TypeScript Architecture Documentation

Overview

This is a complete TypeScript rewrite of the DAF420 permit parser, maintaining 100% functional compatibility with the Python version while adding comprehensive type safety and modern TypeScript best practices.

Type System Architecture

Type Hierarchy

```
types/
├── common.ts       - Base types, enums, and utility types
├── config.ts       - Configuration-related types
├── permit.ts       - Permit and record type definitions
└── index.ts        - Central type exports
```

Key Type Features

1. **Strict Type Safety**
   - All `any` types eliminated
   - Strict null checks enabled
   - No implicit returns
   - Comprehensive interface definitions

2. **Generic Types**
   ```typescript
   type RecordData = Record<string, string | number | null | undefined>;
   type FieldType = 'str' | 'date' | 'int' | 'float';
   type ValidatorType = 'county_code' | 'app_type' | 'well_type' | ...;
   ```

3. **Discriminated Unions**
   ```typescript
   type StorageKey =
     | 'daroot'
     | 'dapermit'
     | 'dafield'
     | ...;
   ```

Module Architecture

1. Configuration Layer (`src/config/`)

Responsibilities:
- Load and parse YAML configuration
- Provide type-safe schema definitions
- Manage lookup tables
- Supply validation rules

Classes:
- `Config` : Main configuration manager with type-safe getters
- `RecordSchema` : Strongly-typed schema definition
- `FieldSpec` : Field specification with type information

Key Features:

```
class Config {
  settings: ISettings;
  schemas: Map<string, RecordSchema>;
  lookupTables: LookupTables;
  validationRules: IValidationRules;

  getSchema(recordType: string): RecordSchema | undefined;
  getLookup(tableName: string): Record<string, string>;
}
```

2. Models Layer (`src/models/`)

Responsibilities:
- Define data structures with strong typing
- Encapsulate business logic
- Provide type-safe data access

Classes:
- `Permit` : Strongly-typed permit container
- `ParseStats` : Statistics with type-safe counters
- `ParsedRecord` : Record wrapper with generic field access

Key Features:

```
class Permit implements PermitData {
  daroot: DaRootRecord | null;
  dapermit: DaPermitRecord | null;
  dafield: DaFieldRecord[];
  // ... more typed fields

  addChildRecord(storageKey: StorageKey, data: RecordData): void;
  toObject(): PermitData;
}
```

3. Validators Layer (`src/validators/`)

Responsibilities:
- Type-safe validation logic
- Separate error and warning tracking
- Support multiple validation types

Classes:
- `Validator` : Main validation engine with typed validators

Key Features:

```
class Validator {
  validate(
    validatorName: ValidatorType,
    value: string,
    context: string
  ): boolean;

  getSummary(): {
    errorCount: number;
    warningCount: number;
    errorsByType: Record<string, string[]>;
    warningsByType: Record<string, string[]>;
  };
}
```

4. Parser Layer (`src/parser/`)

Responsibilities:

- Core parsing engine with async/await
- Type-safe state machine
- Comprehensive error handling

Classes:

- `PermitParser` : Async parser with typed results

Key Features:

```
class PermitParser {
  async parseFile(inputPath: string): Promise<{
    permits: Record<string, PermitData>;
    stats: ParseStats;
  }>;

  private processLine(lineNumber: number, record: string): void;
  private routeRecord(recType: string, parsed: RecordData, lineNumber: number): void;
}
```

5. Exporter Layer (`src/exporter/`)

Responsibilities:

- Type-safe CSV export
- Async file operations
- Strongly-typed row building

Classes:

- `CSVExporter` : Async CSV writer

Key Features:

```typescript
class CSVExporter {
  async export(
    permits: Record<string, PermitData>,
    outputPath: string
  ): Promise<void>;

  private buildRow(permitNum: string, data: PermitData): CSVRow;
}
```

6. Utilities Layer (`src/utils/`)

Responsibilities:

- Type-safe conversion functions
- Reusable helper functions

Functions:

```typescript
function parseDate(value: string): string | null;
function parseInt(value: string): number | null;
function parseFloat(value: string): number | null;
function parseNumeric(value: string, validatorName?: string): number | null;
function extractField(record: string, start: number, end: number): string;
```

Data Flow

```
Input File (DAF420)
    ↓
Config.ts (Load YAML schemas)
    ↓
PermitParser.ts (Parse with type safety)
    ├→ Validator.ts (Validate fields)
    ├→ Models (Create typed instances)
    └→ State Machine (Typed state management)
    ↓
CSVExporter.ts (Export typed data)
    ↓
Output File (CSV)
```

