Module Interface Specification for Software Engineering

Team 13, Speech Buddies
Mazen Youssef
Rawan Mahdi
Luna Aljammal
Kelvin Yu

November 12, 2025

1 Revision History

Date	Version	Notes
Nov 12, 2025	1.0	Added Modules M6-M11
Date 2	1.1	Notes

2 Symbols, Abbreviations and Acronyms

See SRS Documentation at [give url —SS]

[Also add any additional symbols, abbreviations or acronyms —SS]

Contents

3 Introduction

The following document details the Module Interface Specifications for [Fill in your project name and description —SS]

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at .... [provide the url for your repo —SS]

4 Notation

[You should describe your notation. You can use what is below as a starting point. —SS]

The structure of the MIS for modules comes from **?**, with the addition that template modules have been adapted from **?**. The mathematical notation comes from Chapter 3 of **?**. 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 Software Engineering.

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 Software Engineering 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, Software Engineering 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	
Behaviour-Hiding	Input Parameters Output Format Output Verification Temperature ODEs Energy Equations Control Module Specification Parameters Module
Software Decision	Sequence Data Structure ODE Solver Plotting

Table 1: Module Hierarchy

6 MIS of [Module Name —SS]

[Use labels for cross-referencing —SS]

[You can reference SRS labels, such as R??. —SS]

[It is also possible to use LaTeXfor hypperlinks to external documents. —SS]

6.1 Module

[Short name for the module —SS]

6.2 Uses

6.3 Syntax

6.3.1 Exported Constants

6.3.2 Exported Access Programs

Name	In	Out	Exceptions
[accessProg —SS]	-	-	-

6.4 Semantics

6.4.1 State Variables

[Not all modules will have state variables. State variables give the module a memory. —SS]

6.4.2 Environment Variables

[This section is not necessary for all modules. Its purpose is to capture when the module has external interaction with the environment, such as for a device driver, screen interface, keyboard, file, etc. —SS]

6.4.3 Assumptions

[Try to minimize assumptions and anticipate programmer errors via exceptions, but for practical purposes assumptions are sometimes appropriate. —SS]

6.4.4 Access Routine Semantics

[accessProg —SS]():

- transition: [if appropriate —SS]
- output: [if appropriate —SS]

- exception: [if appropriate —SS]

[A module without environment variables or state variables is unlikely to have a state transition. In this case a state transition can only occur if the module is changing the state of another module. —SS]

[Modules rarely have both a transition and an output. In most cases you will have one or the other. —SS]

6.4.5 Local Functions

[As appropriate —SS] [These functions are for the purpose of specification. They are not necessarily something that is going to be implemented explicitly. Even if they are implemented, they are not exported; they only have local scope. —SS]

7 MIS of Command Execution Layer

7.1 Module

Executes verified, mapped commands on the host device via the browser control bridge. Responsible for dispatch, cancellation, basic guardrails, timeout handling, and result reporting.

7.2 Uses

Browser control/automation client (e.g., `BrowserUse` bridge); OS process APIs; timer utilities; audit logger; configuration store for timeouts and retries.

7.3 Syntax

7.3.1 Exported Constants

None.

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
`CommandExecLayer`	`backend: Client, timeout_s: int`	self	-
`execute`	`cmd: ExecCmd`	`Result`	`PermissionError, TimeoutError, ExecError`
`cancel`	`cmd_id: UUID`	`bool`	`NotFoundError`
`status`	`cmd_id: UUID`	`ExecStatus`	`NotFoundError`
`validate`	`cmd: ExecCmd`	`bool`	-
`map_to_backend`	`cmd: ExecCmd`	`BackendReq`	`MappingError`
`rollback`	`cmd_id: UUID`	`bool`	`ExecError`

7.4 Semantics

7.4.1 State Variables

- `pending_q: map[UUID]`→`ExecCtx` — commands in flight
- `default_timeout_s: int` — global timeout for executions
- `retries: int` — max retry attempts for transient failures
- `backend: Client` — handle to browser automation bridge
- `audit: Logger` — sink for execution logs

7.4.2 Environment Variables

- `BROWSER_BRIDGE_URL` — connection string for automation client
- `EXEC_HARD_LIMIT_S` — absolute upper bound on execution time
- `LOG_LEVEL` — audit verbosity

7.4.3 Assumptions

7.4.4 Access Routine Semantics

`CommandExecLayer`(`backend`, `timeout_s`):

- transition: initialize `pending_q`, set `default_timeout_s`, bind `backend`, configure `audit`
- output: initialized instance
- exception: -

`execute`(`cmd`):

- transition: add to `pending_q`; invoke `map_to_backend`; dispatch to `backend`; update status; remove on completion
- output: `Result` with success flag, message, and optional payload
- exception: `PermissionError` if disallowed; `TimeoutError` if exceeds limits; `ExecError` on backend failure

`cancel`(`cmd_id`):

