TaesLab Arquitecture Overview

TaesLab is a MATLAB toolbox for thermoeconomic analysis of industrial energy systems. This is an object-oriented MATLAB package with strict naming conventions and validation patterns focused

Architecture Overview

Core Component Structure

- **Classes/**: Object-oriented framework with strict inheritance hierarchy
- **Base/**: High-level functions that create and manipulate core objects
- **Functions/**: Utility functions with validation and helper methods
- **Apps/**: MATLAB App Designer applications (TaesApp.mlapp, TaesTool)
- **Examples/**: Complete plant models in JSON/XLSX/MAT formats

Key Design Patterns

Inheritance Hierarchy: All classes inherit from `cTaesLab` base class which provides object validation and unique ID management:

```
cTaesLab (base) → cMessageLogger → cDataModel/cThermoeconomicModel
cTaesLab → cResultId → cExergyModel/cExergyCost/cDiagnosis
cTable (abstract) → cTableData/cTableResult → cTableCell/cTableMatrix
```

Object Validation: Every operation uses `isValid(obj)` and `isObject(obj,'cClassName')` before processing. Objects have a `status` property (true/false) that determines validity.

Message Logging: All classes inherit logging through `cMessageLogger` with standardized error/warning reporting via `cMessages` constants.

Handle Classes: All classes extend `handle` for reference semantics and unique object IDs via `cTaesLab.sequence` auto-increment.

Factory Pattern: `readModel()` automatically detects file format and creates appropriate `cReadModel` implementation.

Results Architecture: Three-layer system for computation and presentation:

- **Computation**: `cResultId` classes perform calculations
- **Organization**: `cResultInfo` collects and organizes tables
- **Presentation**: `cTable` classes handle display and export

Naming Conventions (Critical)

Class Names

- **Prefix**: All classes start with lowercase `c` (e.g., `cDataModel`, `cThermoeconomicModel`)
- **Pattern**: `c` + PascalCase descriptive name
- **Abstract Classes**: Use clear inheritance (e.g., `cReadModel` → `cReadModelJSON`)

Function Names

- **Validation Functions**: Use `is` prefix (`isValid` , `isObject` , `isFilename` , `isMatlab` , `isOctave`)
- **Flow Keys**: Must match `^[A-Z][A-Za-z0-9]{1,7}$` pattern (validated by `cParseStream.checkTextKey`)
- **State/Sample Names**: Must match `^[A-Za-z]\w{1,9}$` pattern (validated by `cParseStream.checkName`)

File Extensions and Data Models

- **Supported Formats**: JSON, XML, CSV, XLSX, MAT
- **Data Model Structure**: Each model contains ProductiveStructure, ExergyData, ResourceData sections
- **Key Validation**: Flow keys are critical identifiers validated throughout the system

Core Workflow Patterns

Creating Thermoeconomic Models

```
% Always start with data model file
model = ThermoeconomicModel('Examples/rankine/rankine_model.json');
% Check validity before use
if isValid(model)
    results = model.thermoeconomicAnalysis();
end
```

Object Validation Pattern

```
% Standard validation check used everywhere
if ~isValid(obj) || ~isObject(obj, 'cExpectedClass')
    % Log error and return
    return;
end
```

File Operations

- Use `isFilename(filename)` before any file operations
- File type detection via `cType.getFileType(filename)`
- Consistent error handling through `cMessageLogger`

Constants and Enumerations

cType Class: Central repository for all constants, enums, and validation methods:

- Flow types: `cType.FlowType.RESOURCE` , `cType.FlowType.INTERNAL` , `cType.FlowType.OUTPUT` , `cType.FlowType.WASTE`
- Process types: `cType.ProcessType.PRODUCTIVE` , `cType.ProcessType.DISSIPATIVE`
- Validation methods: `cType.checkFlowTypes()` , `cType.checkProcessTypes()`

MATLAB/Octave Compatibility

Platform Detection: Use `isMatlab()` and `isOctave()` for platform-specific code

```
if isMatlab()
    % MATLAB-specific features (App Designer, advanced table properties)
else
    % Octave fallback implementations
end
```

File Reading Interface (`cReadModel`)

TaesLab provides unified file reading through the `cReadModel` interface that handles multiple formats:

Structured Data Files

- **JSON**: `cReadModelJSON` - Direct JSON parsing with `jsondecode`
- **XML**: `cReadModelXML` - XML→JSON conversion for uniform processing

Tabular Data Files

- **XLSX**: `cReadModelXLS` - Multi-sheet Excel files
- **CSV**: `cReadModelCSV` - Multiple CSV files in directory

Configuration-Driven Tables

Table readers use `printformat.json` to define expected sheets/files and field structures:

```
{
  "datamodel": [
    {"name": "Flows", "fields": [{"name": "key"}, {"name": "type"}]},
    {"name": "Processes", "fields": [{"name": "key"}, {"name": "fuel"}, {"name":
"product"}]}
  ]
```

Data Model Structure

Unified Data Format

All file formats convert to standardized structure in `cModelData` :

1. **ProductiveStructure**: Flows (with keys/types) and Processes (with fuel/product definitions)
2. **ExergyStates**: Thermodynamic states for different operating conditions
3. **ResourcesCost**: Cost information for resource flows (optional)
4. **WasteDefinition**: Information about waste cost allocation (optional)
5. **Format**: Results format definition

Example Data Flow

```
{
  "ProductiveStructure": {
    "flows": [
      {"key": "CMB", "type": "RESOURCE"},
      {"key": "WN", "type": "OUTPUT"},
      {"key": "QC", "type": "WASTE"}
    ],
    "processes": [
      {"key": "BOIL", "fuel": "CMB", "product": "B1+B2"}
    ]
  }
}
```

