Module Guide for Software Engineering

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

November 6, 2025

1 Revision History

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

2 Reference Material

This section records information for easy reference.

2.1 Abbreviations and Acronyms

symbol	description
AC	Anticipated Change
DAG	Directed Acyclic Graph
M	Module
MG	Module Guide
OS	Operating System
R	Requirement
SC	Scientific Computing
SRS	Software Requirements Specification
Software Engineering	Explanation of program name
UC	Unlikely Change
UI	User Interface
3D	Three Dimensional
API	Application Programming Interface

Contents

List of Tables

List of Figures

3 Introduction

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team (**?**). We advocate a decomposition based on the principle of information hiding (**?**). This principle supports design for change, because the "secrets" that each module hides represent likely future changes. Design for change is valuable in SC, where modifications are frequent, especially during initial development as the solution space is explored.

Our design follows the rules layed out by **?**, as follows:

- System details that are likely to change independently should be the secrets of separate modules.
- Each data structure is implemented in only one module.
- Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module.

After completing the first stage of the design, the Software Requirements Specification (SRS), the Module Guide (MG) is developed (**?**). The MG specifies the modular structure of the system and is intended to allow both designers and maintainers to easily identify the parts of the software. The potential readers of this document are as follows:

- New project members: This document can be a guide for a new project member to easily understand the overall structure and quickly find the relevant modules they are searching for.
- Maintainers: The hierarchical structure of the module guide improves the maintainers' understanding when they need to make changes to the system. It is important for a maintainer to update the relevant sections of the document after changes have been made.
- Designers: Once the module guide has been written, it can be used to check for consistency, feasibility, and flexibility. Designers can verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

The rest of the document is organized as follows. Section 4 lists the anticipated and unlikely changes of the software requirements. Section 5 summarizes the module decomposition that was constructed according to the likely changes. Section 6 specifies the connections between the software requirements and the modules. Section 7 gives a detailed description of the modules. Section 8 includes two traceability matrices. One checks the completeness of the design against the requirements provided in the SRS. The other shows the relation between anticipated changes and the modules. Section 9 describes the use relation between modules.

4 Anticipated and Unlikely Changes

This section lists possible changes to the system. According to the likeliness of the change, the possible changes are classified into two categories. Anticipated changes are listed in Section 4.1, and unlikely changes are listed in Section 4.2.

4.1 Anticipated Changes

Anticipated changes are the source of the information that is to be hidden inside the modules. Ideally, changing one of the anticipated changes will only require changing the one module that hides the associated decision. The approach adapted here is called design for change.

AC1: The specific hardware on which the software is running.

AC2: The format of the initial input data.

...

[Anticipated changes relate to changes that would be made in requirements, design or implementation choices. They are not related to changes that are made at run-time, like the values of parameters. —SS]

4.2 Unlikely Changes

The module design should be as general as possible. However, a general system is more complex. Sometimes this complexity is not necessary. Fixing some design decisions at the system architecture stage can simplify the software design. If these decision should later need to be changed, then many parts of the design will potentially need to be modified. Hence, it is not intended that these decisions will be changed.

UC1: Input/Output devices (Input: File and/or Keyboard, Output: File, Memory, and/or Screen).

...

5 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets in Table 1. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

M1: STL Interpreter Module

M2: Voxel Slicing Module

M3: Display Partitioning Module

M4: Project Manager Module

M5: Serialization Manager Module

M6: Backend Communication Manager Module

M7: Interaction Controller Module

M8: Visualization State Manager Module

M9: Model Manager Module

M10: History Manager Module

M11: Autosave Manager Module

M12: Voxel Tracking Module

M13: Highlight Manager Module

M14: Export Validation Module

M15: Export Manager Module

M16: Error Diagnostic Handler Module

M17: Model Structure Module

M18: Graphics Adapter Module

M19: Database Handler Module

M20: Export Structure Module

6 Connection Between Requirements and Design

The design of the system is intended to satisfy the requirements developed in the SRS. In this stage, the system is decomposed into modules. The connection between requirements and modules is listed in Table 2.

