# Chapter 7 Organizational Consequences

This chapter describes what problems that may occur for the users when accessing this new feature.

*How will the end-product will be tested?*

The end-product which will be in the form of a functionality on top of an existing web application, with working servers will be tested by the author, the Company Supervisor, and the Company Stakeholders.

GitHub and Azure DevOps will be the main platform where the author and his supervisor (the development team) can use for developing, collaborating, and making the version control Each function in FCS and Flutter Widgets and pages will be tested with test input, along with their outputs will be written down in Test Log documentation to make sure the software meets the desired requirements. A Design Document (included in this document) containing desired product mock-ups and wireframes will be produced for product’s UI to make sure it is easy to use for a good UX.

*How will the acceptance by the users of the new system be arranged?*

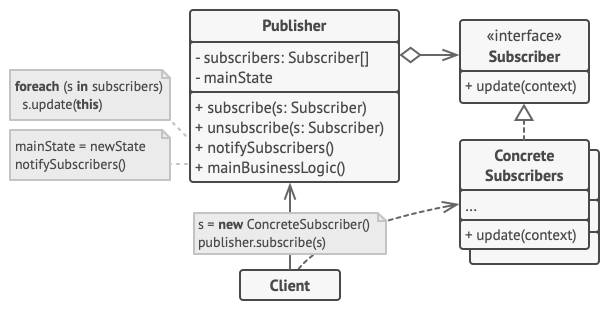
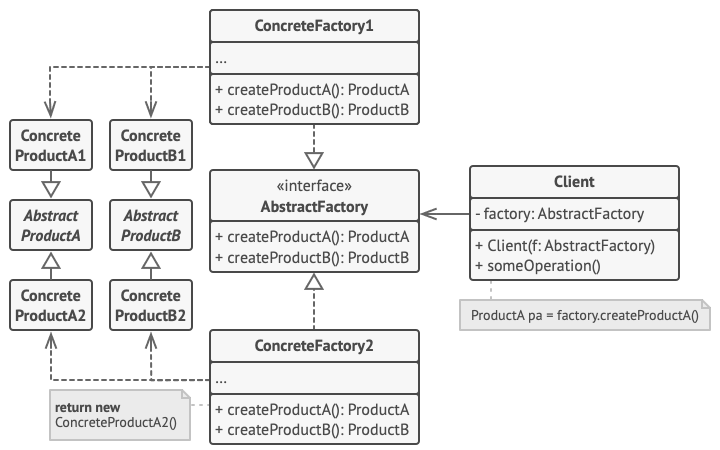
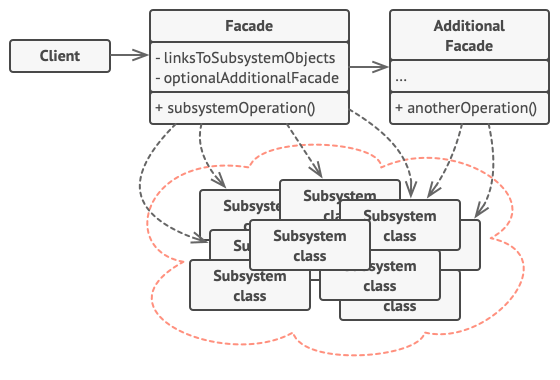
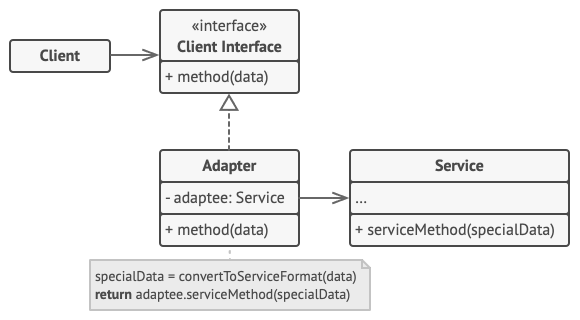
In the middle of the Realization phase, a pilot testing will be conducted with a small group of users, preferably the stakeholders and other people who do not know much about IT, to evaluate the system’s usability, functionality, and performance in a real-world environment. Feedback will be gathered from pilot users and necessary adjustments will be made before full deployment.

*Which conversion problems can be expected?*

The class base model from the SentinelOne API data format may differ from the format expected by existing infrastructure of the QaaS app. If this is the case, the author may need to create a new model just for SentinelOne to make it work. Other than that, the author will make sure that the new implementation will follow the skeletal structure of the QaaS app, along with using the same Widgets.

*Which training courses are required for the end users?*

There are no specific courses to make use of the system, as the page itself is designed to be user friendly to a customer that does not know anything about IT and cybersecurity. However, the end-users are expected to know the details of their own computers (specifications, what are the installed software, drivers, and hardware). The end-users are encouraged to read the SentinelOne official documentations about their own products, as well as

* **Guided Response Workflow:** Upon receiving an alert (whether by e-mailing the privileged users or by displaying a notification with a sound in the dashboard), the app should be able to present a guided workflow outlining step-by-step instructions on how to respond to the specific type of threat detected.
* **Interactive Response Options:** The guidance provided should offer interactive response options, including mitigation measures, containment strategies, and escalation procedures.
* Observer Pattern
* It is a behavioural design pattern where an object, known as the subject, maintains a list of its dependents, called observers. This is used commonly when there is a one-to-many relationships between objects, and changes to one object require updates to multiple other objects. This pattern will be useful, especially when dealing problems that may occur with sending e-mail and alert notifications in case of a cyber threat happening on a client’s machine. It will guarantee the handling of events or updates in real-time, making it easier for the application to be aware when certain changes occur.
* 
* Abstract Factory Pattern
* This is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. This pattern will be useful for adding or deleting modules and models from SentinelOne APIs for future use.
* 
* Provider
* This is not a design pattern but a modelling concept of state management solution that is commonly used in Flutter applications. It is based on the concept of Inherited Widgets and allows for a clean way to manage state in Stateful and Stateless widgets throughout the application.
* It will provide PODO (Plain Old Dart Object), *not to be confused in POJO (Plain Old Java Object) and JavaBeans classes in Java*, as a convenient way to manage the state of Flutter objects and propagate changes throughout the UI.
* BLoC (Business Logic Component)
* This is not a design pattern, but a concept related to MVC. It is particulary useful for managing complex state and business logic in Flutter apps. It separates UI from business logic and state management, providing a clear and scalable architecture.
* Façade
* It is a structural design pattern that provides a simplified interface to a complex system, hiding its complexities behind a single interface. This pattern will be used in the Table\_Page\_View.dart, where the page will display the data in the form of Paginated Data Table 2 for all the SentinelOne data models, depending on what was given.
* 
* Adapter Pattern
* The adapter is a great addition to use for the code scalability and maintenance for the future projects. It helps by providing a unified interface to interact with different kinds of external APIs or services, whether they are RESTful or SOAP API, abstracting away the implementation details.
* 
* Service Layer Pattern
* This pattern can be used to encapsulate the business logic of the application. It provides a way to organize the code that performs operations on the data, separating it from the Controller and Model. This is useful for handling complex business logic and keeping the Controllers and Models clean.
* Dependency Injection
* This design pattern can be used to manage dependencies regarding the Node environment. It will make the code more modular, easier to test, and improves the scalability of the application. It is particularly useful in a server-side application where there are multiple services and repositories that need to be injected in the Controllers.