Type Safety Examples

Before (Python - Dynamic Typing)

```python
def parse_date(value: str) -> Optional[str]:
    # Runtime type checking
    if not value:
        return None
    # ...
```

After (TypeScript - Static Typing)

```typescript
function parseDate(value: string): string | null {
  // Compile-time type checking
  if (!value) {
    return null;
  }
  // TypeScript ensures return type matches
}
```

Async/Await Pattern

TypeScript version uses modern async/await for I/O operations:

```
// Python (synchronous)
def parse_file(self, input_path: Path) -> Tuple[Dict, ParseStats]:
    with input_path.open() as f:
        for line in f:
            self._process_line(line)

// TypeScript (asynchronous)
async parseFile(inputPath: string): Promise<{
  permits: Record<string, PermitData>;
  stats: ParseStats;
}> {
  return new Promise((resolve, reject) => {
    const rl = readline.createInterface({ ... });
    rl.on('line', line => this.processLine(line));
    rl.on('close', () => resolve(results));
  });
}
```

Error Handling

Type-Safe Error Handling

```typescript
class ParseError extends Error {
  constructor(
    message: string,
    public readonly lineNumber: number,
    public readonly recordType?: string
  ) {
    super(message);
    this.name = 'ParseError';
  }
}
```

Try-Catch with Type Guards

```
try {
  const parsed = this.parseRecord(record, recType, lineNumber);
  if (parsed) {
    this.routeRecord(recType, parsed, lineNumber);
  }
} catch (error) {
  // TypeScript knows error is unknown
  this.stats.logMalformed(lineNumber, `Parse error: ${String(error)}`);
  if (this.strictMode) {
    throw error;
  }
}
```

Interface Segregation

Following SOLID principles, interfaces are segregated by concern:

```
// Configuration interfaces
interface IFieldSpec { ... }
interface IRecordSchema { ... }
interface ISettings { ... }
interface IValidationRules { ... }

// Data interfaces
interface PermitData { ... }
interface ParseStats { ... }

// Specific record interfaces
interface DaRootRecord extends RecordData { ... }
interface DaPermitRecord extends RecordData { ... }
```

Generic Programming

Using TypeScript generics for type-safe operations:

```
class ParsedRecord {
  getField<T = string | number | null | undefined>(
    fieldName: string
  ): T | undefined {
    return this.data[fieldName] as T | undefined;
  }
}

// Usage with type inference
const permitNum = record.getField<string>('permit_number');
const depth = record.getField<number>('total_depth');
```

Testing Strategy

TypeScript enables better testing through:

1. **Type Mocking**
   `typescript`

```
  const mockConfig: Config = {
    getSchema: jest.fn(),
    getLookup: jest.fn(),
    // ...
  } as unknown as Config;
```

2. **Interface Testing**
   ```
   typescript
     describe('FieldSpec', () => {
       it('should extract field correctly', () => {
         const spec: IFieldSpec = { ... };
         // Test with type safety
       });
     });
   ```

Performance Considerations

1. **Lazy Loading**: Schemas loaded once and cached
2. **Map vs Object**: Using `Map<string, T>` for O(1) lookups
3. **String Interning**: Reusing string references
4. **Async I/O**: Non-blocking file operations

Build System

TypeScript Compiler Options

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "strict": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    // ... more strict options
  }
}
```

Path Mapping

```
{
  "paths": {
    "@models/*": ["src/models/*"],
    "@config/*": ["src/config/*"],
    "@validators/*": ["src/validators/*"],
    // ...
  }
}
```

Migration Notes

Python to TypeScript Mappings

Python	TypeScript
`Dict[str, Any]`	`Record<string, unknown>` or typed interface
`List[T]`	`T[]` or `Array<T>`
`Optional[T]`	`T | null` or `T | undefined`
`@dataclass`	`class` with explicit types
`Counter`	`Map<string, number>`
`defaultdict`	`Map` with default values
`Path`	`string` (with fs module)

Key Improvements Over Python

1. **Compile-Time Safety**: Catch errors before runtime
2. **IDE Support**: Full IntelliSense and autocomplete
3. **Refactoring**: Safe automated refactoring
4. **Documentation**: Types serve as documentation
5. **Performance**: V8 engine optimizations
6. **Async/Await**: Native async support

Future Enhancements

1. **Streaming Parser**: For very large files
2. **Worker Threads**: Parallel processing
3. **WebAssembly**: Performance-critical sections
4. **CLI Improvements**: Interactive mode
5. **Type Guards**: Runtime type validation