Central Data Hub (`cDataModel`)

`cDataModel` acts as the **validated data warehouse** and **calculation coordinator**:

Core Responsibilities

- **Data Validation**: Ensures consistency across ProductiveStructure, ExergyData, ResourceData
- **Component Initialization**: Creates `cProductiveStructure` , `cExergyData` , `cWasteData` , etc.
- **Calculation Interface**: Provides validated data to computation modules
- **State Management**: Handles multiple operating conditions and cost samples
- **Results Container**: Inherits from `cResultSet` for data model table presentation

Key Access Patterns

```
% Get thermodynamic data for analysis
exergyData = dataModel.getExergyData('design_state');
resourceData = dataModel.getResourceData('summer_costs');

% System information
fprintf('States: %d, Flows: %d\n', dataModel.NrOfStates, dataModel.NrOfFlows);
capability = dataModel.isDiagnosis;  % Can compare states?
```

Table Presentation System (`cTable`)

Three specialized table types handle different data presentation needs:

`cTableData` - Raw Data Model Tables

- **Purpose**: Present input data (flows, processes, exergy states)
- **Usage**: Data model validation and inspection
- **Example**: Flow definitions, process fuel/product relationships

`cTableCell` - Formatted Mixed-Type Tables

- **Purpose**: Results tables with column-specific formatting
- **Features**: Mixed data types, units, field names, custom formatting per column
- **Usage**: Summary tables, process descriptions, efficiency tables

`cTableMatrix` - Matrix-Structured Results

- **Purpose**: Matrix representations (adjacency, cost allocation, FP tables)
- **Features**: Row/column totals, specialized graph options, matrix operations
- **Usage**: Flow-Process matrices, cost allocation tables, diagnosis matrices

Validation Rules

- Flow keys must be unique and follow naming pattern `^[A-Z][A-Za-z0-9]{1,7}$`
- Process fuel/product definitions use specialized syntax (validated by `cParseStream`)
- State/Sample names must match `^[A-Za-z]\w{1,9}$` pattern
- Numeric data must be non-negative where applicable

Key Utility Functions

- `readModel(filename)`: Auto-detects format and reads data model
- `isFilename()`: Validates filename patterns before file operations
- `exportMAT()`, `exportCSV()`: Standard export functions
- `cType.check*()` methods: Validation for all enum types

Application Structure

TaesApp: Main MATLAB App Designer application for GUI interaction **TaesTool**: Command-line compatible interface for MATLAB/Octave **ViewResults**: Results viewer with table/graph display capabilities

Computation and Results System

Calculation Layer Architecture

Computation Modules (`cResultId` derivatives):

- `cExergyModel` - Flow-process exergy analysis, irreversibilities, efficiencies
- `cExergyCost` - Direct and generalized cost calculations
- `cDiagnosis` - Comparative state analysis and malfunction detection
- `cWasteAnalysis` - Waste cost recycling optimization
- `cSummaryResults` - Multi-state/sample comparison tables

Results Organization (`cResultInfo`)

- **Contains**: Multiple `cTable` objects organized by analysis type
- **Provides**: Unified interface for table access, display, and export
- **Manages**: Table indexing and metadata for interactive exploration

Base Functions Workflow

All analysis functions in `Base/` follow consistent patterns:

```
function res = AnalysisFunction(cDataModel, NameValuePairs...)
    % 1. Input validation and parameter processing
    validateInput(cDataModel);

    % 2. Data extraction for specific analysis
    analysisData = cDataModel.getExergyData(state);

    % 3. Calculation module creation and computation
    calculator = cCalculationModule(analysisData);

    % 4. Results packaging
    res = calculator.buildResultInfo(formatData);

    % 5. Optional presentation/persistence
    if Show; printResults(res); end
    if SaveAs; SaveResults(res, filename); end
end
```

Development Workflow

Key Entry Points and Workflow

- **ReadDataModel.m**: File→cDataModel conversion with validation
- **ThermoeconomicModel.m**: cDataModel→cThermoeconomicModel with configuration
- *Base/Analysis.m**: Specific analysis functions (ExergyAnalysis, ThermoeconomicAnalysis, etc.)
- **ShowResults.m/SaveResults.m**: Unified presentation and persistence

Complete Analysis Pipeline

```
% 1. Load and validate data model
data = ReadDataModel('plant_model.json', 'Debug', true);

% 2. Create configured analysis engine
model = ThermoeconomicModel(data, 'CostTables', 'ALL', 'Summary', 'STATES');

% 3. Run analyses (each returns cResultInfo)
structure = model.productiveStructure();
exergy = model.exergyAnalysis();
costs = model.thermoeconomicAnalysis();

% 4. Present and save results
ShowResults(costs, 'Table', 'dcost', 'View', 'HTML');
SaveResults(exergy, 'exergy_analysis.xlsx');
```

Testing and Examples

- Use `Examples/rankine/rankine_model.json` as the canonical test model
- All analysis functions expect a valid `cDataModel` object as first parameter
- Scripts in `Scripts/` demonstrate individual analysis workflows
- Base functions provide both programmatic and interactive interfaces

When modifying this codebase:

1. Always validate inputs using existing `is*` functions
2. Follow the `c` prefix convention for new classes
3. Inherit from appropriate base classes (`cTaesLab` , `cMessageLogger`)
4. Use `cMessages` constants for error text
5. Check MATLAB/Octave compatibility for new features
6. Validate flow keys and names using `cParseStream` methods