Module Guide for Software Engineering

Team #16, The Chill Guys
Hamzah Rawasia
Sarim Zia
Aidan Froggatt
Swesan Pathmanathan
Burhanuddin Kharodawala

November 13, 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
[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: Matching Algorithm: The matching algorithm focuses on the goal of finding the best available drivers for the riders. This algorithm takes into account the user's location, timetable, time, and distance. For instance, it matches a rider with a driver that has the same timetable and lives in the same area. The current algorithm will be simply based on these four parameters, and it will have to change in the future to consider more parameters to increase its efficiency and accuracy. Moreover, subtle changes to the algorithm may also be expected for further optimization of its complexity and speed.

AC2: User Interface: As the application proceeds to go through its initial phase of development and use, usability testing is expected to be done to ensure user satisfaction. As the users provide feedback, changes will be made to the user interface in the future phases of development. Moreover, even after that, changes will be made based on accessibility requirements in the future. The layout and navigation are the two main things from the user interaction that would need the most changes as per user's feedback.

AC3: Payment Processing: Additional payment processors and pricing formulas will be used to determine the price estimate for drivers. An addition of parameters to estimate the cost would cause the algorithms to change.

AC4: Development Environment: Changes could be made to the CI/CD configuration to improve the reliability and stability of the system.

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: Devices: Hitchly is an application that is developed for mobile devices only. A version that is compatible to run on a desktop is an unlikely change as it requires major changes to the software design.

UC2: Language Support: The application is primarily based on the English language. Adding additional languages to the interface would require a major design decision.

UC3: Programming Language and Technologies: The core of this application is supposed to be developed using React Native, Typescript, and SQL. Making changes to how the application is developed is an unlikely change as it requires a change in the entire design of the application.

UC4: Verification Requirement: This app is strictly restricted for McMaster affiliated users as the entire concept of this application is designed for them. It is not subject to removal. Verification of the users will be their McMaster affiliated email.

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: Hardware-Hiding 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]

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

Level 1	Level 2
Hardware-Hiding Module	
Behaviour-Hiding Module	?
	?
	?
	?
	?
	?
	?
	?
Software Decision Module	?
	?
	?

Table 1: Module Hierarchy

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 (M1)

Secrets: The data structure and algorithm used to implement the virtual hardware.

Services: Serves as a virtual hardware used by the rest of the system. This module provides the interface between the hardware and the software. So, the system can use it to display outputs or to accept inputs.

Implemented By: OS

7.2 Behaviour-Hiding Module

Secrets: The contents of the required behaviours.

Services: Includes programs that provide externally visible behaviour of the system as specified in the software requirements specification (SRS) documents. This module serves as a communication layer between the hardware-hiding module and the software decision module. The programs in this module will need to change if there are changes in the SRS.

Implemented By: –

7.2.1 Input Format Module (M??)

Secrets: The format and structure of the input data.

Services: Converts the input data into the data structure used by the input parameters module.

Implemented By: [Your Program Name Here]

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 Etc.

7.3 Software Decision Module

Secrets: The design decision based on mathematical theorems, physical facts, or programming considerations. The secrets of this module are *not* described in the SRS.

Services: Includes data structure and algorithms used in the system that do not provide direct interaction with the user.

Implemented By: –

7.3.1 Etc.

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	M1, 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
AC??	M1
AC??	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. 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 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

This section describes the design of the communication protocols implemented in Hitchly. Communication occurs primarily between the client-side mobile application (developed with React Native / Expo) and the backend server (implemented with Node.js, Express, and tRPC). The system also integrates third-party APIs for geolocation and notifications.

11.1 11.1 Overall Architecture

All communication between the mobile client and the backend follows a request–response model over HTTPS, ensuring confidentiality and integrity. Data is serialized in JSON format and transferred through REST and type-safe tRPC endpoints. The architecture adopts a modular service-based design to support scalability and maintainability:

- **Frontend (Client):** React Native app that sends authenticated requests to backend endpoints via HTTPS using the tRPC client.
- **Backend (Server):** Express server exposing routes through the tRPC API layer. It validates and processes incoming requests, interacts with the PostgreSQL database through Drizzle ORM, and returns structured JSON responses.

- **External Services:** Mapping and geolocation (Google Maps / Mapbox API), Expo Push Notification Service for alerts, and optional OAuth verification via McMaster SSO (for email validation).

11.2 11.2 Communication Flow

A typical data exchange involves the following sequence:

1. The user performs an action in the mobile application (e.g., requests a ride, updates schedule, or rates a driver).
2. The frontend constructs a tRPC call encapsulating the action data and attaches the user's JWT authentication token in the header.
3. The server authenticates the request, validates input schemas, and executes the corresponding service logic.
4. The backend queries or updates PostgreSQL through Drizzle ORM and returns a success or error payload to the client.
5. The client updates its UI state using React Query based on the received data.

11.3 11.3 API Endpoints and Message Types

Hitchly's backend exposes the following logical API groups:

- **Auth API:** Handles user registration, login, McMaster email verification, and token refresh. Exchanges encrypted credentials and JWT tokens.
- **User API:** Manages user profiles, schedules, and preferences.
- **Matching API:** Accepts commute data, performs matching algorithm, and returns ranked ride offers.
- **Trip API:** Records completed trips, generates summaries, and calculates cost-sharing.
- **Rating API:** Sends and retrieves ratings and reviews.
- **Notification API:** Uses Expo Push Notification service to deliver trip confirmations, cancellations, and updates in real time.

All messages follow the JSON structure:

```
{
  "status": "success",
  "data": { ... },
  "timestamp": "2025-01-03T12:00:00Z"
}
```

11.4 11.4 Security and Reliability

Security and reliability are ensured through:

- **Transport Security:** HTTPS with TLS 1.3 and HSTS enforcement.
- **Authentication:** JWT Bearer tokens included in headers of all authorized requests.
- **Input Validation:** All incoming payloads validated using Zod schemas in tRPC to prevent injection or malformed data.
- **Error Handling:** Standardized error codes with human-readable messages returned to the client.
- **Rate Limiting:** Express middleware limits excessive requests to prevent abuse.
- **Retry Mechanism:** Client-side React Query retries transient network failures automatically.

11.5 11.5 Design Considerations

The communication design of Hitchly emphasizes the following principles:

- **Consistency:** Shared TypeScript types across frontend and backend guarantee end-to-end type safety through tRPC.
- **Extensibility:** Additional endpoints (e.g., payment API or advanced analytics) can be introduced without modifying existing client modules.
- **Scalability:** The stateless REST/tRPC design supports horizontal scaling of the Express server behind a load balancer.
- **Low Latency:** JSON over HTTPS is lightweight; caching of common responses and asynchronous notification delivery minimize perceived delay.

This design ensures secure, efficient, and maintainable communication between Hitchly's mobile application, backend, and external services, supporting its goals of safety, reliability, and sustainability for McMaster commuters.

12 Timeline

[Schedule of tasks and who is responsible —SS]

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

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.