Module Interface Specification for AutoVox

Team #10, Five of a Kind
Omar Abdelhamid
Daniel Maurer
Andrew Bovbel
Olivia Reich
Khalid Farag

November 13, 2025

1 Revision History

Date	Version	Notes
November 13, 2025	1.0	Initial draft by All

2 Symbols, Abbreviations and Acronyms

See SRS Documentation.

Contents

3 Introduction

The following document details the Module Interface Specifications for AutoVox. AutoVox is a desktop-based CAD enhancement tool that enables researchers and engineers to assign magnetic and material properties to individual voxels within a 3D model. The system allows users to import CAD files (STL format), automatically convert them into voxel grids, and interactively select and modify voxels layer by layer to assign magnetization directions and material assignments. This tool streamlines the magnetization planning process for multi-material 3D printing workflows in research laboratory environments.

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at https://github.com/OmarHassanAdelhamid/Five-of-a-Kind-capstone-project-

4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | ... | c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by AutoVox.

Data Type	Notation	Description
character	char	a single symbol or digit
integer	$\mathbb{Z}$	a number without a fractional component in (-∞, ∞)
natural number	$\mathbb{N}$	a number without a fractional component in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)

The specification of AutoVox uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, AutoVox uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

Level 1	**Level 2**
Hardware-Hiding Module	None
Behaviour-Hiding Module	Input Interpreter Module Voxel Slicing Module Display Partitioning Module Project Manager Module Serialization Manager Module Backend Communication Manager Module Interaction Controller Module Visualization State Manager Module Model Manager Module History Manager Module Autosave Manager Module Voxel Tracking Module Highlight Manager Module Export Validation Module Export Manager Module Error Diagnostic Handler Module
Software Decision Module	Model Structure Module Graphics Adapter Module Database Handler Module Export Structure Module

Table 1: Module Hierarchy

6 MIS of Input Interpreter

InputInterpreter

6.1 Module

The InputInterpreter module is responsible for importing and parsing model or project files (e.g., STL, JSON) and normalizing them into a consistent internal format that downstream modules can process.

6.2 Uses

- `VoxelSlicing` for voxel grid generation.
- `SerializationManager` for project deserialization.
- `ModelManager` for organizing parsed geometry data.

6.3 Syntax

6.3.1 Exported Constants

- `SUPPORTED_FORMATS`: list[string] — supported file types (STL, JSON, binary).
- `DEFAULT_SCALE`: float — default scaling applied to geometry.

6.3.2 Exported Access Programs

Name	In	Out	Exceptions
`loadFile`	filePath: string	IntermediateModel	FileNotFoundError, UnsupportedFile-TypeError
`detectFormat`	filePath: string	string	None
`parseSTL`	fileData: string	MeshModel	ParseError
`deserializeProject`	fileData: string	ProjectModel	SchemaMismatchError

6.4 Semantics

6.4.1 State Variables

- `currentModel`: IntermediateModel — latest parsed internal representation.
- `detectedFormat`: string — identifies file type.

6.4.2 Environment Variables

- `FILE_PATH`: string — path to the input file on the filesystem.

6.4.3 Assumptions

- The file path exists and is readable.
- Data structure conforms to supported file format.
- Deserialized data follows expected schema.

7 MIS of Voxel Slicing

VoxelSlicing

7.1 Module

The VoxelSlicing module is responsible for converting the geometric model into a structured voxel grid. It divides the 3D model along the Z-axis into discrete slices based on a specified resolution, mapping geometry into voxelized layers for simulation and visualization.

7.2 Uses

- `InputInterpreter` to provide intermediate geometric model data.
- `ModelManager` to receive thegenerated voxel grid for management and updates.

7.3 Syntax

7.3.1 Exported Constants

- `DEFAULT_RESOLUTION`: tuple(float, float, float) — default voxel size in x, y, and z directions.
- `MAX_LAYERS`: int — maximum number of layers allowed per model.

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
`voxelizeModel`	model: IntermediateModel, resolution: tuple	VoxelGrid	ResolutionError, GeometryError
`generateLayers`	voxelGrid: VoxelGrid	list[Layer]	None
`setResolution`	resolution: tuple	void	InvalidResolutionError

7.4 Semantics

7.4.1 State Variables

- `voxelGrid`: VoxelGrid — stores the resulting voxelized representation.
- `layerList`: list[Layer] — ordered collection of Z-level layers.
- `resolution`: tuple(float, float, float) — current voxel grid spacing.

