

I'll analyze the class dependencies of TaesLab by examining the inheritance hierarchy and class relationships. Let me start by examining the key classes and their dependencies.

Added 1 todo

Searched codebase for "classdef inheritance hierarchy cTaesLab cMessageLogger handle", 25 results

Now let me examine specific classes to understand their inheritance patterns:

Searched text for ^classdef.*<.* (**/Classes/*.m), 30 results

Let me get more complete inheritance information:

Searched text for ^classdef.*<.* (**/Classes/*.m), 50 results

Let me also check for classes that don't inherit from anything:

Searched text for ^classdef.*[^<]\$ (**/Classes/*.m), 20 results

Let me check the static classes that don't inherit:

Read [](file:///c%3A/Users/ctorr/Documents/Proyectos/TaesLab/Classes/cType.m#1-1), lines 1 to 5

Read [](file:///c%3A/Users/ctorr/Documents/Proyectos/TaesLab/Classes/cMessages.m#1-1), lines 1 to 5

Now let me create a comprehensive dependency analysis. Let me also check some key classes to understand their usage patterns:

Searched codebase for "class dependencies inheritance relationships cDataModel cResultId cExergyModel", 24 results

Now I'll create a comprehensive analysis of TaesLab class dependencies:

Completed (1/5) *Analyze TaesLab class dependencies*

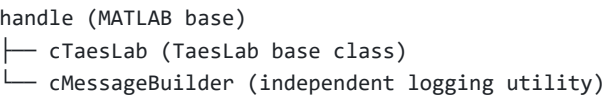
TaesLab Class Dependencies Analysis

Executive Summary

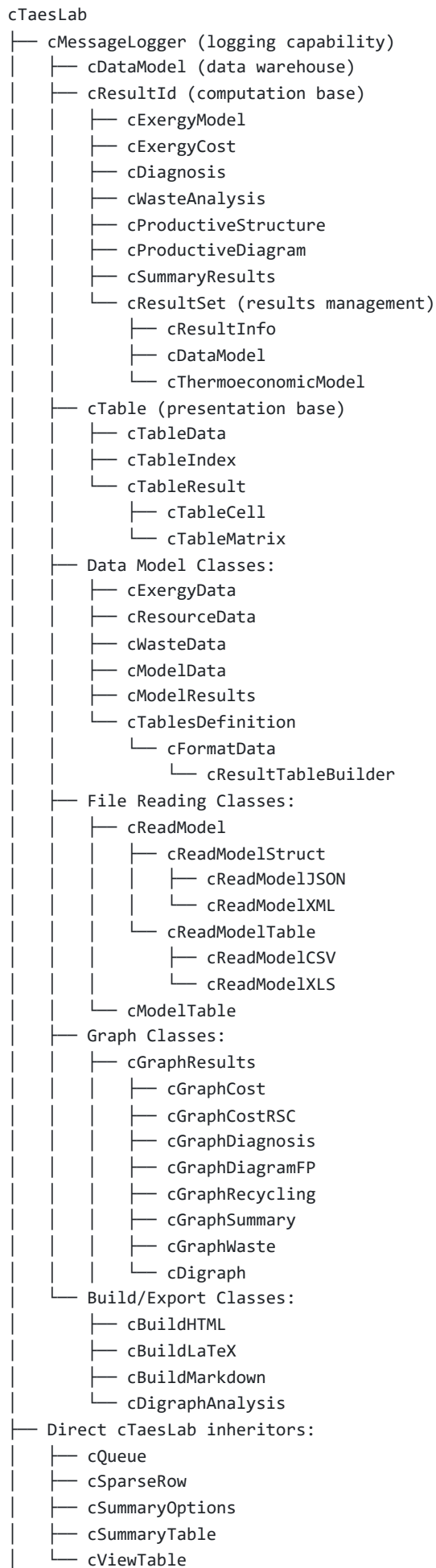
TaesLab follows a **hierarchical object-oriented architecture** with clear separation of concerns. The system is built around a three-layer design: **Data Model**, **Computation Layer**, and **Presentation Layer**, all extending from a common base class hierarchy.

