Ben Henshaw

Prof. Cantrell

CS5004

December 6, 2024

Final Project: Solar Fields

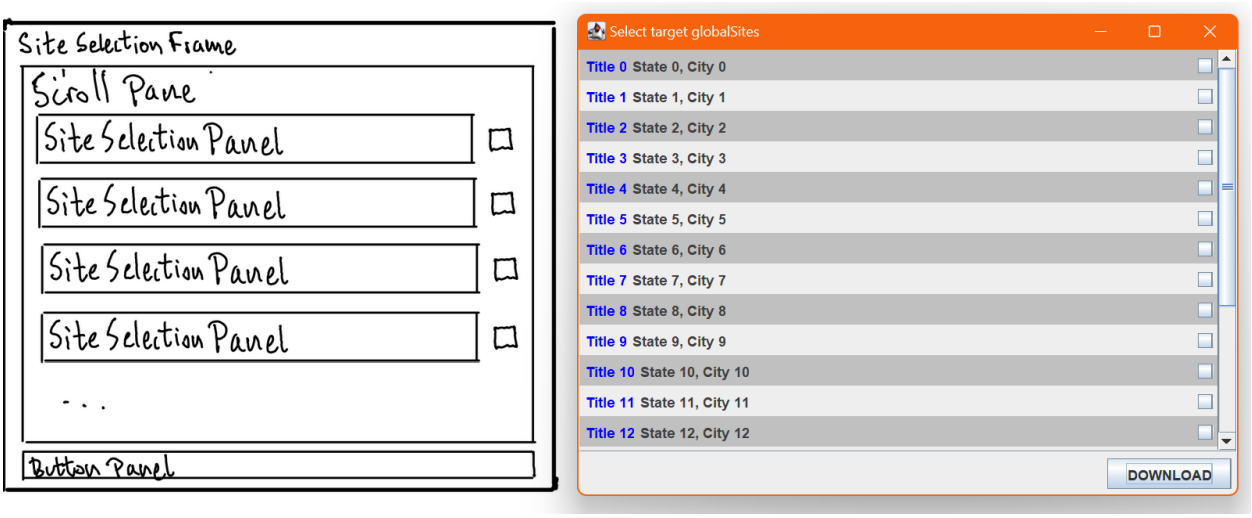
Solar Fields is an interactive client for accessing and generating entries in a text/image database with both online and offline modes. In the online mode, a global database is accessible to the user containing all available server-side data. The user can selectively download specific entries from this database to a local database, where it is available while offline. In the offline mode, the user can generate new entries connected to the entries in the local database. When connection is restored, the user can choose to upload these new entries to the global database, effectively syncing all changes between the local and global databases.

Both databases are characterized by a hierarchy of **Site** : **Ticket** : **Entry**. Each individual element in the database is assigned a corresponding UUID that allow for fast lookup even at discontiguous memory locations.

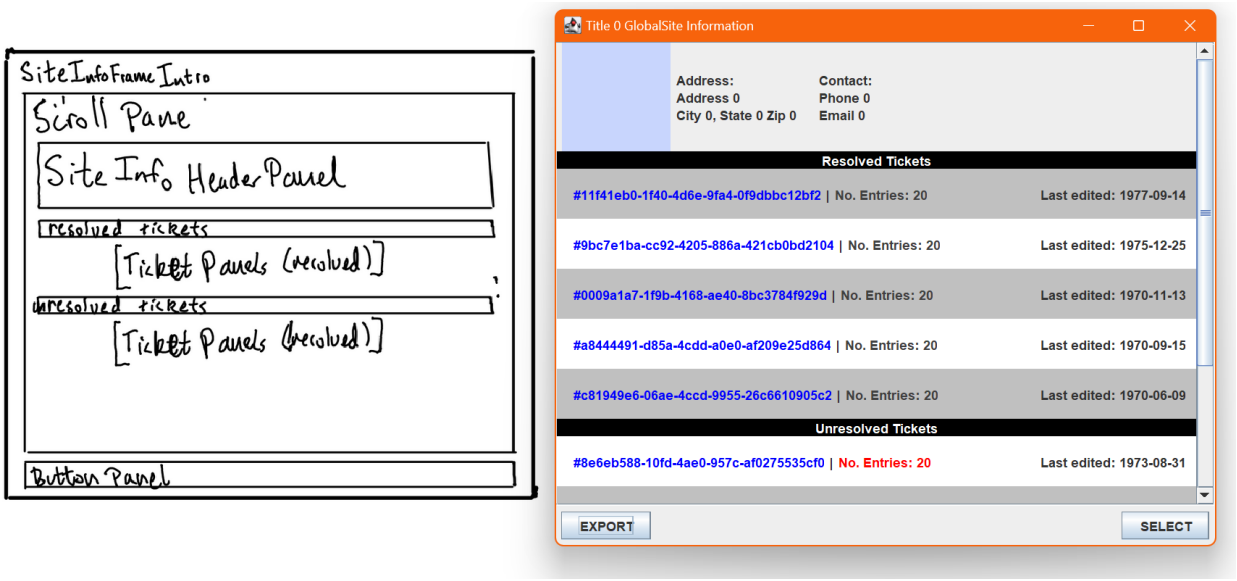
- A **Site** represents a business location, and it stores a title, contact, and address location. A site may have many associated Tickets.
    - A **Ticket** represents a broad category/issue and stores a description. A Ticket may have many associated Entries. A Ticket can be resolved or unresolved.
        - An **Entry** is a specific point of description or development about a Ticket. An Entry stores a description, an image, and a date of upload.