- transition: signal cancellation to backend; mark context as cancelled; remove from `pending_q`
- output: `true` if cancelled; otherwise `false`
- exception: `NotFoundError` if `cmd_id` not tracked

`status`(`cmd_id`):

- transition: none
- output: `ExecStatus` in {`queued`, `running`, `succeeded`, `failed`, `cancelled`}
- exception: `NotFoundError` if unknown

`validate`(`cmd`):

- transition: none

- output: `true` iff command matches allowed action set and guardrails (e.g., requires-confirmation flags met)
- exception: -

`map_to_backend(cmd)`:

- transition: none
- output: `BackendReq` (normalized request for the bridge)
- exception: `MappingError` if no mapping exists

`rollback(cmd_id)`:

- transition: attempt compensating operation (e.g., reopen tab, revert text entry)
- output: `true` on success; otherwise `false`
- exception: `ExecError` if rollback fails

7.4.5 Local Functions

- `start_timer(ctx)` — begin timeout tracking on a context
- `complete(ctx, result)` — finalize status, log, and cleanup
- `is_allowed(cmd)` — check guardrails and policy
- `retryable(err)` — decide if backend error can be retried
- `to_backend(payload)` — format request for bridge

8 MIS of Error Feedback

8.1 Module

Displays user-friendly error messages and recovery prompts. Surfaces execution issues, suggests next steps, and routes critical errors to support logs.

8.2 Uses

UI notification layer; audit logger; configuration store for messages; localization service (optional).

8.3 Syntax

8.3.1 Exported Constants

None.

8.3.2 Exported Access Programs

Name	In	Out	Exceptions
`ErrorFeedback`	`notifier: UiClient`	self	-
`show_error`	`code: string`, `detail: string`	-	-
`show_recovery`	`cmd_id: UUID`, `options: string list`	-	-
`dismiss`	`feedback_id: UUID`	`bool`	-
`log`	`event: ErrorEvent`	-	-

8.4 Semantics

8.4.1 State Variables

- `active: map[UUID] to FeedbackItem` — currently visible items
- `notifier: UiClient` — handle to UI notifications

8.4.2 Environment Variables

- `DEFAULT_LANG` — fallback locale
- `ERROR_COPY_PATH` — message templates location

8.4.3 Assumptions

UI client is available and permitted to display notifications.

8.4.4 Access Routine Semantics

`ErrorFeedback`(`notifier`):

- transition: initialize `active`; bind `notifier`
- output: initialized instance
- exception: -

`show_error`(`code`, `detail`):

- transition: create and register a feedback item in `active`; display via `notifier`
- output: -
- exception: -

`show_recovery`(`cmd_id`, `options`):

- transition: render recovery prompt with provided options
- output: -
- exception: -

`dismiss`(`feedback_id`):

- transition: remove from `active`; instruct UI to hide
- output: `true` if removed; otherwise `false`
- exception: -

`log`(`event`):

- transition: write event to audit log
- output: -
- exception: -

8.4.5 Local Functions

- `format_message(code, detail)` — produce a concise message
- `make_recovery(options)` — build prompt content

9 MIS of BrowserController

9.1 Module

Handles interaction with the browser controller: sends backend requests, receives statuses, and streams results back to higher layers.

9.2 Uses

Browser automation bridge client; transport layer; timer utilities; audit logger.

9.3 Syntax

9.3.1 Exported Constants

None.

9.3.2 Exported Access Programs

Name	In	Out	Exceptions
`BrowserController`	`bridge: Client`, `timeout_s: int`	self	-
`send`	`req: BackendReq`	`BackendResp`	`TimeoutError`, `TransportError`
`get_status`	`cmd_id: UUID`	`ExecStatus`	`NotFoundError`
`cancel`	`cmd_id: UUID`	`bool`	`NotFoundError`
`open_session`	`user_id: UUID`	`SessionId`	`TransportError`
`close_session`	`session_id: SessionId`	`bool`	`TransportError`

9.4 Semantics

9.4.1 State Variables

- `bridge: Client` — transport to the browser controller
- `default_timeout_s: int` — call timeout
- `sessions: set[SessionId]` — open sessions

9.4.2 Environment Variables

- `BROWSER_BRIDGE_URL` — endpoint for the controller

9.4.3 Assumptions

Controller endpoint is reachable and authenticated.

9.4.4 Access Routine Semantics

`BrowserController`(`bridge`, `timeout_s`):

- transition: set `bridge`, `default_timeout_s`, clear `sessions`
- output: initialized instance
- exception: -

`send(req)`:

- transition: none
- output: `BackendResp` from controller
- exception: `TimeoutError`, `TransportError` on failures

`get_status(cmd_id)`:

- transition: none
- output: current `ExecStatus`
- exception: `NotFoundError` if unknown

`cancel(cmd_id)`:

- transition: signal cancellation upstream
- output: `true` if accepted
- exception: `NotFoundError` if unknown

`open_session(user_id)`:

- transition: create session; add to `sessions`
- output: `SessionId`
- exception: `TransportError` if controller rejects

`close_session(session_id)`:

- transition: close remotely; remove from `sessions`
- output: `true` on success; otherwise `false`
- exception: `TransportError` on failure

9.4.5 Local Functions

- `with_timeout(call)` — wrap a bridge call with timeout
- `normalize(resp)` — normalize controller response

10 MIS of Session Manager

10.1 Module

Manages ongoing user sessions and their states: start, stop, track activity, and associate commands with sessions.