7.4.2 Environment Variables

None

7.4.3 Assumptions

- Input model geometry is well-defined and watertight.
- Resolution values are positive real numbers.
- The number of layers does not exceed system memory capacity.

8 MIS of Display Partitioning

DisplayPartitioning

8.1 Module

The DisplayPartitioning module provides core functionality for partitioning the model into distinct display segments.

8.2 Uses

- `ModelManager` to obtain the sliced model's data representation.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

8.3 Syntax

Exported Constants

- `MAX_PARTITION`: *int* — maximum number of partitions that can be created.
- `MAX_VOXEL_PARTITION`: *int* — maximum number of voxels per partition.
- `AVAILABLE_PARTITIONS`: *List[string]* — configuration of valid partition IDs used by the display layout.

Exported Access Programs:

Name	In	Out	Exceptions
`getPartitions`	None	*dict[string]*	None
`setCurrentPartition`	id: *string* partition: *PartitionItem*	void	None
`resizePartitions`	id: *string* width: *int* height: *int* depth: *int*	void	None

8.4 Semantics

8.4.1 State Variables

- `PartitionItem`: *record* — configuration for data that describes a partition.
- `PartitionDict`: *dict[string] = PartitionItem* — tracks all partitions.

8.4.2 Environment Variables

None

8.4.3 Assumptions

- A valid and complete `ModelStructure` exists.
- Display partitions adhere only to positive dimensions.
- `AVAILABLE_PARTITIONS` provides the display layout with a preset mapping to UI grid positions.

9 MIS of Project Manager

ProjectManager

9.1 Module

The ProjectManager module manages the creation, initialization, and persistence of project workspaces, handles project metadata storage, and ensures project resources are properly allocated.

9.2 Uses

- `ModelManager` to obtain and create the model structure and data representation.
- `DisplayPartitioning` to create and manage the display partitions.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

9.3 Syntax

9.3.1 Exported Constants

None

9.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
ProjectManager	-	self	IOError
create_project	project_path: str, config: dict	None	• IOError • ValueError
load_project	project_path: str	None	• IOError • FileNot-FoundError
save_project	project_path: str	None	IOError
get_project_metadata	-	dict	-
initialize_workspace	workspace_path: str	None	IOError

9.4 Semantics

9.4.1 State Variables

- `project_path`: str - Path to the current project directory
- `workspace_root`: str - Root directory of the workspace
- `project_metadata`: dict - Dictionary containing project configuration and metadata
- `model`: ModelStructure - Reference to the model structure instance
- `is_initialized`: bool - Flag indicating if the project has been initialized

9.4.2 Environment Variables

- `FILE_SYSTEM`: The file system where project files are stored
- `WORKSPACE_ROOT`: Root directory environment variable for workspace location

9.4.3 Assumptions

- The file system has sufficient space for project creation
- The provided project path is a valid directory path
- Write permissions are available for the project directory
- The workspace root directory exists or can be created

9.4.4 Local Functions

- `validate_config(config: dict) -> bool`: Validates that the configuration dictionary contains required fields
- `create_project_structure(path: str) -> None`: Creates the directory structure for a new project
- `read_metadata_file(path: str) -> dict`: Reads and parses project metadata from file
- `write_metadata_file(path: str, metadata: dict) -> None`: Writes project metadata to file

10 MIS of Serialization Manager

SerializationManager

10.1 Module

The SerializationManager module handles conversion of all internal model data between in-memory and persistent storage forms (JSON or binary). It ensures data integrity, schema consistency, and supports both saving and restoring complete project states.

10.2 Uses

- `ModelManager` to supply the model state to be serialized.
- `DatabaseHandler` to store the serialized data persistently.
- `ExportManager` to export the encoded model structures to files.

10.3 Syntax

10.3.1 Exported Constants

- `SUPPORTED_FORMATS`: list[string] — allowed serialization formats {JSON, BIN}.
- `CURRENT_SCHEMA_VERSION`: string — defines latest project data schema.