[The intention of this section is to document decisions that are made "between" the requirements and the design. To satisfy some requirements, design decisions need to be made. Rather than make these decisions implicit, they are explicitly recorded here. For instance, if a program has security requirements, a specific design decision may be made to satisfy those requirements with a password. —SS]

Level 1	Level 2
Hardware-Hiding Module	None
Behaviour-Hiding Module	STL 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

7 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by **?**. The *Secrets* field in a module decomposition is a brief statement of the design decision hidden by the module. The *Services* field specifies *what* the module will do without documenting *how* to do it. For each module, a suggestion for the implementing software is given under the *Implemented By* title. If the entry is *OS*, this means that the module is provided by the operating system or by standard programming language libraries. *Software Engineering* means the module will be implemented by the Software Engineering software.

Only the leaf modules in the hierarchy have to be implemented. If a dash (–) is shown, this means that the module is not a leaf and will not have to be implemented.

7.1 Hardware Hiding Modules (M??)

This system has no hardware components.

7.2 Behaviour-Hiding Module

7.2.1 STL Interpreter Module (M1)

Secrets:

Services:

Implemented By: AutoVox

Type of Module: [Record, Library, Abstract Object, or Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

7.2.2 Voxel Slicing Module (M2)

Secrets:

Services:

Implemented By: AutoVox

Type of Module: [Record, Library, Abstract Object, or Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

7.2.3 Display Partitioning Module (M3)

Secrets: How to determine the boundaries that define model partitions. How to define the relationship between display segments. How to represent partition boundaries. How to minimize the number of partitions created. How to ensure that display partitions maintain balanced spatial proportions. How to maintain partition stability during model modification. How to validate that a partition of voxels is viable to display within the UI.

Services: Provide functionality for partitioning the model into distinct display segments defined by specific voxel groupings. Manages the relationships and alignment between display segments.

Implemented By: AutoVox

Type of Module: Abstract Object

7.2.4 Project Manager Module (M4)

Secrets:

Services:

Implemented By: AutoVox

Type of Module: [Record, Library, Abstract Object, or Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

7.2.5 Serialization Manager Module (M5)

Secrets:

Services:

Implemented By: AutoVox

Type of Module: [Record, Library, Abstract Object, or Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

7.2.6 Backend Communication Manager Module (M6)

Secrets: How the connection between the backend and frontend is established. How data is transformed, validated and transferred between the backend and frontend. How synchronization errors are detected during communication. How failed communication attempts are detected and handled.

Services: Handles communication between the backend server and the frontend UI. Ensures synchronization of model data throughout the system.

Implemented By: AutoVox

Type of Module: Abstract Object

7.2.7 Interaction Controller Module (M7)

Secrets: How events are internally represented. How to differentiate between similar events. How to define the rules that associate high-level action with a raw event. How to handle raw events where user intent is inconclusive. How to determine when multiple events should be interpreted as a single user action. How to apply context from UI when interpreting events.

Services: Maps raw events from interaction with UI to high-level system actions that reflect user intent.

Implemented By: AutoVox

Type of Module: Abstract Object

7.2.8 Visualization State Manager Module (M8)

Secrets: How changes are grouped and aggregated. How the unified state-change representation is structured. How a state change is validated as complete. How event propagation from multiple sources is tracked concurrently. How conditions that trigger a UI update are determined. How the order of UI updates is established. How conflicting state changes are resolved. How invalid or irrelevant data-change notifications are handled.

Services: Aggregates data-change notifications from event propagation into a unified state-change representation. Triggers UI update used the derived state differences.

Implemented By: AutoVox

Type of Module: Abstract Object

7.2.9 Model Manager Module (M9)

Secrets:

Services:

Implemented By: AutoVox

Type of Module: [Record, Library, Abstract Object, or Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

7.2.10 History Manager Module (M10)

Secrets:

Services:

Implemented By: AutoVox

Type of Module: [Record, Library, Abstract Object, or Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

7.2.11 Autosave Manager Module (M11)

Secrets: How to collect and aggregate metrics to represent the current status of update operations. How to handle errors in metric collection. How multiple metrics are tracked concurrently. How the state of update execution is defined from collected metric data. How autosave execution is triggered. How autosave failures or interruptions are detected and handled.

