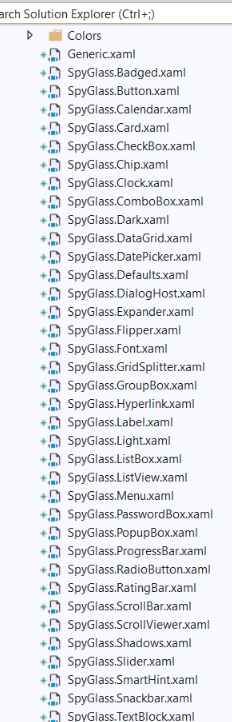
UI/UX Initial Assessment

**AREA:** SpyGlass Graphics

**OBSERVATIONS:** SpyGlass graphics are low quality, inconsistent in design, colors and fonts that do not adhere to Corporate branding guidelines. The graphics and styles are not organized in theme, and upon requirement to change or adjust a color or looks of lets a button, those changes would require explicit code change on every control being extremely time and cost consuming. Application does not respond to expected keyboard shortcuts and is not touch optimized.

**CONCERNS:** Primary concern is that design executed is not a professional grade and consecutively users will perceive poor design to underlying software and engineering behind it, even if that’s not the case. The graphics, icons, fonts , colors, controls should be organized into well defined theme mirroring Branding Guidelines and Web application development. Organizing application look and feel into a them, also will provide a robust access to change and any changes will always stay consistent with overall design.

To achieve this I will create a library **SpyGlass.Design.dll** that will host all the required design elements: colors, fonts, controls (aka buttons, dropdowns and etc), icons , domain specific controls, animation and etc. Such library will act as overall theme for the application GUI and can be reused in any other WPF type application. It also provides an streamlined way of experimenting and adjust GUI to achieve best satisfying look and feel for any given application.

**Example of variations for Button Controls:**



**AREA:** SpyGlass Architecture

**OBSERVATIONS:** Spyglass does not have an application architecture . The application is designed in such a way that each page does its own thing, yet components are tightly coupled and unrelated changes to one component will brake another. It is a cornerstone standard to use MVVM Architecture for Desktop Client Applications, Android and etc. **MVVM** is an architecture pattern which facilitates separation of projects into three logical layers with their own responsibilities. It facilitates modern techniques such as Dependency Injection, Separation of Concern, Unit testing and TDD.

**CONCERNS:** I am concerned about this because under current architecture is impossible to extend application and have predictable application behavior. In the code failure event it is very difficult or near impossible to identify the case of failure and address its fix. Also, application cannot be properly unit tested or broken done into components with well defined boundaries of responsibility, leading in turn to unexpected flow of application logic or lack thereof;

**Recommendation**

In 2005 **Microsoft Patterns and Practices** team have published and extensive architectural guidelines for designing WPF Desktop Applications (**PRISM**) with emphasis on modularity and plugin-based design using Dependency Injection rather than closely coupled dependent components. This approach allows to isolate development into logical components and features that can be assembled like lego cubes, and changes or modifications to each of such component will not break the entire application. Each such component can independently unit tested and used ad-hoc as needed. One of the most widely frameworks used is **PRISM** , with variety of Dependency Injection (containers) providers such as **MEF** (used in Visual Studio), **Unity**, Castle Windsor, AutoFac , Ninject and etc. I recommend Unity since its one of the most widely used and has small learning curve compered to MEF. Castle Windsor is excellent for introducing **Interceptors**, which are application features for automating logging or intercepting events or anything that needs to be addressed the application runtime.