10.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`encodeModel`	model: ModelState	string	SerializationError
`decodeModel`	data: string	ModelState	DeserializationError, SchemaMismatchError
`verifySchema`	data: string	bool	SchemaMismatchError

10.4 Semantics

10.4.1 State Variables

- `serializedData`: string — current encoded representation.
- `schemaVersion`: string — identifies schema used during last encode/decode.

10.4.2 Environment Variables

- `SAVE_PATH`: string — default file path for serialized model data.

10.4.3 Assumptions

- Input model follows internal data structure specification.
- Decoded data matches the expected schema version.
- File system paths used for storage are valid and writable.

11 MIS of Backend Communication Manager

BackendCommunicator

11.1 Module

The BackendCommunicator module manages the transfer and synchronization of data between the backend server and frontend UI.

11.2 Uses

- `ModelManager` for interpretation into backend-compatible commands.
- `SerializationManager` for necessary JSON and binary transforms.
- `InteractionController` for processing raw UI information.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

11.3 Syntax

Exported Constants

- None

Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`checkServerStatus`	None	Promise<void>	Network Error
`initDataLogs`	None	Promise<boolean>	Network Error
`getDataLogs`	None	string	None
`getInteraction`	None	UIdata: string	None

11.4 Semantics

11.4.1 State Variables

- `serverStatus`: *ServerStatusType* — tracks the current connection status of the backend server.
- `data_log`: *string* — all data gathered from the frontend that requires interpretation on the backend.

11.4.2 Environment Variables

- `SERVER_URL`: *string* — backend server URL from environment configuration.

11.4.3 Assumptions

- Backend service is reachable and operational.
- Connection is available for data synchronization.
- Valid configurations exist for necessary data transfer operations.

12 MIS of Interaction Controller

InteractionController

12.1 Module

The InteractionController module manages the process of raw events from interaction with the UI to associated actions within internal code.

12.2 Uses

- `VisualizationManager` to establish current active view.
- `BackendCommunicator` to take the collected raw UI data from the frontend and send it for interpretation.
- `ModelManager` to handle user intent related to model modification.
- `ErrorDiagnosticHandler` to handle error handling and diagnostics.

12.3 Syntax

Exported Constants

- `UI_EVENTS`: *list[string]* — identifies all supported UI events that can be detected from user interaction.
- `UI_ACTIONS`: *list[string]* — identifies all supported interactions that can be derived from UI events.

Exported Access Programs

Name	In	Out	Exceptions
`interpretEvent`	UIdata: *string*	UIEvent	None

12.4 Semantics

12.4.1 State Variables

- `currentEvent`: *UIEvent* — configuration for a UI event.
- `currentView`: *string* — identifies what view is currently displayed during interaction.
- `pointerPosition`: *[int, int]* — pointer coordinates captured from the most recent relevant event.
- `allEvents`: *list[string]* — identifies all supported UI events that can be detected from user interaction.

12.4.2 Environment Variables

None

12.4.3 Assumptions

- User intent does not exceed the scope of supported actions that can be derived.
- There can only be one current active view at a given moment.
- `pointerPosition` is contained within the screen resolution.

13 MIS of Visualization State Manager

VisualizationManager

13.1 Module

The VisualizationManager module oversees the creation of UI views while managing subsequent updates to the display state of the specified current UI views on a backend level.

13.2 Uses

- `DisplayPartitioning` for tracking different interface states across partitions.
- `GraphicsAdapter` for rendering updates visible to user.
- `ModelManager` for managing model and metadata updates that affect state visualization.
- `HighlightManager` for managing change in voxels being highlighted.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

13.3 Syntax

13.3.1 Exported Constants

- `DEFAULT_VIEWS`: *list[string]* — list of identifiers corresponding to the default views generated upon project initialization.

13.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`getCurrentView`	None	ViewState	None
`setView`	state: *ViewState*	void	None
`getHighlightState`	None	HighlightState	None
`updateModel`	voxelCoords: *List[VoxelCoord]*		
updateType: *ModelUpdateType*	void	None	
`updateHighlight`	semanticKey: *string*	void	None

13.4 Semantics

13.4.1 State Variables

- `currentView`: *string* — identifies which UI view is currently shown to the user.
- `viewStatus`: *dict[string] = ViewItem* — stores all data encapsulated within a UI view.
- `updateBundle`: *dict[string] = UpdateItem* — stores all data needed to update a UI view upon a render request.
- `pendingUpdates`: *list[UpdateKey]* — updates waiting to be processed in rendering.
- `renderStatus`: *boolean* — records the success status of the rendering process.
- `supportedViews`: *list[string]* — identifiers corresponding to all available project views.