To simulate the behavior of a relational database, the Model interface I implemented utilizes a UUID-based ticketing system that relates a given UUID to in-memory objects. In-memory Sites store a list of Tickets, which themselves store a list of Entries. The ticketing system queries this hierarchy by iterating through the tree seeking specific UUIDs. Because UUID assignment is handled internally, the possibility of invalid UUIDs being passed to the ticketing system is unlikely. To differentiate between Locally and Globally stored objects and reduce the likelihood of data collisions, separate Local and Global implementations of the Site, Ticket, and Entry interfaces were created. This creates an easily-enforced control path for the process of “uploading” local values to the global database and “downloading” global values to a local database, with dedicated constructors designed for bidirectional conversion. New Entries and Tickets can only be generated as **LocalEntry** and **LocalTicket** objects, which can then be assimilated into the global database through an upload method. Flags indicating whether or not an Entry/Ticket is new or a Ticket/Site contains updated information allow the Model to add new data without having to rewrite the entire database. Because operations on the local database only involve adding new Entry or Ticket instances, many users can simultaneously locally operate on the same cross-sections of the global database without risking read/write collisions. While it hasn’t been implemented for this version of the application, global instances of Ticket and Entry have internal flags indicating whether or not the issues have been resolved/reviewed, which leaves final control over documentation to some administrator. The GUI was implemented using the Java Swing library.

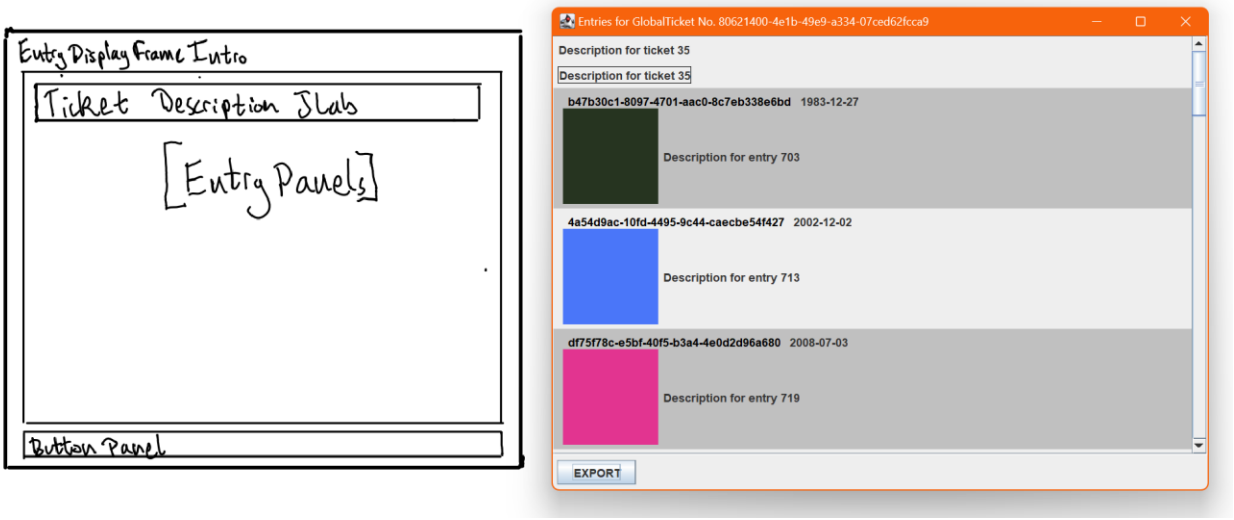
The application has two sessions: Intro and Edit. Intro is the client with a simulated connection to the global database. The application initializes to a **SiteSelectionFrame**, which displays its data in a JFrame**.** An internal flag indicates whether or not this frame is part of the intro or edit section. The frame has a content panel, which has two major components: a **ScrollPane** and a **ButtonPanel.** The **ScrollPane** is populated with **SiteSelectionPanel** objects, which each represent a unique site in the global database. Each panel has the site's title, location, and a togglable checkbox. In the intro section, the ButtonPanel has a Download button on the right side, which will download the data associated with each site selected in the ScrollPane to the local database when pressed. Below is a diagram of each component (left) with a screenshot of the app (right).



