Module Guide for Flow

Team 9, min-cut
Ethan Patterson
Hussain Muhammed
Jeffrey Doan
Kevin Zhu
Chengze Zhao

January 22, 2026

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
F	Functional Requirement
NF	Non-Functional Requirement
SRS	Software Requirements Specification
SC	Scientific Computing
SRS	Software Requirements Specification
Flow	Explanation of program name
UC	Unlikely Change
[etc. —SS]	[... —SS]

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 (Parnas et al., 1984). We advocate a decomposition based on the principle of information hiding (Parnas, 1972). 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 Parnas et al. (1984), 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 (Parnas et al., 1984). 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 operating system environment on which the software is running.

AC2: The UI framework or technology used for the display of Flow.

AC3: The format used for saving and loading the note files and diagrams (e.g., XML, JSON, binary).

AC4: The syntax used for editing and navigating the note and diagrams.

AC5: The internal representation of geometric shapes and diagrams.

AC6: The rules as well as algorithms used for manipulating shapes (resizing, rotating, grouping, etc).

AC7: The user preferences and configuration options available for the user to customize.

AC8: The text formatting options available (e.g., font size, color, style).

AC9: The formats and options available for importing or exporting notes (e.g., PDF, image formats).

AC10: The implementation of the undo/redo functionality.

AC11: The method used for saving user data and session state.

AC12: The implementation of how the state of the canvas will be captured and restored.

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: Screen display and file storage).

UC2: The target platform will be desktop computers (Windows, MacOS, Linux).

UC3: The note structure will remain a combination of text and geometric shapes.

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: Display Interface Module

M2: Input Interface Module

M3: File Interface Module

M4: Geometry State Parser Module

M5: Geometry State Converter Module

M6: Commands Parser Module

M7: Mode Commands Module

M8: Shape Interface Module

M9: User Preference Module

M10: Text Buffer Module

M11: Geometry State Module

M12: Geometry State Mutator Module

M13: Undo Redo Module

M14: User Persistence Module

Level 1	Level 2
Hardware-Hiding Module	Display Interface Module (M1)
	Input Interface Module (M2)
	File Interface Module (M3)
Behaviour-Hiding Module	Geometry State Parser Module (M4)
	Geometry State Converter Module (M5)
	Command Parser Module (M6)
	Mode Commands Module (M7)
	Shape Interface Module (M8)
	User Preference Module (M9)
Software Decision Module	Text Buffer Module (M10)
	Geometry State Module (M11)
	Geometry State Mutator Module (M12)
	Undo Redo Module (M13)
	User Persistence Module (M14)

Table 1: Module Hierarchy

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 design of our system stems directly from the key functional and non-functional requirements outlined in the SRS. The requirements identified features such as note creation, text editing, diagram integration, file management, and undo/redo functionality, all of which influenced how we structured the system into distinct but interconnected modules. For instance, the user interaction and data handling requirements motivated the separation between interface-related modules and logic-handling components, ensuring maintainability and scalability. Likewise, requirements emphasizing usability, responsiveness, and persistence informed the decision to isolate rendering, state management, and data storage responsibilities into separate layers. Overall, the transition from requirements to design aims to ensure that each major function defined in the SRS is mapped to a cohesive part of the architecture, supporting modular growth and clear boundaries between user interface, behavior logic, and data management layers.

7 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by Parnas et al. (1984). 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. *Flow* means the module will be implemented by the Flow 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

7.1.1 Display Interface Module (M1)

Secrets: Implementation of rendering to the screen.

Services: Provides API to render graphics and text to the screen.

Implemented By: [OS, Electron, React]

7.1.2 Input Interface Module (M2)

Secrets: User input event handling.

Services: Provides API to capture user input events (i.e. keyboard input).

Implemented By: [OS, Electron]

7.1.3 File Interface Module (M3)

Secrets: File system interaction.

Services: Provides API to read and write files (open, save).

Implemented By: [OS]

7.2 Behaviour-Hiding Module

7.2.1 Geometry State Parser Module (M4)

Secrets: Logic for parsing the input file format into internal geometry representation state.

Services: Parses input files and converts them into the internal data structures used by the system.