13.4.2 Environment Variables

None

13.4.3 Assumptions

- Partitions are initialized.
- ViewItem is properly formatted in accordance with ViewItem specification.
- UpdateItem is properly formatted in accordance with UpdateItem specification.
- `currentView`, `ViewState`, and `DEFAULT_VIEWS` are elements of `supportedViews`.

14 MIS of Model Manager

ModelManager

14.1 Module

The ModelManager module oversees the management and manipulation of the voxel-based 3D model. It provides APIs to add, remove, or modify voxels, update magnetization/material properties, and synchronize changes with autosave and history tracking modules.

14.2 Uses

- `VoxelSlicing` to supply initial voxel grid data.
- `SerializationManager` to encode or decode model states for storage.
- `HistoryManager` to record changes for undo/redo.
- `VoxelTracking` to query voxel properties.
- `ErrorDiagnosticHandler` to handle error handling and diagnostics.
- `AutosaveManager` to synchronize changes with autosave.
- `HighlightManager` to manage change in voxels being highlighted.
- `ModelStructure` to define the voxel and layer data structures.
- `GraphicsAdapter` to render the model.
- `DatabaseHandler` to store the model.
- `ExportValidation` to validate the model.

14.3 Syntax

14.3.1 Exported Constants

- `MAX_VOXEL_COUNT`: int — maximum allowed voxel entries per project.
- `AUTOSAVE_INTERVAL_S`: float — time threshold (in seconds) before triggering autosave.

14.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`addVoxel`	coord: VoxelCoord, material: MaterialType, mag: Vector3	void	InvalidInputError
`removeVoxel`	coord: VoxelCoord	void	NotFoundError
`modifyVoxel`	coord: VoxelCoord, newData: VoxelData	void	NotFoundError
`saveModel`	None	bool	IOerror
`loadModel`	filePath: string	bool	DeserializationError

14.4 Semantics

14.4.1 State Variables

- `voxelGrid`: VoxelGrid — stores all voxel elements and layer mapping.
- `metadata`: dict — contains file name, author, timestamp, etc.
- `autosaveTimer`: float — time since last autosave.
- `unsavedChanges`: bool — true if edits have occurred since last save.

14.4.2 Environment Variables

- `SAVE_PATH`: string — default save location for serialized project data.

14.4.3 Assumptions

- Voxel coordinates fall within the defined model boundaries.
- Data supplied for voxel modifications are valid according to schema.
- Autosave operations will not interrupt ongoing edits.

15 MIS of History Manager

HistoryManager

15.1 Module

The HistoryManager module maintains the complete change history of the 3D voxel model. It allows undoing and redoing edits by recording incremental changes (deltas) after every modification.

15.2 Uses

- `ModelManager` to provide the model changes to record.
- `SerializationManager` to encode snapshots for persistence.
- `AutosaveManager` to reference latest history state during periodic saves.
- `ErrorDiagnosticHandler` to handle error handling and diagnostics.

15.3 Syntax

15.3.1 Exported Constants

- `MAX_HISTORY_SIZE`: int — maximum number of undo states stored.

15.3.2 Exported Access Programs

Name	In	Out	Exceptions
`recordChange`	delta: ModelDelta	void	None
`undo`	None	bool	EmptyHistoryError
`redo`	None	bool	EmptyHistoryError
`clearHistory`	None	void	None

15.4 Semantics

15.4.1 State Variables

- `historyStack`: list[ModelState] — sequence of previous model states.
- `redoStack`: list[ModelState] — sequence of undone model states available for redo.
- `currentIndex`: int — current pointer in the history sequence.

15.4.2 Environment Variables

None

15.4.3 Assumptions

- Changes are discrete and atomic.
- No new change is made while undo or redo is in progress.
- The number of stored states does not exceed `MAX_HISTORY_SIZE`.

