Adapting to Change: A Study of the Software Architecture Evolution of a Physical Security Information Management System

Oğuzhan Özçelik
ASELSAN A.Ş.
Ankara, Turkey
e-mail: oozcelik@aselsan.com.tr

Halit Oğuztüzün
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
e-mail: oguztuzn@ceng.metu.edu.tr

***Abstract*—Physical Security Information Management (PSIM) system customizations tend to be similar to each other with core requirements being more or less the same in different projects. One of the most common differences in these projects is the sensors being used. Some sensors could be integrated into the PSIM system easily if they are compatible with a standard communication interface such as Open Network Video Interface Forum (ONVIF) protocols. But sensors that use a special communication interface need to be integrated one by one. A PSIM system is always expected to integrate additional sensors to its inventory. In order to do this easily, the modules that need to be developed to integrate a sensor must be segregated and developed individually for each sensor. These modules can be seen as features to be used in a software product line architecture. The planned reuse mentality of software product line engineering makes it possible to deliver similar products within a short amount of time. In this work, we aim to segregate the sensor integration of a PSIM system and compare the old and new generations of the architecture both qualitatively, based on their architecture models, and quantitatively, based on test results. Several tests and surveys have conducted in order to inspect the new architecture's performance.**

Keywords-Physical Security Information Management Systems; Physical Protection Systems; Software Product Line Engineering.

I. INTRODUCTION

A Physical Security Information Management (PSIM) system integrates diverse independent physical security applications and devices. Applications such as building management or network video recorder systems, and devices such as security cameras, access control systems, radars and plate recognition systems are used interconnectedly through a centralized platform. It is designed to ensure the physical security of a facility, city or an open field, while providing a complete user interface to the security operators to monitor and control them. With the help of PSIM systems, security personnel can make prompt decisions about a security situation by investigating the comprehensive picture the PSIM system generated with the data that it gathered, associated and analyzed.

This work is a continuation of our previous work [1]. We have conducted a survey to see the problem, and the gains we achieve with this architectural change better. And we have tested the old and new architecture in order to see whether the new architecture comes with a performance loss.

Physical Protection System (PPS) is also a common term to refer to such a system. Mary Lynn Garcia described the PPS functions, which can be seen in Figure 1, in three main categories: detection, delay, response [2]. Detection is the discovery of a malevolent incident. Measuring the threat level of an action would also be beneficial while deciding the following functions' extents. This measurement must provide information about the importance of detection and if it is important, every detail about the cause of the alarm. The level of detail is primarily based on the type of sensor that detected the alarm. Next function of a PPS is delay. After the adversary action got detected, the first thing to do is delaying its operations. This can be accomplished by locks, barriers or security personnel in the perimeter. The reason for this function is basically stalling the adversary in order to gain time for the next function, response. Response is the cumulative actions taken by the security personnel or system, in order to prevent adversary action.

The subject PSIM system of this work is called SecureX, which is not the name of the actual system, but a placeholder used for confidentiality reasons. SecureX is a PSIM system that aims to satisfy the needs mentioned above and to provide an easy integration environment for new sensors and applications. The ever-increasing number of such new systems and particular security needs of different customers drove SecureX team to embrace a software product line engineering approach in order to reduce the response time to reply to the customers' demands. These demands vary from

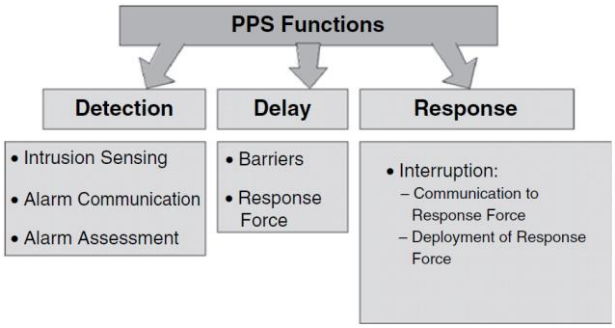

Figure 1. Functions of a Physical Protection System [2].

practical improvements to integrating a new sensor or security application as a feature to the system. SecureX is deployed with the full feature set and only at runtime these features are reduced to the ones required by a given customer, using different configuration files. Any new integration required by a customer needs to be developed as a feature in SecureX. Afterwards, a new SecureX build must be generated. Following every new integration, a new testing process takes place and because the previously integrated system might not always be available for testing, it must be guaranteed that the new integration will not affect the previously completed integrations. In this work, a new method for integrating such new systems while reducing the number of required tests is proposed. The new method is also going to be an evolutionary step toward a product line architecture.