A screenshot of a cell phone

Description automatically generated**Design Patterns used in Prism:**

**Adapter Pattern**

**Application Controller Patterns**

**Command Pattern**

**Composite View Pattern**

**Dependency Injection Pattern**

**Event Aggregator Pattern**

**Façade Pattern**

**Inversion of Control Pattern (IoC)**

**Observer Pattern**

**MVVM Pattern**

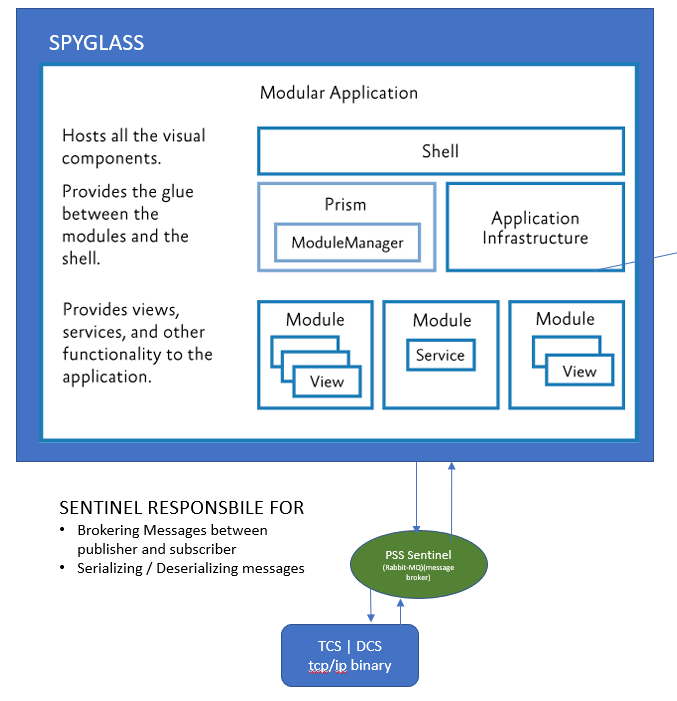
**Registry Pattern**

**Repository Pattern**

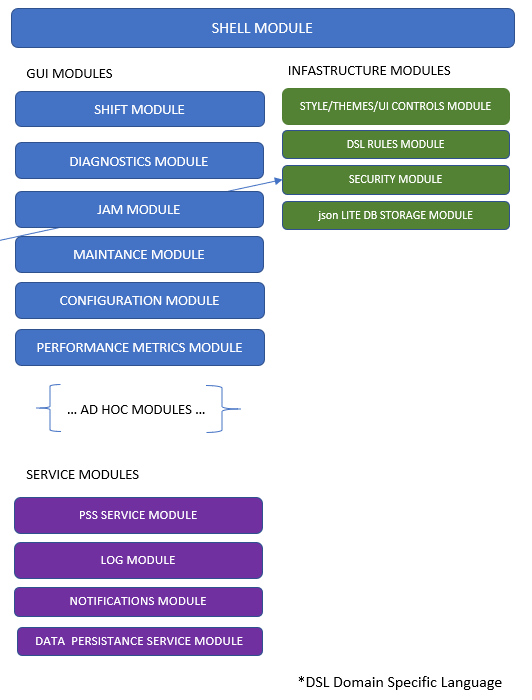
**Separated Interface and Plugin**

**Service Locator Pattern**

**SpyGlass ARCHITECTURE**

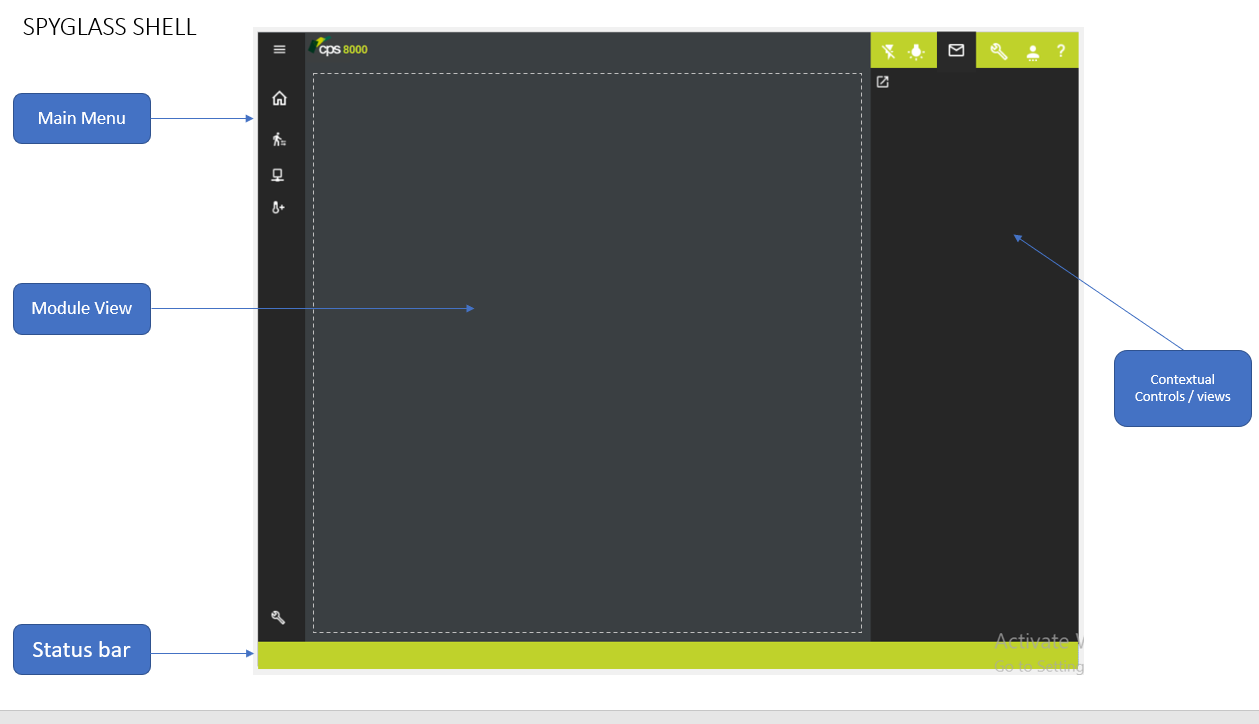


SpyGlass Logical Modules



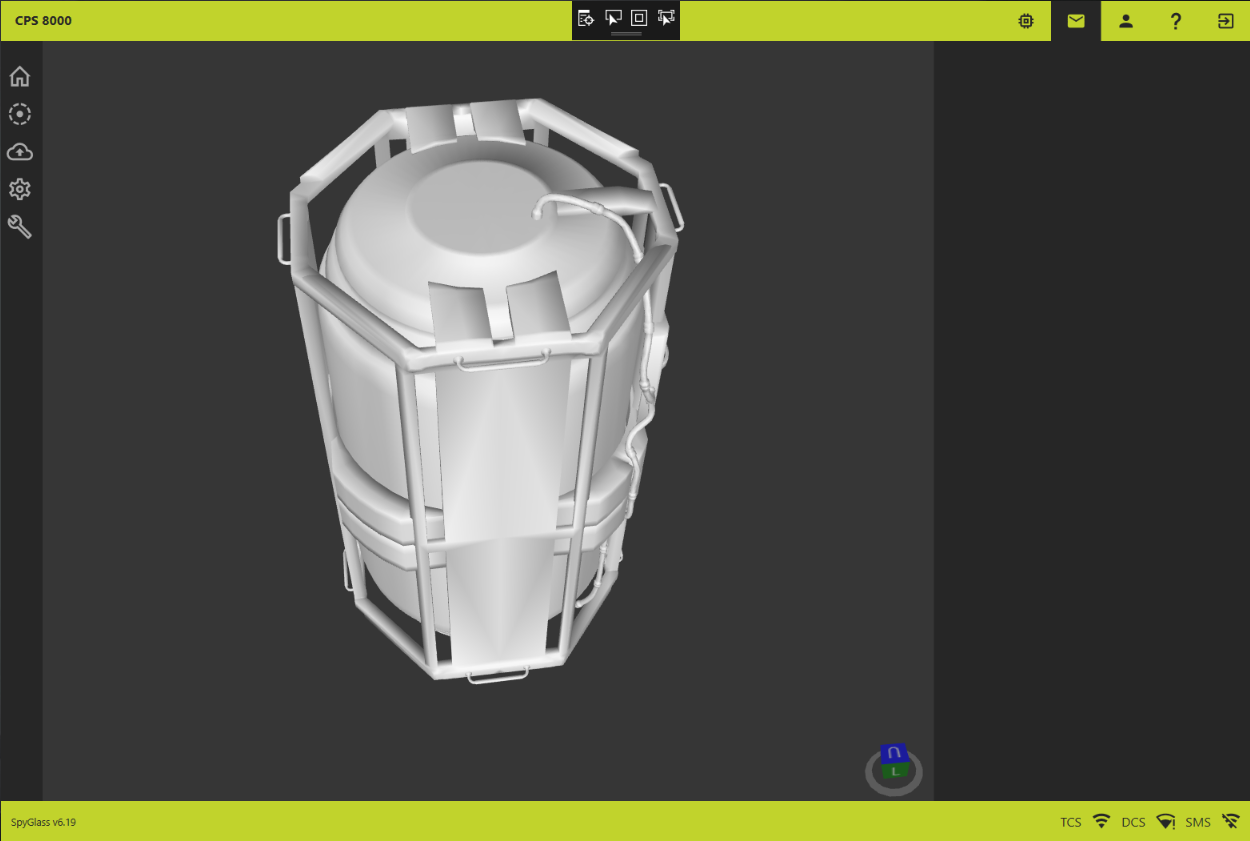
SpyGlass Shell

Application shell is a base application construct that provides all the application infrastructure and module management. Visually is represents component “landing” areas . The typical constant application areas are : TITLE Bar, MAIN MENU, WORK / CONTENT AREA, CONTEXTUAL MENU, USER / APPLICATION SETTINGS, STATUS BAR , MESSAGING NOTIFICATION AREA . Each application module or plugin will know its landing areas only at the runtime and coexist with other modules creating application ecosystem using PRISM framework described above which will provide: Region Management, Event Aggregation and Publish Subscribe, Communication between modules , unified logging and etc. THE GUI IN THIS ARCHITECTURE ACTS MERELLY AS A SKIN AND CAN BE COMPLELTY CHANGED WITHOUTH AFFECTING UNDERLYING COMPENT LOGIC. IN FACT Underlying COMPENENT , lets say JAM Component , has absolutely zero awareness of its GUI as its design is abstracted from its Visual Representation. Basically it means that Jam component could be web app, console app, mobile app because it has no hard coded dependencies to its UI and therefore its functionality should only be bound to executing logical commands versus being coupled to lets a specific button on a specific page. (Commanding Pattern)