16 MIS of Autosave Manager

AutosaveManager

16.1 Module

The AutosaveManager module prepares data to be saved, monitors when updates require saving, and invokes periodic scheduled transfers to file-based storage.

16.2 Uses

- `ModelManager` for tracking changes that require savings.
- `SerializationManager` for data preparation.
- `HistoryManager` for logging recent changes.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

16.3 Syntax

Exported Constants

- `AUTOSAVE_INTERVAL_MS`: *int* — minimum amount of time (in milliseconds) between autosave operations.
- `FILE_ID`: *string* — file to which autosave operations are written.

Exported Access Programs

Name	In	Out	Exceptions
`forceSave`	None	None	IO Error
`enableAutosave`	None	None	None
`disableAutosave`	None	None	None
`lastAutosaveTime`	None	DateTime	None

16.4 Semantics

16.4.1 State Variables

- `autosavePermission`: *bool* — indicates whether autosave is enabled or disabled.
- `lastAutosave`: *DateTime* — timestamp of last transfer to file-based storage.
- `autosaveHistory`: *dict[DateTime] = Update* — complete log of transfers to file-based storage.

16.4.2 Environment Variables

- `SYSTEM_TIME`: *DateTime* — global time supplied by the current system.

16.4.3 Assumptions

- `SYSTEM_TIME` always increases monotonically.
- Data preparation method is compatible with file-based storage.
- Connection is available to file-based storage.
- Valid configuration exists for operations to file-based storage.

17 MIS of Voxel Tracking

VoxelTracking

17.1 Module

The VoxelTracking module interprets the voxel data structure to locate and track voxels that satisfy particular property criteria, such as selection state, material type, or user-defined rules. It efficiently identifies and accesses relevant voxels to determine which voxels are currently subject to operations like highlighting, selection, or further processing.

17.2 Uses

- `ModelStructure` to access the voxel grid data structure.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

17.3 Syntax

17.3.1 Exported Constants

- `MAX_QUERY_RESULTS`: int — maximum number of voxels returned in a single query result.
- `SUPPORTED_PROPERTIES`: list[string] — list of property names that can be queried (e.g., ”material”, ”magnetization”, ”selected”, ”layer”).

17.3.2 Exported Access Programs

Name	In	Out	Exceptions
`queryByMaterial`	materialType: MaterialType	list[VoxelCoord]	InvalidMaterialError
`queryBySelection`	isSelected: bool	list[VoxelCoord]	None
`queryByRegion`	bounds: BoundingBox	list[VoxelCoord]	InvalidBoundsError
`queryByLayer`	layerZ: int	list[VoxelCoord]	IndexError
`queryByProperty`	propertyName: string, value: Any	list[VoxelCoord]	InvalidPropertyError
`getVoxelProperties`	coord: VoxelCoord	VoxelProperties	NotFoundError

17.4 Semantics

17.4.1 State Variables

- `queryCache`: dict[string, list[VoxelCoord]] — cached results of recent queries for performance optimization.
- `activeSelections`: set[VoxelCoord] — set of currently selected voxel coordinates.

17.4.2 Environment Variables

None

17.4.3 Assumptions

- Voxel coordinates provided in queries are within valid grid boundaries.
- Property names used in queries match those defined in `SUPPORTED_PROPERTIES`.
- The voxel grid structure remains consistent during query operations.

18 MIS of Highlight Manager

HighlightManager

18.1 Module

The HighlightManager module manages the voxel highlights in accordance with visual semantic meaning.

18.2 Uses

- `Visualization State Manager` for managing visual updates related to voxel highlights.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

18.3 Syntax

Exported Constants

- `DEFAULT_HIGHLIGHT_MAP`: *dict[string, string]* — mapping from semantic keys to their default highlight colours.

Exported Access Programs

Name	In	Out	Exceptions
`getHighlight`	semanticKey: *string*	ColourValue	None
`editPalette`	semanticKey: *string* newColour: *ColourValue*	void	None
`setHighlight`	semanticKey: *string*	void	None
`resetPalette`	None	void	None

18.4 Semantics

18.4.1 State Variables

- `highlightColourMap`: *dict[string, string]* — current mapping from semantic keys to active highlight colours.