Services: Maintains real-time autosave status. Tracks metrics that are used to determine the current state of update execution.

Implemented By: AutoVox

Type of Module: Abstract Object

7.2.12 Voxel Tracking Module (M12)

Secrets:

Services:

Implemented By: AutoVox

Type of Module: [Record, Library, Abstract Object, or Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

7.2.13 Highlight Manager Module (M13)

Secrets: How to define the rules that associate voxel semantics with a highlight colour. How high colours are internally represented. How conflicts are resolved when a voxel qualifies for multiple highlight colours simultaneously. How colour palette, which encapsulates all highlight colours, is chosen. How accessibility requirements influence the selection of highlight colours.

Services: Maps relationship between highlight colour and semantic meaning of highlighted voxels.

Implemented By: AutoVox

Type of Module: Abstract Object

7.2.14 Export Validation Module (M14)

Secrets:

Services:

Implemented By: AutoVox

Type of Module: [Record, Library, Abstract Object, or Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

7.2.15 Export Manager Module (M15)

Secrets:

Services:

Implemented By: AutoVox

Type of Module: [Record, Library, Abstract Object, or Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

7.2.16 Error Diagnostic Handler Module (M16)

Secrets:

Services:

Implemented By: AutoVox

Type of Module: [Record, Library, Abstract Object, or Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

7.3 Software Decision Module

7.3.1 Model Structure Module (M17)

Secrets:

Services:

Implemented By: AutoVox

Type of Module: [Record, Library, Abstract Object, or Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

7.3.2 Graphics Adapter Module (M18)

Secrets: How the graphics API is initialized and configured for rendering. How input data and formats are configured to remain compliant with API requirements. How rendering parameters are derived from visualization state. How resources are allocated and managed during rendering.

Services: Facilitates the use of the graphics API to render the 3D model and general UI in accordance with the state provided by the Visualization State Manager.

Implemented By: AutoVox

Type of Module: Library

7.3.3 Database Handler Module (M19)

Secrets:

Services:

Implemented By: AutoVox

Type of Module: [Record, Library, Abstract Object, or Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

7.3.4 Export Structure Module (M20)

Secrets:

Services:

Implemented By: AutoVox

Type of Module: [Record, Library, Abstract Object, or Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

8 Traceability Matrix

This section shows two traceability matrices: between the modules and the requirements and between the modules and the anticipated changes.

Req.	Modules
R1	M??, M??, M??, M??
R2	M??, M??
R3	M??
R4	M??, M??
R5	M??, M??, M??, M??, M??, M??
R6	M??, M??, M??, M??, M??, M??
R7	M??, M??, M??, M??, M??
R8	M??, M??, M??, M??, M??
R9	M??
R10	M??, M??, M??
R11	M??, M??, M??, M??

Table 2: Trace Between Requirements and Modules

AC	Modules
AC1	M??
AC2	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??

Table 3: Trace Between Anticipated Changes and Modules

9 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. **?** said of two programs A and B that A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification. That is, A *uses* B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B. Figure 1 illustrates the use relation between the modules. It can be seen that the graph is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system, and modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels.

[The uses relation is not a data flow diagram. In the code there will often be an import statement in module A when it directly uses module B. Module B provides the services that module A needs. The code for module A needs to be able to see these services (hence the import statement). Since the uses relation is transitive, there is a use relation without an import, but the arrows in the diagram typically correspond to the presence of import statement. —SS]

[If module A uses module B, the arrow is directed from A to B. —SS]

Figure 1: Use hierarchy among modules

10 User Interfaces

[Design of user interface for software and hardware. Attach an appendix if needed. Drawings, Sketches, Figma —SS]

11 Design of Communication Protocols

[If appropriate —SS]

12 Timeline

[Schedule of tasks and who is responsible —SS]

[You can point to GitHub if this information is included there —SS]

References