TaesLab User Interface Systems Analysis

Analysis Date: November 2, 2025

Overview

TaesLab provides a sophisticated multi-layered user interface system that caters to different user types and interaction preferences. The system includes GUI applications, console interfaces, and programmatic APIs, all built around the powerful `cThermoeconomicModel` class as the central analysis engine.

1. User Interface Architecture

1.1 Interface Layers

TaesLab implements a **four-tier user interface architecture**:

```
GUI Applications (TaesApp.mlapp, TaesTool, ViewResults)
       ↓
Console Interface (ThermoeconomicModel + cThermoeconomicModel methods)
       ↓
Base Functions (Analysis functions with simplified parameters)
       ↓
Object Classes (Direct class method access)
```

1.2 User Target Groups

- **Novice Users**: GUI applications with visual parameter selection
- **Intermediate Users**: Console interface with `cThermoeconomicModel` object
- **Advanced Users**: Direct Base Function calls
- **Expert Users**: Direct class method access

2. GUI Applications Analysis

2.1 TaesApp.mlapp - Main Application

Type: MATLAB App Designer application (.mlapp file)

Purpose: Full-featured GUI for comprehensive thermoeconomic analysis

Key Features:

- Complete graphical interface for model loading and parameter selection
- Visual results display with tables and graphs
- Integrated file management (load/save in multiple formats)
- MATLAB-specific advanced UI components

Architecture: Built using MATLAB's App Designer framework, providing native MATLAB GUI experience.

2.2 TaesTool - Cross-Platform Interface

Type: MATLAB/Octave compatible GUI class

Core Architecture:

```
classdef (Sealed) TaesTool < handle
    properties(GetAccess=public,SetAccess=private)
        model   % cThermoeconomicModel object - Central analysis engine
    end
```

Key Components:

Parameter Selection Panel

- File selection and model loading
- State selection (design, off-design conditions)
- Resource sample selection (cost scenarios)
- Analysis configuration (cost tables, diagnosis methods)
- Waste analysis parameters

Results Display Panel

- Table index view showing available results
- Individual table display (console, HTML, GUI table)
- Graph visualization capabilities
- Export functionality

Classes Used by TaesTool:

- `cThermoeconomicModel` - Primary analysis engine
- `cDataModel` - Data loading and validation
- `cResultInfo` - Result containers
- `cTable` **variants** - Table display and formatting
- `cGraph` **variants** - Graphical presentations
- `cType` - Constants and enumerations
- **GUI Components** - Native MATLAB/Octave widgets

Analysis Workflow in TaesTool:

```
% 1. Load data model
data = cDataModel.create(filename);

% 2. Create analysis engine
tm = cThermoeconomicModel(data, 'Debug', app.debug);

% 3. Configure parameters via GUI
tm.setState(selectedState);
tm.setResourceSample(selectedSample);
tm.setCostTables(selectedCostType);

% 4. Get results
results = tm.thermoeconomicAnalysis();

% 5. Display via interface
app.ViewIndexTable(results);
```

2.3 ViewResults - Results Viewer

Type: MATLAB App Designer application for result visualization

Core Architecture:

```
classdef ViewResults < matlab.apps.AppBase
    properties (Access = private)
        ResultInfo       % Result Info container
        CurrentTable     % Current Table being displayed
        TableIndex       % Table Index for navigation
    end
```

Key Features:

- **Tree Navigation**: Hierarchical table selection
- **Multi-Tab Display**: Index, Tables, and Graphs tabs
- **HTML Rendering**: Advanced table formatting via `cBuildHTML`
- **Graph Integration**: Automatic graph generation for compatible tables
- **Context Menus**: Save and export options

Classes Used by ViewResults:

- **`cResultSet`** - Input validation (any result container)
- **`cTable` variants** - Table data management
- **`cBuildHTML`** - HTML rendering for table display
- **`cGraph*` classes** - Graph generation and display
- **`cTableIndex`** - Navigation structure

3. Console Interface - cThermoeconomicModel

3.1 Interactive Analysis Engine

The `cThermoeconomicModel` class serves as a **powerful console interface** that bridges the gap between simple Base Functions and complex class hierarchies.