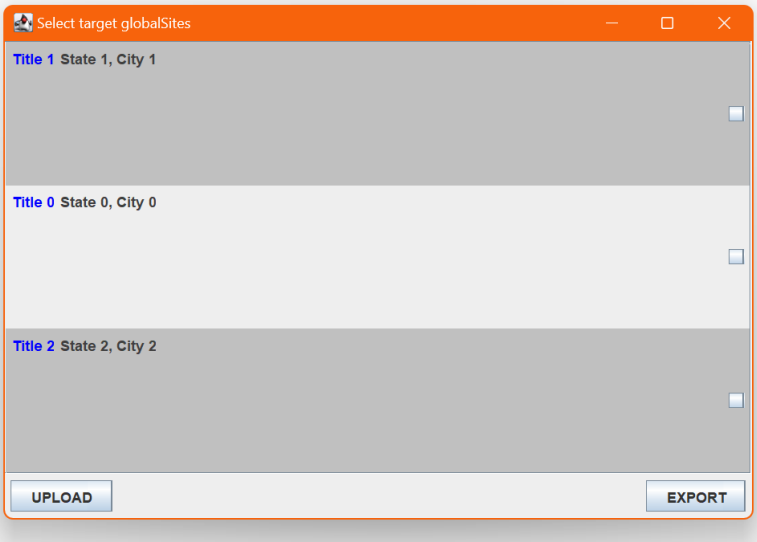
Clicking on any one of the SiteSelectionPanels will open a **SiteInfoDisplayFrameIntro.** This contains a header with more information about the site and a series of **TicketPanels** that show each ticket at a glance associated with that site. **TicketPanels** are sorted by resolved vs. unresolved, and they each show the number of related entries, the date of the most recent entry, and the Ticket ID.
SiteInfoDisplayFrameIntro has a ButtonPanel: one for exporting JSON data and the other that selects the corresponding SiteSelectionPanel on the previous frame and closes the current frame.



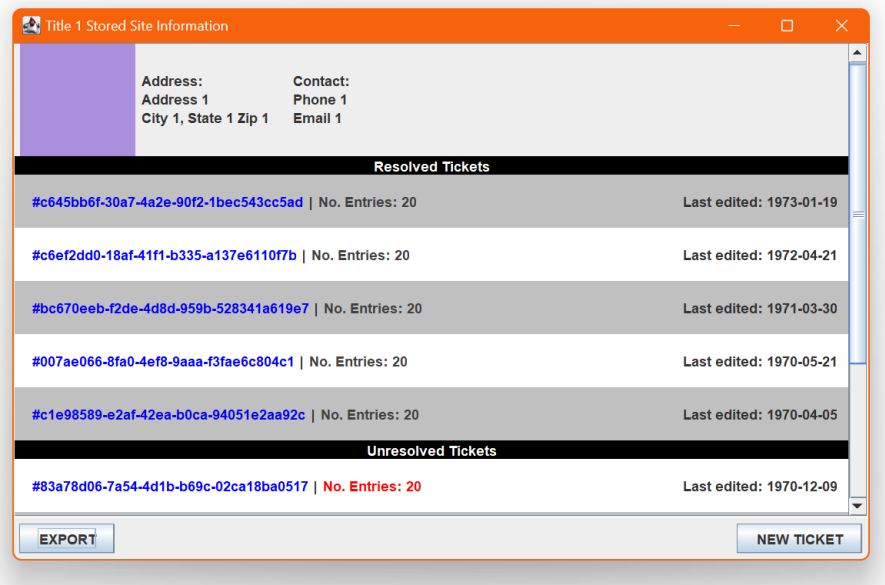
Clicking on any one of the TicketPanels will open an **EntryDisplayFrameIntro,** which is populated by **EntryPanel** objects that represent each entry associated with that ticket.