18.4.2 Environment Variables

None

18.4.3 Assumptions

- Semantic keys must correspond to valid entries maintained by the TrackingManager.
- Colour values are valid colour representations.

19 MIS of Export Validation

ExportValidation

19.1 Module

The ExportValidation module validates export readiness by checking export file format requirements, verifying completeness of voxel properties (material and magnetization), and ensuring all export constraints are met.

19.2 Uses

- `ModelStructure` to obtain the model structure and data representation.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

19.3 Syntax

19.3.1 Exported Constants

- `MAX_VOXELS`: $\mathbb{Z} = 13996800000$ - Maximum number of voxels allowed
- `MAX_LAYERS`: $\mathbb{Z} = 518400$ - Maximum number of layers allowed

19.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
validate_export_readiness	model: ModelStructure	ValidationResult	-
check_printer_compatibility	model: ModelStructure	bool	-
check_property_completeness	model: ModelStructure	list[str]	-
validate_file_format	file_path: str	bool	IOError

19.4 Semantics

19.4.1 State Variables

None

19.4.2 Environment Variables

None

19.4.3 Assumptions

- The model structure provided is valid and properly initialized
- All voxels in the model have consistent property structures
- File paths provided are accessible and readable

19.4.4 Local Functions

- `count_voxels(model: ModelStructure) -> ℤ`: Counts the total number of voxels in the model
- `count_layers(model: ModelStructure) -> ℤ`: Counts the total number of layers in the model
- `check_printer_specs(model: ModelStructure) -> bool`: Validates model against printer-specific constraints

20 MIS of Export Manager

ExportManager

20.1 Module

The ExportManager module coordinates the export process, transforms internal model data into export-compatible formats, and serializes project data including voxel grids, metadata, material properties, and magnetization information according to export specifications.

20.2 Uses

- `ModelStructure` to obtain the model structure and data representation.
- `ExportValidation` to get the export validation result.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

20.3 Syntax

20.3.1 Exported Constants

- `DEFAULT_EXPORT_FORMAT`: str = ”CSV” - Default export file format

20.3.2 Exported Access Programs

Name	In	Out	Exceptions
ExportManager	-	self	-
export_project	model: ModelStructure, export_path: str, format: str	None	• IOError • ValueError
transform_to_export_format	model: ModelStructure, format: str	ExportData	-
serialize_data	data: ExportData, format: str	str	ValueError

20.4 Semantics

20.4.1 State Variables

- `export_format`: str - Current export format being used

- `export_config`: dict - Configuration settings for export operations

20.4.2 Environment Variables

- `FILE_SYSTEM`: The file system where export files are written

20.4.3 Assumptions

- The model structure provided has been validated for export readiness
- The export path directory exists or can be created
- Write permissions are available for the export directory
- The export format is supported

20.4.4 Local Functions

- `group_voxels_by_layer(model: ModelStructure) -> dict`: Groups voxels by their layer Z-coordinate
- `encode_csv(export_data: ExportData) -> str`: Encodes export data into CSV format string
- `validate_export_path(path: str) -> bool`: Validates that the export path is writable
- `create_export_directory(path: str) -> None`: Creates the export directory if it does not exist

21 MIS of Error Diagnostic Handler

ErrorDiagnosticHandler

21.1 Module

The ErrorDiagnosticHandler module detects, diagnoses, and handles errors that occur during model operations, graphics rendering, and file interactions. It provides error classification, logging, and recovery mechanisms to ensure system stability and user feedback.

21.2 Uses

None

21.3 Syntax

21.3.1 Exported Constants

- `ERROR_MODEL_UNRESPONSIVE`: str = "MODEL_UNRESPONSIVE" - Error code for unresponsive model
- `ERROR_GRAPHICS_UPDATE`: str = "GRAPHICS_UPDATE_FAILURE" - Error code for graphics update failure
- `ERROR_FILE_ISSUE`: str = "FILE_OPERATION_ERROR" - Error code for file operation errors

21.3.2 Exported Access Programs

Name	In	Out	Exceptions
detect_error	error_context: dict	ErrorDiagnostic	-
classify_error	error_code: str, error_context: dict	ErrorType	-
log_error	error_diagnostic: ErrorDiagnostic	None	IOError
handle_error_recovery	error_diagnostic: ErrorDiagnostic	RecoveryAction	-
get_error_source	error_diagnostic: ErrorDiagnostic	str	-

