StressSpec

Design & Architecture Document

Version: 1.0.0

Target Audience: Engineers & Developers

Last Updated: November 2025

Table of Contents

Project Overview

What is StressSpec?

StressSpec is a requirements analysis tool that acts as a "wind tunnel" for requirement documents, detecting potential risks before development begins.

Key Capabilities

- **8 Risk Detection Categories**: Ambiguity, Missing Detail, Security, Conflict, Performance, Availability, Traceability, Scope
- **Multi-format Support**: Analyzes `.txt` and `.md` requirement files
- **Risk Scoring**: Calculates severity-based scores and identifies top 5 riskiest requirements
- **Multi-format Reporting**: Generates HTML, Markdown, CSV, and JSON reports
- **Web Interface**: User-friendly FastAPI-based web application
- **CLI Support**: Command-line interface for automation

System Architecture

High-Level Architecture

```
┌─────────────────────────────────────────────────────────┐
│                   Presentation Layer                     │
├──────────────────────┬──────────────────────────────────┤
│  Web UI (FastAPI)    │    CLI (main.py)                  │
│  - Templates         │    - Argument Parsing             │
│  - Static Files      │    - Progress Output              │
│  - API Endpoints     │                                   │
└──────────────────────┴───────────────────────────────────┘
           │                        │
           ▼                        ▼
┌─────────────────────────────────────────────────────────┐
│                   Service Layer                          │
├─────────────────────────────────────────────────────────┤
│  StressSpecService                                       │
│  - Orchestrates workflow                                 │
│  - Progress notifications (Observer)                     │
│  - Error handling                                        │
└─────────────────────────────────────────────────────────┘
           │
           ▼
┌─────────────────────────────────────────────────────────┐
│                   Core Processing                        │
├─────────────────────────────────────────────────────────┤
│  FileLoader → Parser → Analyzer → Scoring → Reporter    │
└─────────────────────────────────────────────────────────┘
           │
           ▼
┌─────────────────────────────────────────────────────────┐
│                   Detection Layer                        │
```

Core Components

1. Service Layer

`StressSpecService`

Location: `src/services/stress_spec_service.py`

Responsibilities:

- Orchestrates complete analysis workflow
- Coordinates file loading, parsing, analysis, and reporting
- Manages progress notifications via Observer pattern
- Provides dependency injection for testing

Key Methods:

2. File Processing

`FileLoader`

Location: `src/file_loader.py`

Responsibilities:

- Loads requirement files from disk
- Supports `.txt` and `.md` formats
- Handles file encoding and errors
- Provides structured parsing for complex formats

`RequirementParser`

Location: `src/requirement_parser.py`

Responsibilities:

3. Analysis Engine

`analyzer.py`

Location: `src/analyzer.py`

Responsibilities:

- Orchestrates risk detection across all detectors
- Applies risk filters (Chain of Responsibility)
- Handles detector errors gracefully
- Aggregates risks by requirement

Key Function:

```
analyze_requirements(
    requirements: List[Requirement],
    detectors: List[RiskDetector],
    error_handler: Optional[DetectorErrorHandler]
```

4. Risk Detection System

Detector Architecture

Base Classes:

- `RiskDetector` (ABC): Interface contract
- `BaseRiskDetector`: Common functionality and utilities

Detector Types (8 total):

1. **AmbiguityDetector**: Vague language detection
2. **MissingDetailDetector**: Incomplete requirements
3. **SecurityDetector**: Missing security requirements
4. **ConflictDetector**: Duplicate/contradictory requirements
5. **PerformanceDetector**: Missing performance specs
6. **AvailabilityDetector**: Missing uptime requirements
7. **TraceabilityDetector**: Missing IDs/test references
8. **ScopeDetector**: Scope creep detection

Each Detector:

- Inherits from `BaseRiskDetector`
- Implements `detect_risks()` method
- Uses configuration from `rules.json`
- Creates `Risk` objects via `RiskFactory`

5. Risk Scoring

`scoring.py`

Location: `src/scoring.py`

Responsibilities:

- Calculates risk scores per requirement
- Aggregates severity values (1-5 scale)
- Identifies top N riskiest requirements
- Provides ranking and prioritization

Key Functions:

- `calculate_risk_scores()`: Computes total/avg scores
- `get_top_riskiest()`: Returns top N requirements

6. Reporting System

Reporter Architecture

Base Interface:

- `Reporter`: Abstract interface
- `write(data: ReportData, output: Optional[str]) -> Path`

Report Formats:

- `MarkdownReporter`: Technical documentation
- `CSVReporter`: Spreadsheet-compatible data
- `JSONReporter`: Machine-readable format
- `HTMLReporter`: Professional presentations