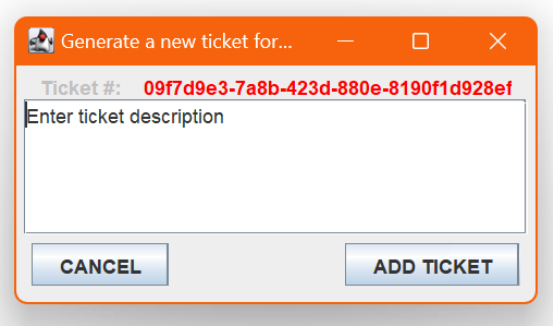
On the original SiteSelectionFrame, the user can select any number of sites. When the Download button is pressed, all of the windows that were open in the Intro section are closed and a new instance of SiteSelectionScreen is opened, now bearing a flag that indicates that an Edit session has begun. Whatever sites that were selected will be carried over to this new screen, indicating that their data was downloaded to the local database.



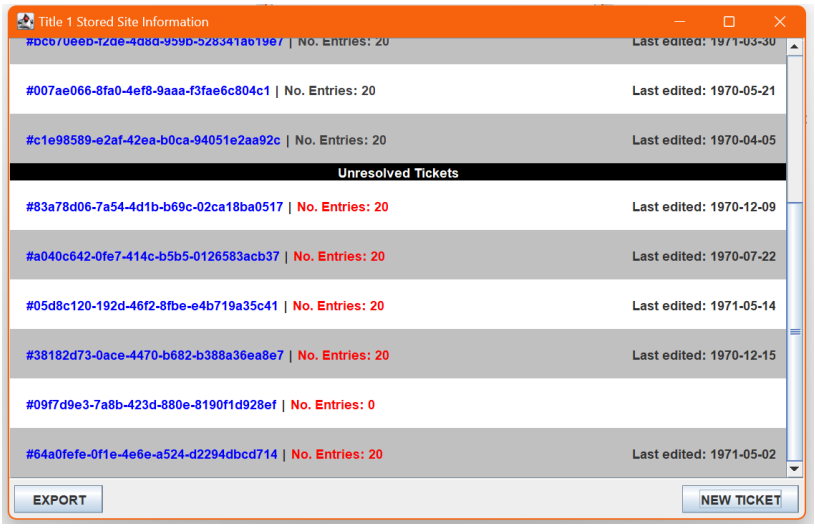
Selecting a site now opens a **SiteInfoFrameEdit.** A separate class was created for this frame because the underlying variables and mechanics stored in the object differed significantly enough from its sister Intro component that it made sense to generate a new one. This class carries references to its inner SiteInfoDisplayPanel and parent SiteSelectionPanel to simplify updating them when new tickets/entries are generated.



The most visible difference is the presence of an AddTicket button, which opens its own popup screen for entering a new ticket, which allows the user to designate a custom description to a new ticket with a randomly generated UUID.

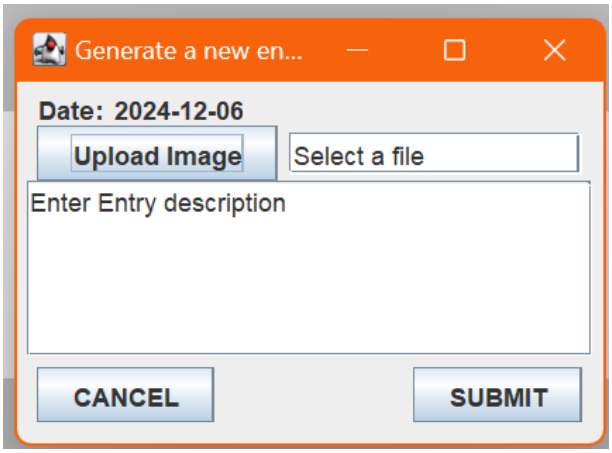


Upon generating a new ticket, the SiteInfoDisplayPanel is instantly updated to display the new ticket:

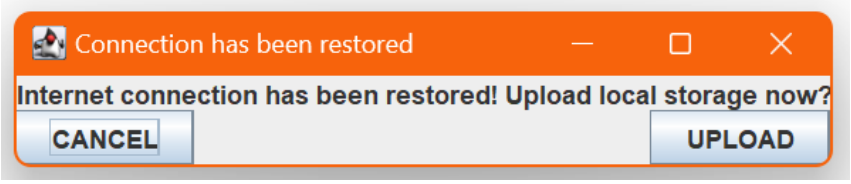


Clicking on any ticket will display an **EntryDisplayFrameEdit,** which has an AddEntry button but is otherwise identical to its Intro section counterpart. Like SiteInfoFrameEdit, it is built from references to its parent JFrames/Labels, which allows it to quickly update the visible display. The AddEntry button

reveals an **AddEntryScreen,** where a user can upload an image and designate a text description associated with that entry. Creating a new entry flags the parent ticket as 'updated' so new entries can be readily located during the upload process. It also will "unresolved" resolved tickets as the addition of new entries indicates that the ticket is no longer closed. Generating a new entry updates the parent TicketPanel and the EntryDisplayFrameEdit displaying the ticket information.

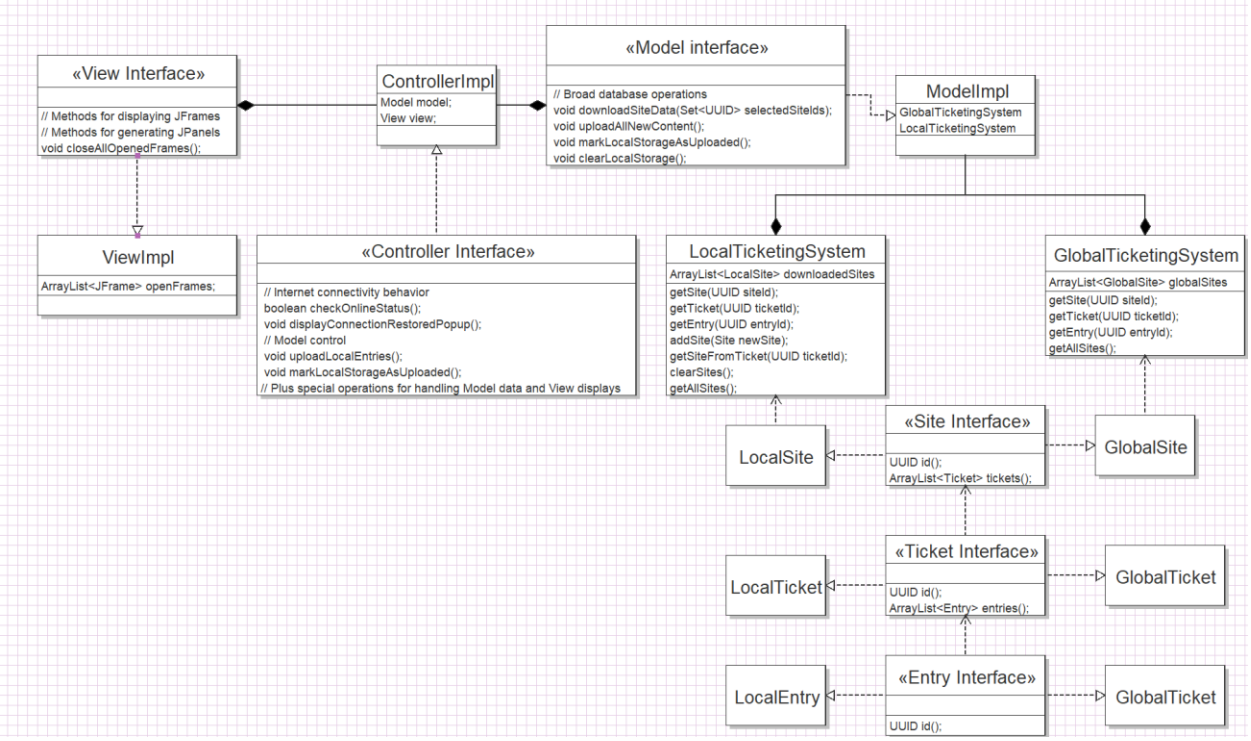


One of the behaviors that I wanted to simulate was a background process that checks for internet connectivity. Since this process was beyond the scope of the prototype I was developing for this application, I made a recursive method that checked to see whether or not a text file was empty to simulate this process. If the text file is given content at any point after the Edit section is opened, a **ConnectionRestoredPopup** frame opens and prompts the user to upload any new content.