Core Inheritance Hierarchy

1. Base Foundation



2. Primary Inheritance Tree



3. Static Utility Classes (no inheritance)

cType (constants and validation)
cMessages (error messages)
cParseStream (string parsing utilities)

Dependency Relationships

Core Data Flow Architecture

1. Data Input Layer

- **cReadModel → cModelData → cDataModel**
 - File format abstraction (JSON, XML, CSV, XLSX)
 - Data validation and structure conversion
 - Central data warehouse creation

2. Computation Layer

- **cDataModel → cExergyData → cExergyModel → cExergyCost**
 - Sequential computation pipeline
 - Each class builds on previous results
 - **cResultId** base provides common functionality

3. Results Management

- **cResultId → cResultSet → cResultInfo → cTable**
 - Results collection and organization
 - Multiple presentation formats
 - Unified export capabilities

Key Dependencies by Category

Data Model Dependencies

cDataModel requires:

- └─ cProductiveStructure (plant structure)
- └─ cExergyData (thermodynamic states)
- └─ cResourceData (cost information)
- └─ cWasteData (waste allocation)
- └─ cResultTableBuilder (formatting)
- └─ cModelData (raw data container)

Analysis Dependencies

cThermoeconomicModel requires:

- └─ cDataModel (data source)
- └─ cModelResults (results container)
- └─ cExergyCost (computation engine)
- └─ cSummaryResults (summary generation)
- └─ cResultTableBuilder (formatting)

Table Dependencies

cTable hierarchy requires:

- └─ cMessageLogger (error handling)
- └─ cType (constants and validation)
- └─ cFormatData (display formatting)
- └─ cGraphResults (visualization)

Architectural Patterns

1. Factory Pattern

- `readModel()` auto-detects file format
- `cResultTableBuilder` creates appropriate table types
- Graph creation based on table types

2. Observer Pattern

- Message logging throughout inheritance chain
- Status propagation from child to parent objects
- Error aggregation via `addLogger()`

3. Strategy Pattern

- File readers implement different parsing strategies
- Graph builders handle different visualization types
- Export formats use different output strategies

4. Composite Pattern

- `cResultInfo` contains multiple `cTable` objects
- `cDataModel` aggregates multiple data components
- `cThermoeconomicModel` manages multiple analysis results

5. Template Method Pattern

- `cResultId` defines computation structure
- `cTable` provides common table operations
- `cReadModel` standardizes file reading workflow

Critical Dependencies

High-Impact Classes

1. `cTaesLab` - Base class for entire system
2. `cMessageLogger` - Logging infrastructure for most classes
3. `cType` - Constants and validation used everywhere
4. `cDataModel` - Central data hub

5. `cResultTableBuilder` - Results formatting engine

Dependency Bottlenecks

1. `cType` - Static constants class used by ~90% of other classes
2. `cMessages` - Error message definitions used system-wide
3. `cMessageLogger` - Required by most functional classes
4. `cResultId` - Base for all computation classes

Circular Dependencies (Managed)

- `cDataModel` ↔ `cResultSet` (both inherit from each other's interfaces)
- `cTable` ↔ `cResultInfo` (mutual containment)
- `cExergyModel` ↔ `cResultTableBuilder` (computation ↔ formatting)

Optimization Opportunities

Classes for cTaesLab Conversion

These classes inherit from `cMessageLogger` but may not need full logging:

Candidates for `cTaesLab` conversion:

- └ `cDigraphAnalysis` (graph analysis utility)
- └ `cModelResults` (simple container)
- └ `cGraphResults` (visualization base)
- └ `cBuildHTML` (export utility)
- └ `cBuildLaTeX` (export utility)
- └ `cBuildMarkdown` (export utility)

Dependency Reduction Strategies

1. **Static utilities** could use composition instead of inheritance
2. **Export classes** could be independent of logging hierarchy
3. **Graph classes** might only need `cTaesLab` base functionality
4. **Simple containers** don't require full `cMessageLogger` overhead

System Integration Points

External Dependencies

- **MATLAB Handle Graphics** (for `cMessageBuilder`)
- **File System** (all `cReadModel` classes)
- **JSON/XML parsers** (`cReadModelJSON`, `cReadModelXML`)
- **MATLAB Table/UI components** (`cViewTable`, graph classes)

Internal Coupling

- **Tight coupling:** Data Model ↔ Computation Layer
- **Moderate coupling:** Computation ↔ Presentation Layer

- **Loose coupling:** File Reading ↔ Core System
- **No coupling:** Static utility classes

This dependency analysis reveals TaesLab's well-structured architecture with clear separation of concerns, though there are opportunities to optimize inheritance relationships for classes that don't require full logging capabilities.