**UX**

User Experience must ensure consistency of application use which means that at any point in the application user can make well informed decisions such as Navigation, Exception Handling, workflow without confusion. At any point during work a USER MUST have very clear understanding of a system status, navigation options and very clear access to other application areas at any point. System in turn must always inform user of “WHO AM I”, “WHAT I DO”, “HOW TO GET OUT OF HERE” and “HELP”



UX must be clean and avoid information noise. Contextual menus are essential for providing user with additional tools aiding in executing current task and they must change according to user task. Some of elements of the application such account /login, notifications must be accessible at any time independently from current user task.

**AREA:** Sentinel

**OBSERVATIONS:** Sentinel is designed to be a message broker between the machine and GUI. However currently it does much more. It also handles application state, events and closely bound to GUI application as per referencing buttons on specific pages, page coordinates and etc. Meaning that any Visual deviation will result in application failure. For example, imagine driving a car with a bumper sticker that displays your favorite quote. One day you come across a sticker that you now like better, you replace the sticker on your car and the car stops driving, because car engine is designed to work only with that sticker in that particular position.

Another issue is state management. Manually managing states is extremely difficult because the application state outcomes grow in geometric progression. For example if there are only 2 states lets on and off, there will be possible 4 outcomes ON ON, ON OFF, OFF ON, OFF OFF and each one has to be described in code. 10 states have 100 outcomes and so on. Our current CPS 8000 System has numerous states and events that results in hundreds or thousands of possible combinations and managing these types of complexities must be automated in such a way so that it reduces potential points of failure to a minimum and is relatively easy to :