Public Properties for User Configuration:

```
properties(Access=public)
    CurrentState        % Active operation state
    CurrentSample       % Active resource sample
    ReferenceState      % Reference state for diagnosis
    CostTables          % Cost table selection
    ActiveWaste         % Waste flow for recycling analysis
    DiagnosisMethod     % Diagnosis calculation method
    Summary             % Summary results activation
    Recycling           % Recycling analysis activation
end
```

3.2 Console Interface Workflow

Step 1: Model Creation

```
% From file
model = ThermoeconomicModel('Examples/rankine/rankine_model.json');

% Or from existing data
data = ReadDataModel('plant_model.json');
model = ThermoeconomicModel(data, 'CostTables', 'ALL', 'Summary', 'STATES');
```

Step 2: Parameter Configuration

```
% View current parameters
model.showProperties

% Configure analysis parameters
model.setState('design');
model.setReferenceState('reference');
model.setResourceSample('summer');
model.setCostTables('ALL');
model.setDiagnosisMethod('WASTE_INTERNAL');
model.setSummary('STATES');
model.setRecycling(true);
```

Step 3: Analysis Execution

```
% Individual analyses (return cResultInfo objects)
structure = model.productiveStructure();
exergy = model.exergyAnalysis();
costs = model.thermoeconomicAnalysis();
diagnosis = model.thermoeconomicDiagnosis();
waste = model.wasteAnalysis();
summary = model.summaryResults();

% Get comprehensive model results
allResults = model.getResultInfo();
```

Step 4: Results Access and Display

```
% List available tables
model.ListOfTables

% Get specific tables
costTable = model.getTable('dcost');
fpTable = model.getTable('dcfp');

% Display results
model.showResults('dcost', 'View', 'HTML');
model.showGraph('dict');

% Show all results
model.showResults();  % Console display
model.printResults(); % Alternative console display
```

Step 5: Export and Save

```
% Save individual results
model.saveResults('thermoeconomic_analysis.xlsx');
model.saveSummary('summary_results.csv');

% Export specific tables
model.exportTable('dcost', 'direct_costs.html');
```

3.3 Advanced Console Features

Model Information Methods:

```
% Check model capabilities
isResourceAvailable = model.isResourceCost;
isDiagnosisCapable = model.isDiagnosis;
hasWasteData = model.isWaste;

% Get model structure info
states = model.StateNames;
samples = model.SampleNames;
wastes = model.WasteFlows;

% Table information
tableDirectory = model.getTablesDirectory();
tableInfo = model.getTableInfo('dcost');
```

Result Management:

```
% Get result containers by type
exergyResults = model.getResultInfo(cType.ResultId.EXERGY_ANALYSIS);
costResults = model.getResultInfo('dcost');  % By table name

% Show result structure
resultStructure = model.showResultInfo();
```

4. Class Integration Patterns

4.1 Central Role of cThermoeconomicModel

All GUI applications use `cThermoeconomicModel` as their **central analysis engine**:

```
% TaesTool pattern
app.model = cThermoeconomicModel(data, 'Debug', app.debug);

% TaesPanel pattern
tm = cThermoeconomicModel(data, 'Debug', app.debug);
app.model = tm;

% Parameter synchronization
tm.setState(selectedState);
results = tm.thermoeconomicAnalysis();
app.ViewIndexTable(results);
```

4.2 Result Flow Architecture

```
cThermoeconomicModel → Analysis Methods → cResultInfo → GUI Display
         ↓                    ↓              ↓            ↓
    Parameter Storage → Computation → Table Storage → Visualization
```

4.3 Class Dependencies in GUI Applications

Core Engine Classes:

- **`cThermoeconomicModel`** - Central analysis coordinator
- **`cDataModel`** - Data source and validation
- **`cModelResults`** - Result container management
- **`cResultInfo`** - Individual analysis results

Computation Classes:

- **`cExergyModel`** - Exergy analysis calculations
- **`cExergyCost`** - Cost analysis calculations
- **`cDiagnosis`** - Comparative analysis
- **`cSummaryResults`** - Multi-state/sample comparisons

Presentation Classes:

- **`cTable*` variants** - Table formatting and display
- **`cGraph*` variants** - Graphical presentations
- **`cBuildHTML`** - HTML rendering for web display
- **`cViewTable`** - GUI table components

Utility Classes:

- **`cType`** - Constants, enumerations, validation
- **`cMessages`** - Error messages and logging
- **`cResultTableBuilder`** - Result formatting

5. User Experience Design

5.1 Progressive Complexity Model

TaesLab implements a **progressive complexity model** allowing users to choose their interaction level:

Level 1: GUI Point-and-Click

```
TaesApp        % Full GUI experience
TaesTool       % Cross-platform GUI
```

Level 2: Console with Object

```
model = ThermoeconomicModel('data.json');
model.setState('design');
results = model.thermoeconomicAnalysis();
model.showResults('dcost');
```

Level 3: Base Functions

```
data = ReadDataModel('data.json');
results = ThermoeconomicAnalysis(data, 'CostTables', 'ALL');
ShowResults(results, 'Table', 'dcost');
```

Level 4: Direct Class Access

```
data = cDataModel.create('data.json');
exd = data.getExergyData('design');
cost = cExergyCost(exd);
results = cost.buildResultInfo(data.FormatData);
```

5.2 Consistent Interface Patterns

All interfaces share common patterns:

Parameter Configuration:

- State selection (design conditions)
- Resource sample selection (cost scenarios)
- Analysis type selection (direct/generalized costs)
- Output format selection (tables/graphs)

Result Access:

- Table index navigation
- Individual table display
- Graph visualization
- Export capabilities

Error Handling:

- Consistent error messages via `cMessages`
- Status propagation through `cMessageLogger`
- User-friendly error display

6. Advanced Console Usage Examples

6.1 Interactive Analysis Session

```
% Load and configure model
model = ThermoeconomicModel('Examples/rankine/rankine_model.json', ...
    'CostTables', 'ALL', 'Summary', 'STATES');

% Inspect model capabilities
model.showProperties
fprintf('States available: %d\n', length(model.StateNames));
fprintf('Resource samples: %d\n', length(model.SampleNames));

% Compare different operating conditions
model.setState('design');
designResults = model.thermoeconomicAnalysis();

model.setState('part_load');
partLoadResults = model.thermoeconomicAnalysis();

% Diagnosis between states
model.setReferenceState('design');
model.setState('part_load');
diagnosis = model.thermoeconomicDiagnosis();

% Display comparative results
model.showResults('mfc', 'View', 'HTML');  % Malfunction costs
model.showGraph('dict');                   % Irreversibility-cost graph
```

6.2 Batch Processing Example

```
% Process multiple resource scenarios
model = ThermoeconomicModel('plant_model.json');
samples = model.SampleNames;

for i = 1:length(samples)
    model.setResourceSample(samples{i});
    results = model.thermoeconomicAnalysis();

    % Save results for each scenario
    filename = sprintf('costs_%s.xlsx', samples{i});
    saveResults(results, filename);

    % Extract key metrics
    costTable = model.getTable('dcost');
    fprintf('Scenario %s: Total cost = %.2f\n', samples{i}, ...
        sum(costTable.Data{:,2}));
end
```