If the user chooses to upload the data, all the new data in the local database will be marked as uploaded and will be synced with the global database and the popup will close. This allows the user to continue working in the edit section if desired. This popup raises an internal flag indicating that the user has been notified, and it will not notify the user again unless the connection is lost and restored again. If the connection is lost, then the upload button on the SiteSelectionFrame will be grayed out. The upload button can be pressed many times to idempotent effect; new content is added to the global database and new content in the local database is marked as having already been uploaded. Closing the SiteSelectionFrame closes all windows associated with the Edit session and initiates an Intro session with an updated global database.

The following UML diagram was simplified. The relationships between the classes that were created to represent graphical components was overly complicated and the original rendering was not useful. The dependency injection used to streamline the application's behaviors transcended the basic MVC relationship, e.g., some frame classes directly instantiate/manipulate other frame classes without making calls to the controller. While this may not have been the best design practice, it did make operations between related Jframes/panels/labels simpler in the long run.



Original Proposal

I am going to make a desktop client that can store image and text data while offline and an API to submit that information to an online client when internet is restored. There are two tables in this database:

1. *Image (5-8MB, BLOB format), text, and date of record. This is related to table 2 by the foreign key Ticket ID. There is also a boolean flag indicating whether this submission has been reviewed by an administrator (server-side)*
2. *A ticket ID table with text and a similar review boolean flag.*

The global database has locked records, so the offline client will store a local read-only view of that database as reference. New entries to table 1 can be composed and related to existing ticket IDs, and new Ticket IDs can be generated for table 2. When offline, this data is stored in a local SQLite instance. When internet connection is restored, this data is submitted to a global PostgreSQL database. The primary deliverable of this assignment is a local client with a GUI composed with Swing.

I have some experience with SQLite, and my understanding of relational databases I believe is sufficient to branch out into Postgres. I have some experience designing GUIs (tkinter with Python, web design with javascript), so I know that doing it in Java will be difficult and have a steep learning curve. Beyond this assignment, my primary motivation for making this program is for my freelance job, so not only is a grade keeping me going but a paycheck is as well. I am excited to make a GUI and work with relational databases. I'm not sure if I can deliver on fully fledged backend but I'm confident I can get a prototype of a desktop client working.

I implemented all of these general behaviors, but replaced the SQL API with a custom implementation of my own ORM-esque data structure. In effect, both databases are stored in memory. The interface design I predicated these structures with will allow for straightforward integration with a new Model implementation further down the line, but I do have a functioning prototype with all general behaviors implemented effectively.

Concept Map

Concept 1: Recursion in Practice	*SiteSelectionFrame.java, line 99* This recursive method runs in the background when a SiteSelectionFrame is instantiated within an Edit session of the app. It uses the Controller to check for simulated internet connectivity, waits a set interval, then calls itself again. A heap-accessible Boolean-flag breaks the loop when the window is closed.
Concept 2: Design Using ~~Abstract Classes~~ composition and Interfaces	I almost entirely avoided inheritance in this project and favored composition. *ConnectionRestoredPopup* is an extension of the JFrame class, but all classes containing the word *screen* or *frame* are compositions of JFrame, while all classes containing the word *panel* are compositions of JPanel. The MVC design pattern was implemented using the interfaces *Model, View,* and *Controller*. This was done partially because the present *ModelImpl* class was generated as a proof of concept and in later versions will be replaced with something that utilizes a Postgres or SQLite API. Because there is a separate type to differentiate local and global elements in the database, *Site, Ticket,* and *Entry* interfaces were generated so that symmetrical operations could be performed
Concept 3: Logical Abstraction using Generics and Lambda Expressions	Generics: As a GUI-driven application with a focused set of behaviors, the need for custom generic classes simply was not there. That layer of abstraction would have increased the complexity without affording any advantages. I originally considered making the display frames generic to the type of data they were meant to display, but as development continued the underlying implementation of different frames required the creation of entirely new classes as—despite outward similarity—they required a fundamentally different approach to implement their desired behaviors However, generic collections in the form of Sets, Lists, and ArrayLists were used repeatedly throughout the application. *ControllerImpl.java, line 83, 90, 94, 100, 123, 154, 175, 243* *LocalTicketingSystem.java, line 6, 63* *Model.java, line 12, 18, 21, 22, 23, 27, 28* *ModelImpl.java, line 36, 50, 54, 55, 60, 97, 102, 112, 113, 150, 191, 207, 214, 233, 239* *Site.java, line 21* *SiteInfoDisplayPanel.java, line 33, 104, 105*