Implemented By: [Flow]

Type of Module: [Abstract Object]

7.2.2 Geometry State Converter Module (M5)

Secrets: Logic for converting internal geometry representation state into output file format.

Services: Converts internal data structures into output files (e.g. xml for local saves, pdf for exporting).

Implemented By: [Flow]

Type of Module: [Abstract Object]

7.2.3 Command Parser Module (M6)

Secrets: Mapping between raw input events to editor commands.

Services: Converts keyboard input into internal commands.

Implemented By: [Flow]

Type of Module: [Abstract Object]

7.2.4 Mode Commands Module (M7)

Secrets: Mapping between active mode and available commands.

Services: Provides the set of available commands based on the current mode.

Implemented By: [Flow]

Type of Module: [Record]

7.2.5 Shape Interface Module (M8)

Secrets: Data structure containing shape properties and methods (e.g. size, position).

Services: Provides an interface for creating and manipulating a shape in the system.

Implemented By: [Flow]

Type of Module: [Abstract Data Type]

7.2.6 User Preference Module (M9)

Secrets: Data structure containing user preferences (e.g. theme, shortcuts) and actions for creating them.

Services: Provides an interface for creating and manipulating user preferences in the system.

Implemented By: [Flow]

Type of Module: [Abstract Object]

7.3 Software Decision Module

7.3.1 Text Buffer Module (M10)

Secrets: Text data structure.

Services: Provides editing functionality for text data.

Implemented By: [Flow]

Type of Module: [Abstract Data Type]

7.3.2 Geometry State Module (M11)

Secrets: Geometry state data structure.

Services: Holds the current state of the geometry.

Implemented By: [Flow]

Type of Module: [Abstract Data Type]

7.3.3 Geometry State Mutator Module (M12)

Secrets: Algorithms for updating geometry state.

Services: Provides methods to mutate the geometry state (add, delete, modify shapes).

Implemented By: [Flow]

Type of Module: [Abstract Object]

7.3.4 Undo Redo Module (M13)

Secrets: Algorithms for managing undo and redo stacks.

Services: Provides functionality to undo and redo actions performed on the geometry state.

Implemented By: [Flow]

Type of Module: [Abstract Object]

7.3.5 User Persistence Module (M14)

Secrets: Mechanism for saving and loading user preferences and shortcuts.

Services: Provides functionality for persisting user preferences and shortcuts.

Implemented By: [Flow]

Type of Module: [Abstract Object]

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
F211	M1, M2, M3, M10, M11, M12, M14
F212	M1, M3, M14
F213	M1, M2, M10
F214	M1, M2, M8, M11, M12
F215	M8, M11, M12
F216	M1, M7, M11
F217	M3, M10, M11, M14
F218	M3, M1, M11
F219	M13, M10, M11, M12
F220	M14
NF211	M1, M2, M8, M10, M11, M12
NF212	M3, M10, M11, M14, M13
NF213	M1, M2, M3
NF214	M6, M7, M2
NF215	M3, M14
NF216	M3, M14, M9
NF217	M1, M8
NF218	M1, M11, M12, M13

Table 2: Trace Between Requirements and Modules

AC	Modules
AC1	M1, M2
AC2	M1
AC3	M3, M14
AC4	M6, M7
AC5	M11, M4, M5
AC6	M12, M8, M11
AC7	M9
AC8	M10, M1
AC9	M3, M14
AC10	M13
AC11	M14
AC12	M11, M14

Table 3: Trace Between Anticipated Changes and Modules

9 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. Parnas (1978) 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 hierarchy for our system organizes the modules into a clear, layered structure that minimizes coupling and simplifies testing. At the top of the hierarchy are the user-facing modules—Display Interface (M1) and Input Interface (M2)—which collect user input and present output but perform no processing themselves. These modules rely on the Mode Commands Module (M7), which acts as the central coordinator that interprets high-level interactions and dispatches tasks to more specialized behavioural modules. M7 uses the Command Parser (M6), Shape Interface (M8), and Text Buffer (M10) modules to interpret commands, manipulate shapes, and edit textual content. None of these behavioural modules modify the document state directly; instead, they depend on the Geometry State Mutator (M12), which centralizes all updates to the application's data. M12 writes changes to both the Geometry State Module (M11), which stores the current in-memory representation of the document, and to the User Persistence Module (M14), which handles permanent storage tasks such as saving, loading, and managing autosave or history information. User pref-

erences (M9) are also stored through this persistence layer. The Geometry State Module (M11) is then used by lower-level utility modules—Geometry State Parser (M4), Geometry State Converter (M5), File Interface (M3), and Undo Redo Module (M13)—each of which derives additional representations or services from the core state. This structure forms a directed acyclic graph with a predominantly downward flow of dependencies, ensuring that higher-level modules rely on simpler, stable, and easily testable lower-level modules.