Factory Pattern:

Design Patterns

1. Factory Method Pattern

`RiskDetectorFactory`

Purpose: Create detector instances dynamically

Benefits:

- Centralized detector creation
- Easy to add new detectors
- Configuration-aware (enabled/disabled)
- Caching for performance

Usage:

```
factory = RiskDetectorFactory()
detectors = factory.create_enabled_detectors()
```

2. Strategy Pattern

Detector Strategy

Purpose: Interchangeable risk detection algorithms

Implementation:

- Each detector implements `RiskDetector` interface
- Different detection strategies per category
- Runtime selection via factory

Benefits:

- Easy to add new detection strategies
- Isolated, testable components
- No coupling between detectors

3. Template Method Pattern

`BaseRiskDetector`

Purpose: Define common detection workflow

Template Steps:

1. Load configuration
2. Normalize text
3. Apply detection rules
4. Create risk objects
5. Return results

Subclasses: Override `detect_risks()` with specific logic

4. Observer Pattern

Progress Notification

Purpose: Decouple progress reporting from analysis

Components:

- `AnalysisProgressSubject`: Notifies observers
- `AnalysisProgressObserver`: Observer interface
- `ConsoleProgressObserver`: CLI output
- `SilentProgressObserver`: No output

Usage:

```
service.add_progress_observer(ConsoleProgressObserver())
```

5. Chain of Responsibility

Risk Filtering

Purpose: Flexible risk filtering pipeline

Components:

- `RiskFilter`: Base filter class
- `SeverityThresholdFilter`: Filter by severity
- `DuplicateRiskFilter`: Remove duplicates
- `CategoryFilter`: Filter by category

Usage:

```
filter_chain = SeverityThresholdFilter(
    SeverityLevel.HIGH,
    DuplicateRiskFilter()
)
```

6. Dependency Injection

Service Initialization

Purpose: Testability and flexibility

Implementation:

- Optional constructor parameters
- Default implementations provided
- Easy to mock for testing

Example:

```
service = StressSpecService(
    file_loader=MockFileLoader(),
    parser=MockParser()
)
```

Data Flow

Analysis Workflow

```
1. File Upload/CLI Input
    │
    ▼
2. FileLoader.load_file()
    └─> Returns: List[str] (raw lines)
    │
    ▼
3. RequirementParser.parse_requirements()
    └─> Returns: List[Requirement]
    │
    ▼
4. DetectorFactory.create_enabled_detectors()
    └─> Returns: List[RiskDetector]
    │
    ▼
5. analyze_requirements()
    └─> Returns: Dict[str, List[Risk]]
    │
    ▼
6. calculate_risk_scores()
    └─> Returns: Dict[str, Dict] (scores)
    │
    ▼
7. get_top_riskiest()
    └─> Returns: List[Dict] (top 5)
    │
    ▼
8. ReporterFactory.create_reporter()
    └─> Returns: Reporter instance
```

Data Models

Requirement Model

Location: `src/models/requirement.py`

```
@dataclass
class Requirement:
    id: str                     # R001, R002, etc.
    line_number: int             # Original file line
    text: str                   # Requirement text
```

Validation:

- ID cannot be empty
- Line number must be positive
- Text cannot be empty/whitespace

Risk Model

Location: `src/models/risk.py`

```python
@dataclass
class Risk:
    id: str                         # R001-AMB-001
    category: RiskCategory          # Enum: AMBIGUITY, SECURITY, etc.
    severity: SeverityLevel         # Enum: LOW(1) to BLOCKER(5)
    description: str
    requirement_id: str
    line_number: int
    evidence: str                   # Text that triggered detection
    suggestion: Optional[str]       # Fix suggestion
```

Severity Levels:

- LOW = 1
- MEDIUM = 2

Technology Stack

Backend

- **Python 3.8+**: Core language
- **FastAPI**: Web framework and API
- **Uvicorn**: ASGI server
- **Jinja2**: Template engine
- **Pydantic**: Data validation

Testing

- **pytest**: Testing framework
- **pytest-cov**: Coverage reporting

Module Structure

Directory Layout

```
StressSpec/
├── src/
│   ├── models/              # Data models (Requirement, Risk)
│   ├── detectors/           # Risk detection algorithms
│   ├── factories/           # Factory implementations
│   ├── patterns/            # Design pattern implementations
│   ├── reporting/           # Report generators
│   ├── services/            # Business logic services
│   ├── config/              # Configuration management
│   ├── utils/               # Utility functions
│   ├── analyzer.py          # Analysis orchestration
│   ├── scoring.py           # Risk scoring logic
│   └── ...
├── web/
│   ├── api/                 # FastAPI endpoints
│   ├── templates/           # HTML templates
│   ├── static/              # CSS, JS, images
│   └── main.py              # FastAPI app
```