10.2 Uses

Persistent store or cache; clock utilities; audit logger.

10.3 Syntax

10.3.1 Exported Constants

None.

10.3.2 Exported Access Programs

Name	In	Out	Exceptions
`SessionManager`	`store: Store, ttl_s: int`	self	-
`start`	`user_id: UUID`	`SessionId`	`StoreError`
`stop`	`session_id: SessionId`	`bool`	`StoreError`
`get`	`session_id: SessionId`	`SessionState`	`NotFoundError`
`attach_command`	`session_id: SessionId, cmd_id: UUID`	`bool`	`NotFoundError`
`set_state`	`session_id: SessionId, state: SessionState`	`bool`	`StoreError`

10.4 Semantics

10.4.1 State Variables

- `store: Store` — persistence for sessions
- `ttl_s: int` — idle expiry threshold
- `index: map[SessionId] to SessionState` — in-memory cache

10.4.2 Environment Variables

- `SESSION_TTL_S` — default idle timeout

10.4.3 Assumptions

Store operations are atomic per session key.

10.4.4 Access Routine Semantics

`SessionManager(store, ttl_s)`:

- transition: set fields; warm cache from store if available
- output: initialized instance
- exception: -

`start(user_id)`:

- transition: create session; write to store; cache in `index`
- output: `SessionId`
- exception: `StoreError` on failure

`stop(session_id)`:

- transition: mark closed; evict from `index`; update store
- output: `true` on success; otherwise `false`
- exception: `StoreError` on failure

`get(session_id)`:

- transition: none
- output: current `SessionState`
- exception: `NotFoundError` if unknown

`attach_command(session_id, cmd_id)`:

- transition: append command reference to session state
- output: `true` if attached
- exception: `NotFoundError` if session unknown

`set_state(session_id, state)`:

- transition: update state in cache and store
- output: `true` if updated
- exception: `StoreError` on write failure

10.4.5 Local Functions

- `now()` — current timestamp
- `expired(state)` — checks idle expiry

11 MIS of Error Handling & Recovery Module

11.1 Module

Classifies errors, applies retry and backoff policies, and coordinates recovery or compensation actions to return the system to a consistent state.

11.2 Uses

Policy store; timer/backoff utilities; audit logger; command execution layer; browser controller.

11.3 Syntax

11.3.1 Exported Constants

None.

11.3.2 Exported Access Programs

Name	In	Out	Exceptions
`ErrorHandler`	`policy: Policy`	self	-
`handle`	`err: ExecError`, `ctx: ExecCtx`	`ActionResult`	-
`retry`	`cmd_id: UUID`	`bool`	`PolicyError`
`compensate`	`cmd_id: UUID`	`bool`	`ExecError`
`classify`	`err: ExecError`	`ErrorClass`	-
`record`	`err: ExecError`, `ctx: ExecCtx`	`UUID`	-

11.4 Semantics

11.4.1 State Variables

- `policy: Policy` — retry and compensation rules
- `history: map[UUID] to ErrorEvent` — recent errors
- `backoff: map[UUID] to int` — current backoff in seconds

11.4.2 Environment Variables

- `MAX_RETRIES` — hard cap for attempts
- `BASE_BACKOFF_S` — initial delay

11.4.3 Assumptions

Compensation actions are idempotent or guarded.

11.4.4 Access Routine Semantics

`ErrorHandler`(`policy`):

- transition: set `policy`; clear `history`, `backoff`
- output: initialized instance
- exception: -

`handle`(`err`, `ctx`):

- transition: classify; decide action (retry, compensate, fail); update `history`, `backoff`
- output: `ActionResult` describing action taken
- exception: -

`retry`(`cmd_id`):

- transition: schedule retry with policy backoff
- output: `true` if scheduled
- exception: `PolicyError` if retries exhausted

`compensate`(`cmd_id`):

- transition: execute compensation; update state
- output: `true` on success; otherwise `false`
- exception: `ExecError` if compensation fails

`classify`(`err`):

- transition: none
- output: `ErrorClass` such as transient, permanent, or user
- exception: -

`record`(`err`, `ctx`):

- transition: persist error event; update `history`
- output: event identifier `UUID`
- exception: `-`

11.4.5 Local Functions

- `calc_backoff(cmd_id)` — compute next delay
- `is_idempotent(cmd_id)` — check compensation safety

12 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)