	SiteInfoFrame.java, line 48, 114, 115 *SiteSelectionFrame.java, line 19, 68, 113* *SiteSelectionPanel.java, line 27* *Ticket.java, line 12, 13* *TicketPanel.java, line 8, 11* Lambda Expressions: Lambda expressions / method referencing was quintessential to the design of this application. As part of the Swing library, the behavior associated with a button press or a mouse click required the implementation of the ActionListener interface, which could be accomplished in shorthand with a lambda expression that encloses the method calls made following a user-generated event, which was done for almost every possible point of user interaction. All other user interaction events were defined using anonymous class definitions. Lambda functions for ActionListeners can be found at these locations: *SiteInfoFrameIntro.java, line 47, 51* *AddEntryScreen.java, line 84, 125, 128* *AddTicketScreen.java, line 64, 68* *SiteInfoFrameEdit.java, line 46, 50* *SiteSelectionPanel.java, line 65* Lambda expressions were also used to map and filter streams, often for display purposes, at these locations: *GlobalTicket,java, line 50* *LocalSite.java, line 49* *LocalTicket.java, line 71* *ModelImpl.java, line 98, 103, 147* *SiteInfoDisplayPanel,java, line 104, 105* *SiteInfoFrame.java, line 114, 115* Method References: Where lambda expressions only called an existing method, those could be simplified with method references. *SiteSelectionFrame.java, line 106* *GlobalSite.java, line 102* *ModelImpl.java, line 103, 108, 113, 192, 208, 215* *SiteInfoDisplayPanel.java, line 104, 105* *SiteInfoFrame.java, line 114, 115*
Concept 4: Higher Order Functions	In interacting with a database and displaying different elements from it in different ways, higher order functions were absolutely necessary. In this application, this involved calling them on streams of data to make display panels, filter results, or extract values from collections. Filter:

	When new data is transferred from the local database to the global database, flags were introduced to differentiate Site, Ticket, and Entry objects that are either new or have been updated. Filters were implemented to selectively isolate those objects and only transfer the new data. *ModelImpl.java, line 98, 103, 108, 113* Filters were also utilized to handle displaying resolved vs. unresolved tickets where TicketPanels were being stored in an ArrayList. *SiteInfoDisplayPanel, line 104, 105* Map: Site, Ticket, and Entry objects are all stored in memory in lists as a convenient means of storing and accessing the data contained in them, so that those lists can be mapped to lists of their identifying UUIDs without having to store them separately. An actual relational database would be able to handle that automatically, but maps were used in this application to simulate that behavior. *LocalTicketingSystem.java, line 42* *GlobalSite.java, line 102* *GlobalTicket.java, line 50* *LocalSite.java, line 49* *LocalTicket.java, line 71* *ModelImpl.java, line 192, 208, 216* There was no use for an implementation of a fold function in this application
Concept 5: Hierarchical Data Representation as an ADT	I implemented a hierarchical ADT to emulate the behavior of a relational database so that I could pass around identifying UUIDs to perform operations on the data associated with them. Every Site has a list of pointers to Tickets associated with it and stores that list in a local field. Every Ticket does the same with associated Entry objects. Ticketing systems were implemented so that individual elements could be fetched and manipulated based on UUID. *See: Model, Site, Ticket, Entry, LocalTicketingSystem, GlobalTicketingSystem*
Concept 6: MVC design + design pattern	The MVC design pattern was implemented for this project, and discrete classes were defined that represent each component (See: *Model, View, Controller*). One interesting development from this process was that the delineation between each component can sometimes be blurry. Often, the controller has to handle operations that closely involve use of the model but directly alter elements that would normally be handled by the view. When it came to thread-safe means of updating frames and panels that were already open based on events, it often required that those operations take place within a frame's class instead of delegating it to the controller or the view. In effect, while the MVC design pattern is the central driver for this