Configuration System

Rules Configuration

Location: `data/rules.json`

Structure:

```
{
  "global_settings": {
    "case_sensitive": false
  },
  "detectors": {
    "ambiguity": {
      "enabled": true,
      "severity": "medium",
      "rules": {
        "vague_terms": {
          "keywords": ["should", "might", "could"]
        }
```

Web Architecture

FastAPI Application

Entry Point: `web/main.py`

Key Features:

- RESTful API endpoints
- Automatic API documentation (`/api/docs`)
- Static file serving
- Template rendering
- Error handling (404, 500)
- CORS middleware
- GZip compression

Error Handling

Detector Error Handling

Component: `DetectorErrorHandler`

Location: `src/utils/detector_error_handler.py`

Strategy:

- Catches exceptions from detectors
- Logs errors without breaking analysis
- Returns empty risk list on failure
- Allows other detectors to continue

Benefits:

- Resilient to individual detector failures

Extension Points

Adding New Detectors

1. **Create Detector Class:**

```python
class NewDetector(BaseRiskDetector):
    def detect_risks(self, requirement):
        # Detection logic
        pass

    def get_detector_name(self):
        return "New Detector"

    def get_category(self):
        return RiskCategory.NEW_CATEGORY
```

2. **Register in Factory:**

Adding New Report Formats

1. **Create Reporter Class:**

```python
class NewReporter(Reporter):
    def write(self, data, output=None):
        # Report generation logic
        return Path(output)
```

2. **Register in Factory:**

```python
ReporterFactory.register('new_format', NewReporter)
```

3. **Add to Enum:**

   - Update `ReportFormat` enum

Adding New Risk Filters

1. **Create Filter Class:**

```python
class NewFilter(RiskFilter):
    def _apply_filter(self, risks):
        # Filtering logic
        return filtered_risks
```

2. **Use in Chain:**

```python
filter_chain = NewFilter(SeverityThresholdFilter(...))
```

Testing Strategy

Test Structure

Location: `tests/`

Test Types:

- **Unit Tests**: Individual components
- **Integration Tests**: Component interactions
- **Acceptance Tests**: End-to-end workflows
- **Regression Tests**: Prevent regressions

Test Coverage

- Detector logic

Performance Considerations

Optimization Strategies

1. **Detector Caching:**
    - Factory caches detector instances
    - Reduces object creation overhead
2. **Lazy Loading:**
    - Configuration loaded on demand
    - Detectors created only when needed
3. **Async Processing:**
    - Web API uses background tasks
    - Non-blocking file operations

Security Considerations

Current Implementation

- File upload validation
- Path traversal protection
- File size limits
- Content type validation

Future Enhancements

- Authentication/authorization
- Rate limiting
- Input sanitization
- Secure file storage

Deployment Architecture

Development

```
python web_utils/run_web.py
```

- Single process
- Auto-reload enabled
- Debug mode

Production (Recommended)

```
uvicorn web.main:app --host 0.0.0.0 --port 8000
```

- Multiple workers
- Reverse proxy (nginx)

Future Enhancements

Planned Features

1. **Database Integration:**
    - Persistent storage for analyses
    - Historical tracking
    - User management
2. **Advanced Analytics:**
    - Trend analysis
    - Risk prediction
    - Comparative analysis

3. **Integration:**

Best Practices

Code Organization

- **Single Responsibility**: Each class has one job
- **Dependency Injection**: Testable, flexible
- **Interface Segregation**: Small, focused interfaces
- **Open/Closed**: Extensible without modification

Error Handling

- **Fail Gracefully**: Don't crash on single failures
- **Log Everything**: Comprehensive logging
- **User-Friendly Messages**: Clear error messages

Conclusion

Key Takeaways

1. **Modular Design**: Easy to extend and maintain
2. **Pattern-Based**: Well-established design patterns
3. **Testable**: Dependency injection throughout
4. **Scalable**: Can handle large requirement sets
5. **Flexible**: Multiple interfaces (CLI, Web, API)

Architecture Strengths

- Clear separation of concerns
- Extensible detector system
- Multiple output formats

Questions & Contact

Documentation

- **README.md**: User guide and quick start
- **Test Guide**: `tests/test_suite_guide.md`
- **Progress Docs**: `docs/StressSpec_Project_Progress.md`

Code Comments

- Extensive inline documentation
- Beginner-friendly explanations
- Design pattern references
- Usage examples

End of Architecture Document