1. Describe them both in human language and code
2. State configuration can be described using DSL (domain specific language)
3. States and resulting events can be chained, prioritized, audited
4. Changes to states can occur at both design and runtime (such as account permissions, or training mode ad etc)
5. Code can be unit tested
6. Code Changes in one set of state should not brake other states

Therefore, state management should be delegated to GUI application and because it is directly responsible for providing a User Interface based on each STATE of the machine. Such states are expressed using design patters and implementation is geared towards writing rules against a domain model. A rule consists of a set of conditions (patterns that match facts in the rules engine's memory) and a set of actions executed by the engine should the rule fire.

**Example:**

ACTOR - > CONDINTION ->EVENTS -> RULES -> OUTCOME

(If I am a user with admin rights, and I am application test mode, and I click “shift start” button after 3 pm on Monday, and cover is open and I have 5 dollar bills then “this and this should happen” ) and etc.

**CONCERNS:** The primary purpose of a broker is to take incoming messages from applications and perform some action on them. ... For example, a message broker may be used to manage a workload queue or message queue for multiple receivers, providing reliable storage, guaranteed message delivery and perhaps transaction management. The underlying principal of message broker design is independence from both publisher and subscriber.

A message broker acts as an intermediary platform when it comes to processing communication between two applications. RabbitMQ is one such open-source enterprise messaging system modeled on the Advanced Message Queuing Protocol (AMQP) standard.