# System Level Design

<Provide a high-level overview of how the functionality and responsibilities of the system were partitioned and then assigned to subsystems or components. Don't go into too much detail about the individual components themselves (there is a subsequent section for detailed component descriptions). The main purpose here is to gain a general understanding of how and why the system was decomposed, and how the individual parts work together to provide the desired functionality.>

Per the Software Requirements Specification (SRS), the Handicapped Driver design is required to “provide user/driver interaction via web browsers on both Android and iOS smart devices and Windows and OSx operating systems” (SRS 1.1); therefore, a presentation layer (GUI) was designed as an independent section and partitioned accordingly. Additionally, the system must store information related to parking lots/spaces, registered drivers and ongoing reservations made by drivers for the spaces. Therefore, an independent data layer (database) was also designed and partitioned. The core of the system remains as its own layer and contains both business logic and components to interact with the aforementioned presentation and data layers. The core also contains adapters as interfaces that implement features facilitated by external systems such as communication (SMS and Email), road navigation (Google Maps), and the scheduled “cleanup” of internal data. Overall, the Handicapped Driver System is architected as a multi‑tier software solution within a web‑based client‑server paradigm.

## Hierarchy Chart of Components

<Describe the major responsibilities that the software must undertake and the various roles that the system (or portions of the system) must play. Describe how the system was broken down into its components/subsystems (identifying each top-level component/subsystem and the roles/responsibilities assigned to it). Describe how the higher-level components collaborate with each other in order to achieve the required results. Don't forget to provide some sort of rationale for choosing this particular decomposition of the system (perhaps discussing other proposed decompositions and why they were rejected). >

As shown below (fig 4.1), the system is broken down by design into four major packages: data, presentation, core logic and external systems. The data layer contains the underlying storage implemented in normalized database form plus services that execute SQL instructions for data operations (select/insert/update/delete); this is a common software industry approach to data handling and utilizes Structured Query Language (SQL), another industry standard. The presentation layer is designed for execution within any browser that can render HTML and execute JavaScript. As both mobile devices (Android, iPhone) and desktop systems (Windows, iOS, Linux) employ such browsers, this approach provides a common “look and feel” for users and allows minimal code and GUI design duplication for developers. Access to features provided by external systems are handled by adapters used to implement inter‑system operations transparently to the core system. Business logic for the reservations, as well as, the adapters, and a bridge to the data layer are all designed into the core system; a façade, or “focal point” for communication with the presentation layer components is additionally integrated into the system core. High‑level system architecture was driven first by the requirement that user interaction was to occur via a multitude of disparate devices and operating systems, thus leading to the client‑server model using a web browser client. Further divisions were made to separate the data access component for scalability (i.e. MS Access to Oracle), and to accommodate and group external system dependencies together. The result of multiple divisions is an overall multi‑tiered (client‑server) architecture.



Figure 4.1 – Package Diagram

## Sequence Diagrams and Descriptions

<Describe the interaction behavior between classes. The typical sequence diagram will depict some or all of the behavior described in a use case. Therefore, sequence diagrams serve as an intermediary between use cases and classes. As a result, classes can be traced back to the requirements. >

The system requires a simple login sequence before any interaction with the rest of the system is allowed (req. x), with the exception of the New User feature. Depicted below is the initiation from the Login page within the GUI. The user’s login name and password are collected and passed to the Application Façade, through to the Driver class of the core system for any necessary pre‑processing. The Driver class then verifies that the collected information matches that stored in the database by searching (through the DataBridge) for a match. A success or failure token is then sent back through each initiating class until the user is either logged in or asked to “try again”.



Figure 4.2a – User/Driver Login



Figure 4.2b – Remind Driver

The UML Sequence Diagram for the system’s Remind Driver use case is displayed in figure 4.2b above. The Scheduler actor (external system) runs at predetermined intervals to inspect the reservations stored within the system to determine which ones need a reminder to the driver (i.e. reservation started 15 minutes ago, but driver has not changed status to “Parked”). The Scheduler will also “clean up” reservations that are far past overdue, but that is not shown in this sequence. In summary, the Scheduler requests a check through the Scheduler Adapter, which grabs Reservations through a request to the DataBridge. The DataBridge queries the Reservations table for those needing reminders (the interval for delay is set in the web.config file). Upon receiving a non-Null set of reservations, the Scheduler places notification messages into the SMS Messages table and notifies the SMS System actor (external system) through its adapter that there are messages awaiting.



Figure 4.2c – Make Reservation

The UML Sequence diagram for the system’s Make Reservation use case is displayed in figure 4.2c above. The Driver initiates a Make Reservation request through the MakeReservation form in the GUI through the Application Façade. The façade requests a list of parking lots with handicapped spaces and sends this back to the GUI. The Driver selects a parking lot and sends the selection through the façade, which retrieves a list of parking spaces and displays each space’s availability (e.g. Space #5 – reserved from 5:00pm to 8:00pm; Space #1 – no current reservations, etc.). The Driver selects an available space and indicates the start and stop times for the reservation, and the MakeReservation form calls the façade to commit the reservation. A Reservation object is then instantiated to apply any business rules, and the reservation is committed through the DataBridge to the database tables.



Figure 4.2d – Use Reservation (Park, Leave or Cancel)

The UML Sequence diagram for the system’s Use Reservation use case is displayed in figure 4.2d above. The Driver initiates a Use Reservation request to use a currently assigned reservation through the ShowReservation form in the GUI through the Application Façade. The façade asks the Driver object, which instantiates a Reservation object, which in turn collects the reservation through the DataBridge and back to the GUI. The Driver selects what action to take on the reservation (e.g. Park, Leave, Cancel) and sends the request along the same channels into the database. Note: The Remind Driver sequence (shown in Fig. 4.2b) facilitates drivers’ updating their reservation statuses in a timely manner to keep the overall system functioning as intended and relevant to the end users.