6.3 Advanced Result Manipulation

```
model = ThermoeconomicModel('complex_plant.json', 'Summary', 'STATES');

% Get all available results
allResults = model.getResultInfo();
fprintf('Available tables: %d\n', allResults.NrOfTables);

% Access specific result types
exergyAnalysis = model.getResultInfo(cType.ResultId.EXERGY_ANALYSIS);
costAnalysis = model.getResultInfo(cType.ResultId.THERMOECONOMIC_ANALYSIS);

% Custom table access
fpTable = model.getTable('dcfp');      % Fuel-Product table
costTable = model.getTable('dcost');   % Process costs
flowTable = model.getTable('dfcost');  % Flow costs

% Export in different formats
model.exportTable('dcfp', 'fp_table.html');
model.exportTable('dcost', 'costs.csv');
model.saveResults('complete_analysis.xlsx');
```

7. Error Handling and Debugging

7.1 Debug Mode in Console Interface

```
% Enable debug mode for detailed information
model = ThermoeconomicModel('data.json', 'Debug', true);

% Or toggle debug on existing model
model.setDebug(true);
model.toggleDebug();  % Switch debug state

% Debug information includes:
% - Data validation details
% - Computation step information
% - Error stack traces
% - Performance timing
```

7.2 Error Handling Patterns

All interfaces use consistent error handling:

```
% Status checking
if model.status
    results = model.thermoeconomicAnalysis();
    if results.status
        model.showResults('dcost');
    else
        results.printLogger;  % Show accumulated errors
    end
else
    model.printLogger;  % Show model errors
end
```

8. Integration with MATLAB/Octave Ecosystems

8.1 MATLAB-Specific Features

- **App Designer Integration**: TaesApp.mlapp uses advanced MATLAB GUI components
- **Advanced Graphics**: Enhanced plotting and visualization
- **Toolbox Integration**: Can integrate with other MATLAB toolboxes
- **Live Scripts**: Compatible with MATLAB Live Scripts

8.2 Octave Compatibility

- **TaesTool**: Designed for cross-platform compatibility
- **Base Functions**: Work identically in both environments
- **Console Interface**: Full `cThermoeconomicModel` functionality
- **File Formats**: Consistent data model support

9. Performance and Scalability

9.1 Result Caching

`cThermoeconomicModel` implements intelligent result caching:

```
% Results are cached after first calculation
model.setState('design');
results1 = model.thermoeconomicAnalysis();  % Computation performed

% Subsequent calls return cached results
results2 = model.thermoeconomicAnalysis();  % Returns cached results

% Cache is invalidated when parameters change
model.setState('part_load');
results3 = model.thermoeconomicAnalysis();  % New computation
```

9.2 Memory Management

- **Lazy Loading**: Results computed only when requested
- **Selective Caching**: Only requested analysis types are cached

- **Memory Cleanup**: Results can be cleared when no longer needed

10. Conclusion

TaesLab's user interface system represents a sophisticated **multi-level architecture** that successfully balances ease of use with powerful functionality:

10.1 Key Strengths

1. **Progressive Learning Curve**: Users can start with GUIs and advance to console/programmatic use
2. **Consistent Architecture**: All interfaces use the same underlying `cThermoeconomicModel` engine
3. **Cross-Platform Support**: TaesTool works in both MATLAB and Octave
4. **Flexible Result Access**: Multiple ways to access and display results
5. **Extensible Design**: New analyses can be easily integrated

10.2 Architectural Benefits

- **Central Engine**: `cThermoeconomicModel` provides consistent behavior across all interfaces
- **Parameter Management**: Public properties allow easy configuration
- **Result Caching**: Intelligent caching improves performance
- **Error Handling**: Consistent error propagation and display
- **Format Support**: Multiple input/output formats supported

10.3 User Experience Success

The system successfully serves different user types:

- **Research Engineers**: GUI applications for exploration and presentation
- **Analysis Experts**: Console interface for batch processing and customization
- **Software Developers**: Base Functions and class access for integration
- **Students**: Progressive complexity allows learning at appropriate levels

This multi-tiered approach makes TaesLab accessible to users with varying levels of programming expertise while maintaining the full power of the sophisticated object-oriented architecture underneath.