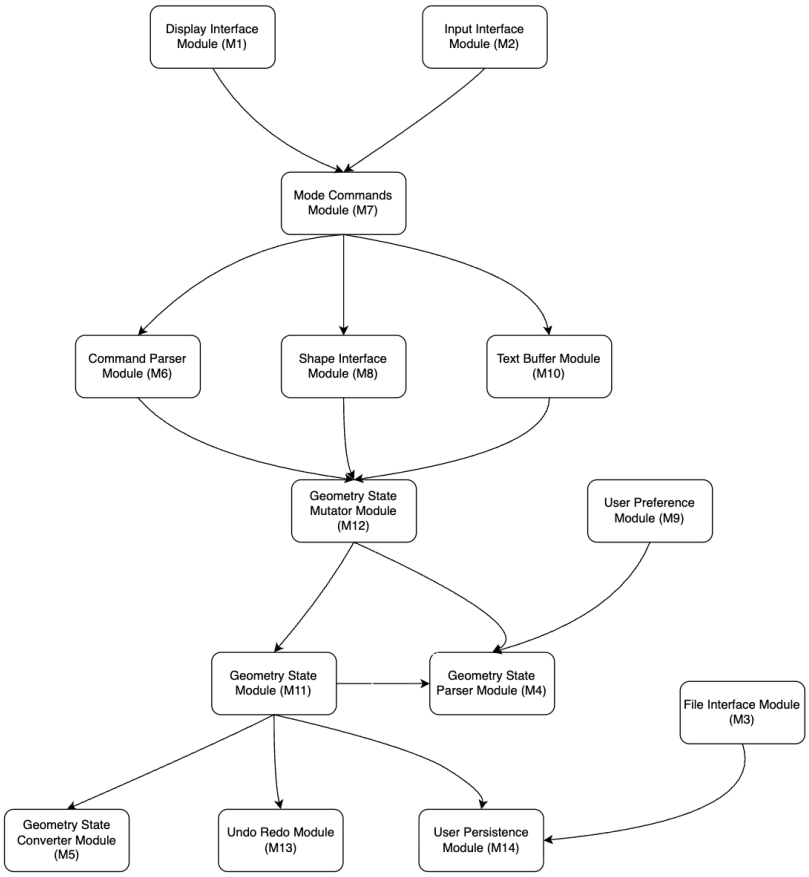

Figure 1: Use hierarchy among modules

10 User Interfaces

A rough drawing / sketch of our basic functionaility of creating elements / diagrams:

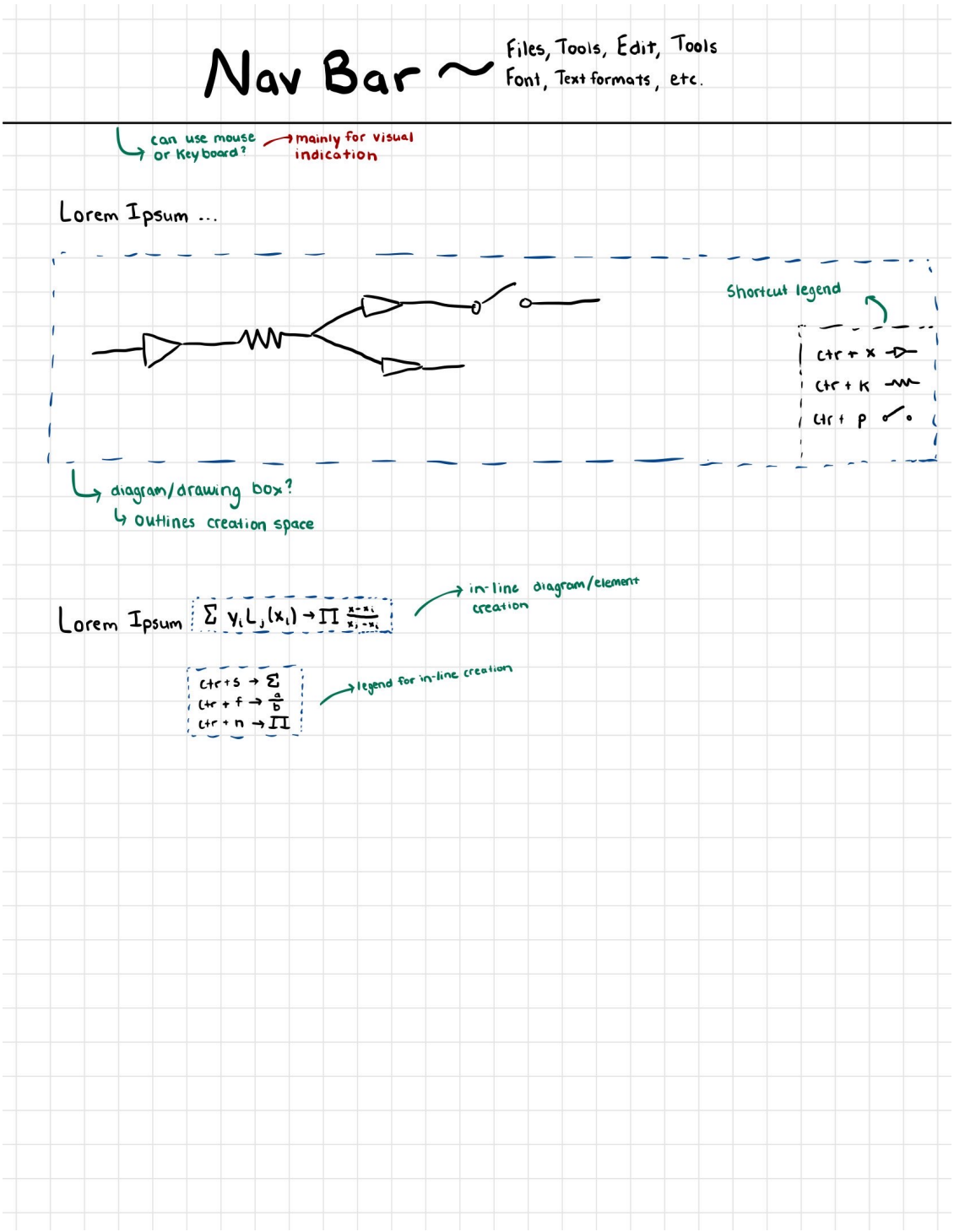

Figure 2: Sketch of UI focusing on creation boxes and legends

11 Design of Communication Protocols

This section was not deemed appropriate for our project.

12 Timeline

Phase	Dates	Focus	Modules	Member(s)
1 - Foundations	Oct 1 – Nov 10	Implement file I/O and basic note text storage	File Interface, Text Buffer	Ethan Hussain
2 - Core Editing	Nov 10 – Dec 15	Adding editing operations, undo/redo stack	Undo Redo, Text Buffer	Ethan Hussain
3 - UI Layer	Dec 1 – Jan 20	Create an editor window and input capture	Display Interface, Input Interface	Jeffrey Chengze
4 - Command and Mode Logic	Jan 10 – Feb 15	Shortcuts, editing modes, keyboard mappings	Command Parser, Mode Commands	Jeffrey Chengze
5 - Preferences	Feb 1 – Mar 10	Store and load user settings	User Preferences, User Persistence	Kevin
6 - Optional Geometry and Drawing Tools	Feb 15 – Apr 10	Add diagram/sketch layer support	Geometry State, Mutator, Parser, Converter, Shape Interface	Chengze, Kevin, Ethan
7 - Integration	Apr 1 – Apr 25	Full app integration, testing, and report write-ups	All Modules	All

Table 4: Development Timeline and Module Responsibilities

References

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press. ISBN none.

D.L. Parnas, P.C. Clement, and D. M. Weiss. The modular structure of complex systems. In *International Conference on Software Engineering*, pages 408–419, 1984.