21.4 Semantics

21.4.1 State Variables

- `error_log`: list[ErrorDiagnostic] - History of detected errors
- `error_patterns`: dict - Patterns for error classification

21.4.2 Environment Variables

- `LOG_FILE`: File system location for error logging

21.4.3 Assumptions

- Error context dictionaries contain sufficient information for diagnosis
- Log file location is writable
- Error codes follow the defined constants

21.4.4 Local Functions

- `analyze_error_pattern(error_context: dict) -> str`: Analyzes error context to identify error patterns
- `determine_error_source(context: dict) -> str`: Determines the source component from error context
- `format_error_message(error_code: str, context: dict) -> str`: Formats a human-readable error message
- `suggest_recovery_steps(error_type: ErrorType) -> list[str]`: Generates recovery step suggestions based on error type

22 MIS of Model Structure

ModelStructure

22.1 Module

The ModelStructure module stores and organizes all voxel and layer data for the model, including per-voxel properties, metadata, and structural dimensions.

22.2 Uses

- `ModelManager` for managing the model structure and data representation.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

22.3 Syntax

22.3.1 Exported Constants

- `VOXEL_SIZE_XY`: $\mathbb{R} = 300.0$ - Voxel size in X and Y dimensions (micrometers)
- `VOXEL_SIZE_Z`: $\mathbb{R} = 110.0$ - Voxel size in Z dimension (micrometers)

22.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
ModelStructure	dimensions: tuple[ℤ, ℤ, ℤ]	self	ValueError
get_voxel	position: tuple[ℤ, ℤ, ℤ]	Voxel	IndexError
set_voxel	position: tuple[ℤ, ℤ, ℤ], voxel: Voxel	None	IndexError
get_layer	z: ℤ	Layer	IndexError
get_property	position: tuple[ℤ, ℤ, ℤ], property_name: str	Any	• IndexError • KeyError
set_property	position: tuple[ℤ, ℤ, ℤ], property_name: str, value: Any	None	• IndexError • ValueError
add_layer	layer: Layer	None	ValueError
get_metadata	-	dict	-
set_metadata	key: str, value: Any	None	-
validate_voxel	voxel: Voxel	bool	-
has_material	voxel: Voxel	bool	-
has_magnetization	voxel: Voxel	bool	-

22.4 Semantics

22.4.1 State Variables

- `voxel_grid`: 3D array[Voxel] - Three-dimensional grid storing voxel data
- `layers`: list[Layer] - Ordered list of layers, indexed by Z-coordinate
- `metadata`: dict - Dictionary containing model metadata (dimensions, material properties, etc.)
- `dimensions`: tuple[ℤ, ℤ, ℤ] - Grid dimensions (X, Y, Z)

22.4.2 Environment Variables

None

22.4.3 Assumptions

- Voxel positions are within the grid boundaries defined by `dimensions`
- Layer ordering follows Z-axis ordering (bottom to top)
- Property names are consistent across voxels
- Material types and magnetization vectors conform to expected formats

22.4.4 Local Functions

- `validate_position(position: tuple[ℤ, ℤ, ℤ]) -> bool`: Checks if position is within grid boundaries
- `maintain_layer_ordering() -> None`: Ensures layers remain ordered by Z-coordinate
- `create_layer_from_z(z: ℤ) -> Layer`: Creates a new layer structure for a given Z-coordinate
- `validate_property_value(property_name: str, value: Any) -> bool`: Validates that a property value conforms to expected type and constraints

23 MIS of Graphics Adapter

GraphicsAdapter

23.1 Module

GraphicsAdapter handles all communication with the graphics API to enable visual rendering and generate a model on the UI.

23.2 Uses

- `VisualizationManager` aggregates necessary backend data and relays updates to the rendered view.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

23.3 Syntax

23.3.1 Exported Constants

- None

23.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`requestRender`	str: *UpdateKey*		
update: *UpdateBundle*	void	NetworkError	
`getUpdateStatus`	str: *UpdateKey*	boolean	None
`checkServerStatus`	None	Promise<void>	NetworkError

23.4 Semantics

23.4.1 State Variables

- `updateStatus`: *boolean* — tracks status of rendering completion.

