Garante que conexões arquiteturais específicas existam.

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

Implementação:

```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 Validação de Padrões de Nomenclatura Sintaxe:

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

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

Implementação: Validação regex em tempo de parse com discriminated unions.

1.5.2 4.2 Higiene de Código (v1.2)

1.5.2.1 4.2.1 Detecção de Sinônimos Problema: Múltiplos desenvolvedores criam classes com nomes diferentes mas funcionalidade idêntica.

Solução: Algoritmo Jaro-Winkler + Normalização baseada em Thesaurus

Sintaxe:

```
- 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"
```

Algoritmo:

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

  // Comparação par-a-par
  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 {
    // Implementação do algoritmo Jaro-Winkler
    // https://en.wikipedia.org/wiki/Jaro–Winkler_distance

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

    // Peso extra para prefixos comuns (até 4 caracteres)
    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;  // Scaling factor padrão
    return jaroSimilarity + (prefixLength * p * (1 - jaroSimilarity));
  }
```

Complexidade: O($N^2 \times L^2$) onde N = símbolos, L = comprimento médio da string

Normalização:

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

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

  // Aplica thesaurus (substitui sinônimos pelo termo canônico)
  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 Detecção de Código Não Referenciado (Zombie Code)

Problema: Arquivos antigos que não são mais usados mas nunca foram deletados.

Solução: Análise de grafo reverso de dependências

Sintaxe:

```
- 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"
```

Algoritmo:

```
private detectUnreferencedCode(
```

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

  // Passo 1: Construir mapa de referências recebidas
  const incomingReferences = new Map<string, number>();

  // Inicializar todas as referências como 0
  for (const symbol of symbols) {
    incomingReferences.set(symbol.path, 0);
  }

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

  // Passo 2: Encontrar arquivos com zero referências
  for (const symbol of symbols) {
    const refCount = incomingReferences.get(symbol.path) || 0;

    // Passo 3: Filtrar por padrões de ignorar
    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;
}
```

Complexidade: O(N) onde N = total de arquivos

1.6 5. Sistema de Tipos

1.6.1 5.1 Discriminated Unions

Para segurança de tipos em tempo de compilação, usamos 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;
```

Benefícios:

- **Segurança em Tempo de Compilação**: TypeScript previne acesso a propriedades erradas
- **Type Guards**: Filtragem e narrowing adequados
- **Validação em Runtime**: Parser YAML valida estrutura

Exemplo de Type Guard:

```
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 Validação de Schema YAML

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

    // Validação de schema
    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,
      };

      // Discriminação por tipo de regra
      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. Resultados e Validação

1.7.1 6.1 Auto-Validação (Dogfooding)

O Nooa valida-se a si mesmo usando `nooa.grammar.yaml`:

```
$ npm start .

   Nooa Core Engine - Análise Arquitetural
================================================
   Analyzing project: .

   Arquitetura válida! Zero violações encontradas.

   Métricas de Performance
------------------------------------------------
    Tempo de Análise: 416ms
   Regras Acionadas: 16
   Total de Violações: 0 (11 info - falsos positivos esperados)
------------------------------------------------
```

Interpretação:

- **0 erros** - Conformidade arquitetural perfeita
- **0 warnings** - Nenhum sinônimo duplicado detectado
- **11 info** - Falsos positivos esperados de barrel exports

1.7.2 6.2 Métricas do Projeto

```
Linguagem: TypeScript
Arquitetura: Clean Architecture (abordagem Rodrigo Manguinho)
Camadas: 5 (Domain, Data, Infrastructure, Presentation, Main)

Arquivos por Camada:
- Domain:         8 arquivos (modelos de negócio puros)
- Data:           3 arquivos (implementações de use cases)
- Infrastructure: 4 arquivos (parsers ts-morph, YAML)
- Presentation:   1 arquivo (controlador CLI)
- Main:           2 arquivos (factories, entry point)
```

```
- Docs:          10 arquivos (documentação abrangente)

Total TypeScript: ~1200 linhas de código de produção
Total Documentação: ~3500 linhas de markdown
```

1.7.3 6.3 Performance

Benchmarks Empíricos de Auto-Validação:

Executamos 5 testes independentes para medir a performance real do Nooa validando-se a si mesmo:

```
Configuração do Projeto:
- Arquivos TypeScript: 22 arquivos
- Linhas de Código: 1920 linhas
- Roles Definidas: 10 roles
- Regras na Gramática: 15 regras
  - 5 regras de dependências proibidas
  - 3 regras de dependências obrigatórias
  - 3 regras de padrão de nomenclatura
  - 1 regra de detecção circular
  - 2 regras de higiene (sinônimos + zombies)
  - 1 regra de inversão de dependência

Resultados dos Benchmarks (5 execuções):
```

Execução	Análise	Tempo Total	Memória (MB)
1	454ms	810ms	220
2	583ms	900ms	194
3	403ms	650ms	220
4	474ms	720ms	216
5	398ms	660ms	221
Média	462ms	748ms	214
Desvio	±68ms	±94ms	±11

```
Performance Detalhada:
- Tempo de Análise Médio: 462ms (±14.7%)
- Tempo Total Médio: 748ms (inclui inicialização)
- Uso de Memória Médio: 214 MB
```

```
- Pico de Memória: 221 MB
- Throughput: ~47 arquivos/segundo
- Latência: ~21ms por arquivo

Resultados da Validação:
- ✅ 0 erros arquiteturais
- ✅ 0 warnings
- ℹ️ 11 info (falsos positivos de barrel exports)
```

Complexidade de Algoritmos:

Feature	Algoritmo	Complexidade	Medido
Detecção Circular	DFS com 3 estados	O(V + E)	~45ms
Dependências Obrigatórias	Travessia de grafo	O(N × M)	~120ms
Dependências Proibidas	Travessia de grafo	O(N × M × R)	~180ms
Detecção de Sinônimos	Jaro-Winkler	$O(N^2 \times L^2)$	~85ms
Código Não Referenciado	Grafo reverso	O(N)	~32ms

Análise de Performance:

- **Escalabilidade**: O tempo de análise cresce linearmente com O(V + E) para grafos de dependência
- **Memória**: Uso conservador de ~10MB por arquivo TypeScript analisado
- **CPU**: Single-threaded, mas altamente otimizado com early returns
- **I/O**: Leitura lazy de arquivos, cache em memória durante análise
- **Gargalo**: Parsing com ts-morph (~60% do tempo), validação de regras (~40%)

1.7.4 6.4 Casos de Uso Reais

1.7.4.1 Exemplo 1: Prevenindo Violações Arquiteturais Antes do Nooa:

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

Depois do Nooa:

```
❌ ERROR: Domain-Independence
domain/models/user.ts não pode depender de infra/database.ts
```

1.7.4.2 Exemplo 2: Detectando Lógica Duplicada Antes do Nooa:

- `CreateUserUseCase` (2023)
- `UserCreatorService` (2024) ❌ Duplicado!

Depois do Nooa:

```
  WARNING: Detect-Duplicate-Use-Cases
89% de similaridade entre CreateUserUseCase e UserCreatorService
Considere consolidar.
```

1.7.4.3 Exemplo 3: Encontrando Código Morto Antes do Nooa:

- Arquivos antigos permanecem no repositório por anos
- Dependências não utilizadas acumulam

Depois do Nooa:

```
  INFO: Detect-Zombie-Files
old-payment-adapter.ts não é importado em nenhum lugar
```

1.8 7. Comparação com Ferramentas Existentes

Ferramenta	Arquitetura	Higiene	Gramática	Dogfooding
Nooa	Completo	Sinônimos + Zombies	YAML formal	Auto-valida
dependency-cruiser	Dependências	Não	JSON complexo	Não
eslint-plugin-boundaries	Básico	Não	Configuração ESLint	Não
ArchUnit (Java)	Completo	Não	Código Java	Não
NDepend (.NET)	Completo	Básico	CQL	Não

Diferencial do Nooa:

1. **Gramática como Primeira Classe**: YAML declarativo, não configuração complexa
2. **Linguagem Natural**: Explicações gramaticais de violações
3. **Higiene de Código**: Além de arquitetura, detecta problemas semânticos
4. **Dogfooding**: Prova que as regras funcionam através de auto-aplicação

---

1.9 8. Trabalhos Futuros

1.9.1 8.1 Curto Prazo (v1.3)

- **Suporte Multi-Linguagem**: JavaScript, Python, Java
- **Relatórios HTML/JSON**: Visualizações ricas de violações
- **Extensão VS Code**: Validação em tempo real durante desenvolvimento
- **Presets de Configuração**: Templates para React, Node.js, etc.

1.9.2 8.2 Médio Prazo (v2.0)

- **Machine Learning**: Detecção de duplicados usando embeddings semânticos
- **Métricas de Complexidade**: Complexidade ciclomática, complexidade cognitiva
- **Rastreamento de Evolução**: Acompanhar degradação arquitetural ao longo do tempo
- **Fitness Functions**: Validação contínua de objetivos arquiteturais

1.9.3 8.3 Longo Prazo (v3.0+)

- **Validação de Microserviços**: Dependências entre serviços
- **Fronteiras de Módulos**: Enforcement de bounded contexts (DDD)
- **Sistema de Plugins**: Validadores customizados
- **ArchUnit-like DSL**: API fluente para regras programáticas

---

1.10 9. Lições Aprendidas

1.10.1 9.1 Dogfooding como Metodologia

Lição Principal: Auto-aplicação força excelência arquitetural.

Durante o desenvolvimento, dogfooding revelou:

- **Edge cases na sintaxe da gramática**: Ambiguidades foram descobertas ao escrever regras para o próprio Nooa
- **Performance bottlenecks**: Análise do próprio código mostrou ineficiências
- **Features faltando**: Necessidade de roles `_ACTUAL` para excluir barrel exports

Resultado: A ferramenta melhorou a ferramenta.

1.10.2 9.2 Restrições Geram Excelência

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

Quando Nooa reporta:

```
  src/domain/models não pode depender de src/infra
```

Não desabilitamos a regra - **refatoramos o código** para ser mais limpo.

1.10.3 9.3 Type Safety vs. Flexibilidade

Desafio: Discriminated unions em TypeScript vs. flexibilidade de YAML

Solução: Parser valida em runtime, TypeScript garante em compile-time

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

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

1.10.4 9.4 Falsos Positivos em Barrel Exports

Problema: Barrel exports (`index.ts`) causam falsos positivos em código não referenciado

Solução: Criação de roles especializadas `_ACTUAL`:

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

1.11 10. Conclusão

1.11.1 10.1 Contribuições Principais

Este trabalho demonstra que:

1. **Arquitetura pode ser formalizada**: Clean Architecture exibe propriedades de gramática formal
2. **Validação automática é viável**: Regras arquiteturais podem ser verificadas automaticamente
3. **Dogfooding funciona**: Auto-aplicação garante qualidade e credibilidade
4. **Higiene de código importa**: Além de estrutura, problemas semânticos (duplicação, código morto) degradam qualidade

1.11.2 10.2 Impacto Prático

O Nooa Core Engine permite equipes:

- **Prevenir erosão arquitetural**: Detectar violações antes de merge
- **Manter consistência**: Garantir que todos seguem os mesmos padrões
- **Reduzir dívida técnica**: Eliminar código duplicado e morto
- **Acelerar onboarding**: Novos desenvolvedores aprendem a arquitetura através das regras

1.11.3 10.3 Visão de Futuro

Objetivo: Tornar validação arquitetural tão comum quanto linting de código.

Assim como ninguém faz commit sem `npm run lint`, o objetivo é que ninguém faça commit sem `npm run arch-validate`.

Chamada à Ação:

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

Se retornar **zero violações**, sua arquitetura está saudável. Se não, o Nooa mostra exatamente o que corrigir.

1.12 Referências

1. **Martin, Robert C.** (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.
2. **Manguinho, Rodrigo**. *Clean Architecture no TypeScript*. Disponível em: 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*. Disponível em: https://martinfowler.com/bliki/Techn
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 Apêndices

1.13.1 Apêndice A: Exemplo Completo de Gramática

```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 Apêndice B: Estrutura de Dados

```typescript
// Modelos de Domain
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;
};
```

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

Documento Gerado com Clean Architecture Principles Licença: MIT **Repositório**: https://github.com/nooa-ai/nooa-core-engine

Para mais informações: - README.md - Documentação principal - docs/DOGFOODING_PHILOSO - Filosofia de auto-validação - docs/HYGIENE_RULES.md - Guia completo de higiene de código - docs/CLEAN_ARCHITECTURE_GRAMMAR_ANALYSIS.md - Análise linguística profunda