	application, the control of application's behavior is not exclusively centralized around the controller. The *Model* class could also be considered a *façade* pattern that abstracts the complex behavior of the underlying dual ticketing systems.
Concept 7: SOLID principles	S—Single Responsibility: In developing a GUI, sometimes where to draw the line between the functionality of different classes isn't totally clear. This is because event-driven programming is beholden to performing related operations within the same thread, and it sometimes made sense to store certain processes in the same classes that store JFrame instances to tie them together. For example, SiteSelectionFrame has different behavior depending on whether it was invoked during an Intro or an Edit session, but the overall role of the class is the same. The recursive internet connectivity check behavior is also stored in this class. This was to tie that behavior to this screen and localize the control variables that can break the loop, which are themselves controlled by the state of that JFrame. However, great effort was taken to design more specified classes for the two session types and for local versus global storage. O—Open-closed principle: Java Swing is a highly modular framework and the classes that I built with this library use composition to separate each individual component into an independent subunit. Each of these subunits can be mixed and matched or replaced depending on future design choices with no additional modification. The structure of each of these GUI-related classes allows for additional functionality to easily be added in the future without requiring any radical refactoring. The modular behavior of the Model is also worth noting. The entire ADT made for this project is internally structurally complete—but it is still a stand-in for a relational database. All of the desired behaviors of the Model have been outlined by its interface, but ModelImpl uses the static classes Local- and GlobalTicketingSystem to simulate relational behavior. In the future, the implementation will be replaced, but the interface itself will remain unchanged. L—Liskov Substitution Principle In designing this application, I favored composition over inheritance. I did this specifically so that each class wasn't totally beholden to just being a JFrame, and aliases for its components could be passed around so that other classes could alter them remotely. Because of the modular nature of Swing, JFrames, JPanels, and JLabels can all be substituted in different parts of the app. For example, there is only one class each for panels displaying Site selection info, Ticket info, and Entry info. The only instance of inheritance in this app is for *ConnectionRestoredPopup.java*, which extends the *JFrame* class. It

	can be freely interchanged with other JFrames within the app, although there is no real reason to do so. I—Interface segregation principle This is relevant to the separate Local and Global implementations of the Site, Ticket, and Entry. The distinction between the two is important, and it is used primarily to enforce segregation of the two types to differentiate data-stored-locally versus data-stored-globally. However, all operations designed for these implementations are symmetric across the two categories, and thus the two can be substituted for one another. Type coercion is sometimes necessary, but only for instantiation of Global from a Local parameter and vice versa. The one stipulation about this design choice is that Global classes have some default behaviors, e.g., a GlobalTicket will never be new or updated, so methods that invoke those fields will always return false. This behavior is important, however, as when this data is transferred to a Local counterpart, all downloaded data must be marked as not new as to differentiate it from new values introduced during an Edit session. D—Dependency inversion principle This is most visible in the MVC implementations. The Model fully encapsulates two ticketing systems, and can pass data upwards to the controller in the form of Site, Ticket, and Entry objects. The View is exclusively responsible for instantiating elements of the GUI, although occasionally aliases to these elements are passed upwards to the Controller for modification or the input of data extracted from the Model.
Extension Concept 1: Event-driven GUI design using Java Swing	This application is characterized by a responsive GUI that updates in real time in response to user input. Its functionality is driven by non-procedural events, that are monitored by ActionListener implementations that can spontaneously perform procedures when an action event is registered, such as a mouse click or a button press. The GUI can take user input and assimilate into the relational ADT, including text and file input with dynamic fields that depend on the local datetime. The data structure is represented visually in an intuitive interface that the user can explore and interact with in both Intro and Edit sessions. I could not get the wysiwyg to do what I wanted it to do so this was all done out manually.
Extension Concept 2: Virtual process threading	In an Edit session of the application, the app starts a virtual thread and runs a recursive internet connectivity check that runs in the background without interrupting the class's control flow. In each loop, the process checks the state of a Boolean in a single-element array. The purpose of this array was to create a consistent shared memory location that the thread—now running parallel to the thread in which the JFrame process is stored—can reference concurrently to

	the JFrame process. The process of losing or gaining internet connectivity can be simulated by adding content to or removing content from *res/online.txt* at any time. Once the user is notified for the first time, they won't be notified again unless simulated connectivity is lost and gained again.
Extension Concept 3: Simple ORM implementation	Working with a relational database, interacting with elements stored in that database is primarily done by passing around ID values. To make the transition from this version of the app to future versions as simple as possible, I built an ADT driven by unique UUID entries instead of primarily relying on passing around instances of the objects I was working with. There are probably more elegant ways of accomplishing what I did, but I ended up making an in-memory ORM that made it easy to generate test values that can be fetched through the Model by any method that has a correct UUID.