23.4.2 Environment Variables

- `API_BASE_URL`: *string* — external base URL for establishing environment configurations and enabling backend API requests.

23.4.3 Assumptions

- Backend service is reachable and operational.
- Connection is available for API.
- Valid configurations exist for necessary API operations.

24 MIS of Database Handler

DatabaseHandler

24.1 Module

The DatabaseHandler module manages persistent storage for project and model data. It provides standardized interfaces for reading, writing, and deleting serialized model files, ensuring version integrity and consistent access to saved projects.

24.2 Uses

- `SerializationManager` to encode and decode model data before storage.
- `ModelManager` to provide model structures to persist.
- `ErrorDiagnosticHandler` to handle error handling and diagnostics.

24.3 Syntax

24.3.1 Exported Constants

- `DEFAULT_SAVE_DIR`: string — default directory path for project files.
- `DB_FORMAT_VERSION`: string — current format version for data compatibility.

24.3.2 Exported Access Programs

Name	In	Out	Exceptions
`saveProject`	id: string, data: SerializedModel	bool	IOError
`loadProject`	id: string	SerializedModel	FileNotFoundError
`deleteProject`	id: string	bool	IOError
`listProjects`	None	list[string]	None

24.4 Semantics

24.4.1 State Variables

- `projectMap`: dict[string, SerializedModel] — mapping of project identifiers to serialized data.
- `storagePath`: string — root directory for persistent project files.

24.4.2 Environment Variables

- `FILE_SYSTEM`: OS-level file system used for reading/writing project data.

24.4.3 Assumptions

- Project IDs are unique.
- File I/O operations succeed given sufficient permissions.
- SerializationManager ensures schema compatibility before writing data.

25 MIS of Export Structure

ExportStructure

25.1 Module

The ExportStructure module defines and implements the internal data structure for representing exported files, including field ordering, data types, and encoding format. It ensures consistency between internal model representations and external file layouts used during export operations.

25.2 Uses

- `ExportManager` to provide the export data structure schema.
- `ErrorDiagnosticHandler` for error handling and diagnostics.

25.3 Syntax

25.3.1 Exported Constants

- `EXPORT_SCHEMA_VERSION`: string — current version of the export structure schema.
- `FIELD_ORDER`: list[string] — ordered list of field names in export format: [”x”, ”y”, ”z”, ”layer”, ”material_id”, ”magnetization_x”, ”magnetization_y”, ”magnetization_z”].
- `DEFAULT_ENCODING`: string — default character encoding for export files (e.g., ”UTF-8”).

25.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`defineStructure`	format: string	ExportSchema	UnsupportedFormatError
`getFieldOrder`	format: string	list[string]	UnsupportedFormatError
`getFieldType`	fieldName: string	DataType	InvalidFieldError
`validateStructure`	data: ExportData	bool	StructureMismatchError
`createHeader`	metadata: dict	ExportHeader	None

25.4 Semantics

25.4.1 State Variables

- `exportSchema`: ExportSchema — current export structure schema definition.
- `fieldTypes`: dict[string, DataType] — mapping from field names to their data types.

25.4.2 Environment Variables

None

25.4.3 Assumptions

- Export format identifiers match supported formats (e.g., ”CSV”, ”JSON”).
- Field names in export data match those defined in `FIELD_ORDER`.
- Data types conform to the schema defined by `fieldTypes`.
- Export structure remains consistent across export operations.

25.4.4 Local Functions

- `mapInternalToExport(internalData: VoxelData) -> ExportData`: Maps internal voxel representation to export format structure.
- `validateFieldValue(fieldName: string, value: Any) -> bool`: Validates that a field value matches its expected data type.
- `encodeFieldValue(fieldName: string, value: Any) -> string`: Encodes a field value according to the export format specification.

References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

26 Appendix

[Extra information if required —SS]

Appendix — Reflection

[Not required for CAS 741 projects —SS]

The information in this section will be used to evaluate the team members on the graduate attribute of Problem Analysis and Design.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?
2. What pain points did you experience during this deliverable, and how did you resolve them?
3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?
4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), it any, needed to be changed, and why?
5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)
6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)