TaesLab Class Dependencies Analysis

Analysis Date: November 2, 2025

Executive Summary

This document provides a comprehensive analysis of the class dependencies and architectural patterns in the TaesLab MATLAB toolbox for thermoeconomic analysis. The analysis reveals a sophisticated three-layer architecture with well-defined separation of concerns and a unique dual-role pattern for data management.

1. Overall Architecture

TaesLab follows a **layered architecture** with clear separation of concerns:

```
Input Files → Data Layer → Computation Layer → Results Layer → Presentation Layer
     ↓           ↓            ↓                ↓              ↓
  Base/*.m  → cDataModel → cResultId* → cResultInfo → Show/Save.m
                 ↓            ↓               ↓
           cMessageLogger → cMessageLogger → cResultSet
```

1.1 Architectural Layers

1. **Input Layer**: File reading and parsing (`cReadModel` interface)
2. **Data Layer**: Central data warehouse and validation (`cDataModel`)
3. **Computation Layer**: Analysis algorithms (`cExergyModel` , `cExergyCost` , etc.)
4. **Results Layer**: Result organization and management (`cResultInfo`)
5. **Presentation Layer**: User interfaces and output (`Base/` functions)

2. Core Inheritance Hierarchy

2.1 Base Class Foundation

```
cTaesLab (handle)
├── cMessageLogger
│   ├── cDataModel → cResultSet
│   ├── cThermoeconomicModel → cResultSet
│   ├── cResultId (abstract)
│   │   ├── cExergyModel
│   │   ├── cExergyCost
│   │   ├── cDiagnosis
│   │   ├── cSummaryResults
│   │   ├── cWasteAnalysis
│   │   ├── cProductiveStructure
│   │   └── cDiagramFP
│   ├── cResultInfo → cResultSet
│   ├── cModelResults
│   ├── cDigraphAnalysis
│   └── Various graph classes
└── Static utility classes (cType, cMessages, cParseStream)
```

2.2 Key Inheritance Patterns

All classes inherit from `cTaesLab` which provides:

- Unique object IDs via `sequence` auto-increment
- Status management (`status` property)
- Basic message printing methods (`printError` , `printWarning` , `printInfo`)

Most classes inherit from `cMessageLogger` which adds:

- Message queue management (`cQueue`)
- Detailed error logging and reporting
- Logger concatenation capabilities

Result classes inherit from `cResultSet` which provides:

- Unified presentation interface
- Consistent table access patterns
- Standard export/save functionality

3. Message Logging Architecture

3.1 cMessageLogger as Interface Layer

`cMessageLogger` serves as the **critical interface layer** between user-facing Base functions and internal object classes. This design pattern provides:

Error Propagation: Object classes accumulate errors in their message loggers without stopping execution.

User Interface Separation: Base functions handle all user interaction and error display.

Centralized Error Handling: Consistent error reporting across the entire toolbox.

3.2 Base Functions Pattern

All Base functions follow this standard pattern:

```
function res = BaseFunction(data, varargin)
    % 1. Create logger object
    res = cTaesLab();

    % 2. Validate inputs with immediate error display
    if nargin < 1
        res.printError(cMessages.NarginError, cMessages.ShowHelp);
        return
    end

    % 3. Parse parameters and handle errors
    try
        p.parse(varargin{:});
    catch err
        res.printError(err.message);
        return
    end

    % 4. Create computation objects (accumulate errors)
    model = cExergyModel(data);

    % 5. Check status and print accumulated errors
    if model.status
        res = model.buildResultInfo(data.FormatData);
    else
        model.printLogger;  % Print accumulated errors
        res.printError(cMessages.InvalidObject, class(model));
        return
    end

    % 6. Final status check
    if ~res.status
        res.printLogger;
        return
    end
end
```

3.3 Error Handling Flow

1. **Object Creation**: Classes accumulate errors via `messageLog()`
2. **Status Propagation**: `status` property reflects object validity
3. **Error Display**: Base functions call `printLogger()` to show accumulated errors
4. **User Feedback**: Consistent error messages through `cMessages` constants

4. Data Model Dual Architecture

4.1 cDataModel's Unique Pattern

`cDataModel` exhibits a sophisticated **dual architecture pattern** through inheritance from `cResultSet`:

```
cDataModel < cResultSet < cResultId < cMessageLogger < cTaesLab
```

This creates two distinct roles:

Data Warehouse Role: Primary responsibility as central data hub

- `getExergyData(state)` - Serve thermodynamic data
- `getResourceData(sample)` - Serve cost data
- `ProductiveStructure` - Serve system structure
- Data validation and state management

Result Presentation Role: Secondary capability through `cResultSet` inheritance

- `showResults()` - Display data model tables
- `exportResults()` - Export to various formats
- `getTable(name)` - Access data tables
- `saveResults()` - Persist data model

4.2 Architectural Benefits

Unified Interface: Same method patterns work across all result types:

```
ShowResults(dataModel);     % Works via cResultSet inheritance
ShowResults(exergyResults); # Works via cResultInfo→cResultSet
```

Code Reuse: Leverages all `cResultSet` functionality without reimplementation.

Consistent User Experience: Users interact with data models using familiar result patterns.

Seamless Integration: Data models participate fully in the results ecosystem.

5. Result Management System

5.1 Three-Layer Result Architecture

Computation Layer (`cResultId` derivatives):

- `cExergyModel` - Flow-process exergy analysis
- `cExergyCost` - Direct and generalized cost calculations
- `cDiagnosis` - Comparative state analysis
- `cSummaryResults` - Multi-state comparison tables

Organization Layer (`cResultInfo`):

- Contains multiple `cTable` objects
- Provides unified table access interface
- Manages table indexing and metadata

Presentation Layer (`cResultSet`):

- Abstract interface for all result containers
- Standardized display and export methods

- Consistent API across result types

5.2 Table Presentation System

Three specialized table types handle different presentation needs:

cTableData: Raw data model tables (flows, processes, states) **cTableCell**: Formatted mixed-type results with column-specific formatting **cTableMatrix**: Matrix representations (FP tables, cost allocation, adjacency)

6. Key Design Patterns

6.1 Factory Pattern

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

6.2 Template Method Pattern

All analysis functions in `Base/` follow consistent execution patterns.

6.3 Observer Pattern

Message logging system allows error accumulation and centralized reporting.

6.4 Strategy Pattern

Multiple file format readers implement common `cReadModel` interface.

6.5 Composite Pattern

Result sets contain collections of tables with unified access methods.

7. Dependencies and Relationships

7.1 Core Dependencies

cTaesLab → Foundation for all classes

- Unique ID management
- Basic status tracking
- Message creation infrastructure

cMessageLogger → Error handling layer

- Message queue management
- Error propagation and display
- Status validation

cType/cMessages → System constants

- Error codes and message templates
- File type definitions
- Table and result identifiers

7.2 Data Flow Dependencies

File Reading:

```
Input Files → cReadModel → cModelData → cDataModel
```

Analysis Workflow:

```
cDataModel → cExergyModel/cExergyCost → cResultInfo → Display/Export
```

Error Propagation:

```
Object Classes → cMessageLogger → Base Functions → User Console
```

8. Validation and Quality Assurance

8.1 Object Validation Patterns

Every operation uses standardized validation:

```
if ~isValid(obj) || ~isObject(obj, 'cExpectedClass')
    % Log error and return
    return;
end
```

8.2 Naming Conventions

Classes: Lowercase `c` prefix + PascalCase (`cDataModel` , `cExergyModel`) **Flow Keys**: `^[A-Z][A-Za-z0-9]{1,7}$` pattern **State/Sample Names**: `^[A-Za-z]\w{1,9}$` pattern **Functions**: Descriptive names with `is` prefix for validation

8.3 Platform Compatibility

MATLAB/Octave compatibility ensured through:

- `isMatlab()` and `isOctave()` detection functions
- Platform-specific implementations for advanced features
- Fallback methods for Octave limitations

9. Recommendations for Maintenance

9.1 Inheritance Cleanup

Several classes currently inherit from `cMessageLogger` that could potentially inherit directly from `cTaesLab` :

- `cDigraphAnalysis`
- `cModelResults`
- Various `cGraph*` classes

Investigation needed: Determine if these classes require message logging capabilities or could use simpler inheritance.

9.2 Error Handling Improvements

The current system is robust but could benefit from:

- Structured error codes for programmatic handling
- Error severity levels beyond current three-tier system
- Optional error callback mechanisms for GUI applications

9.3 Documentation Standards

Maintain consistent documentation patterns:

- All classes should document their inheritance chain
- Method signatures should specify parameter validation
- Examples should demonstrate error handling patterns

10. Conclusion

TaesLab demonstrates sophisticated object-oriented design with clear architectural patterns:

1. **Layered Architecture** provides excellent separation of concerns
2. **Message Logging System** creates robust error handling with clear user interfaces
3. **Dual Role Pattern** in `cDataModel` enables unified data management and presentation
4. **Consistent Inheritance Hierarchy** ensures predictable behavior and maintainability

The architecture successfully balances complexity management with feature richness, providing both novice-friendly Base functions and expert-level object interfaces. The `cMessageLogger` pattern is particularly noteworthy as an excellent example of interface layer design that keeps user-facing functions clean while enabling detailed error tracking in complex object interactions.

This analysis reveals a mature, well-architected MATLAB toolbox that successfully implements enterprise-level design patterns while maintaining the accessibility expected in scientific computing environments.