The rest of the paper is structured as follows. In Section II, several PSIM products and their specializations are mentioned. Also, we briefly explain how they approach the sensor integration problem and why that is not enough in the case of SecureX. In Section III, the general architecture of SecureX is described and the point where sensor integration takes place is shown. Also, the technology that will be used is described. In Section IV, this sensor integration point is described in more detail. In Section V, the problems with the current architecture are explained and in Section VI, a new architecture that solves those problems is described. In Section VII, results of a survey and performance tests are detailed. In Section VIII, the benefits of the new architecture are shown by further explaining how it solves each problem of the current design.

II. Related Works

There are several companies offering PSIM products. Although they provide every essential feature of a PSIM system, they may have different specializations. Some companies are more promident in video management systems and some in geographic information systems. Plate recognition and access control systems are also fields in which a PSIM system can be used. In the SecureX's case, all four types of systems mentioned now can be used together.

Genetec [3] provides a video analytics tool to detect intrusions. They also develop access control systems and use plate recognition systems to monitor vehicles. Milestone [4] uses its own Network Video Recorder (NVR) system and provides an easy-to-use video management system. They work with numerous different companies and provide an easy integration framework to work with them. Nedap [5] is specialized in access control systems and they work with other companies like Genetec and Milestone to get integrated in their PSIM systems as well. However, not many details exist on how they work internally. These products integrate some general communication standards like ONVIF [6] protocols and also release Software Development Kits (SDK) and expect sensor manufacturers or customers to integrate their custom subsystems into the PSIM system as well. This way, they accelerate sensor integration by including numerous 3rd parties. While developing an SDK to use in integrations is a feasible solution, in the SecureX's case, the main objective is developing an architecture that can simplify not only the sensor integrations, but also the component selection to deploy because different customers have different requirements. Another requirement is that the new architecture will be able to remove the update and test overhead. A software product line architecture would be suitable to accomplish this goal.

Recently, Tekinerdogan et al. [7] described how a PSIM system should be designed with software product line engineering methodologies to reduce the cost of development by improving reuse. The present work describes a step in architectural evolution toward a product line architecture.

In different programming languages, there are many frameworks in which a software product line could be implemented. One specific technology that has the software product line implementation capability is called Open Services Gateway Technology (OSGi) framework for Java [8]. Its details will be explained in the coming chapters, but its abilities are shown by Almeida, E. et al [9]. In their work, they tried to provide a method that can be used in the domain implementation phase of software product lines. They conducted experiments using a pilot project in order to investigate the feasibility of their method. Seven M.Sc. students with industrial software development experience are selected and after a short training, the participants were expected to complete the tasks assigned to them. After the project had been completed, the quantitative analyses showed that the method is beneficial in developing software components with high maintainability while lowering the overall complexity. The participants also got surveyed and their answers indicated that the method provides useful guidance, thanks to the OSGi, a technology which is very suitable to be used in software product line development. But subjects without experience with this technology noted that they had challenges using it. However, these challenges are nothing that training cannot be overcome. Overall, their experimental study showed that using a software product line architecture with OSGi helps developers to build products with better quality.

III. Architecture of SecureX

SecureX is a PSIM system that is used in a wide variety of fields from border or airport security to protecting various critical facilities and oil or gas pipelines with special sensors. In some projects, the system is used in low performance computers and tablets while some projects use high performance servers. Some projects requires a dozen sensors to protect a small remote location and some uses thousands of sensors in a highly concentrated manner inside a city. Some projects are a combinations of those. Houndreds of small, secure facilities with dozens of sensors each, connected hierarchically to each other and at the top, controlled by high authority security officers. These different projects comes with different requirements from both public and private instutions. SecureX has to be able to adapt the different needs of each customer. This need

caused SecureX to be a highly configurable system that is tailored for every new customer and project.

SecureX has a distributed architecture which can be seen in Figure 2. Graphical User Interface (GUI) Clients of SecureX are installed on the computers of security officers, enabling them to monitor the entire security infrastructure of the area under surveillance. These clients are connected to the SecureX Server application which handles the communication between SecureX components. The server is also responsible for recording events, including detections and errors sent from adapter components to the central database. SecureX could also be installed in a hierarchical fashion in which higher servers could also control and monitor the security components that are connected to the servers under them. Under the SecureX Server, there are adapter applications for each sensor group such as camera, radar, plate recognition systems, access control systems, etc. These adapters are the points where the SecureX environment makes its connections to the outer world.

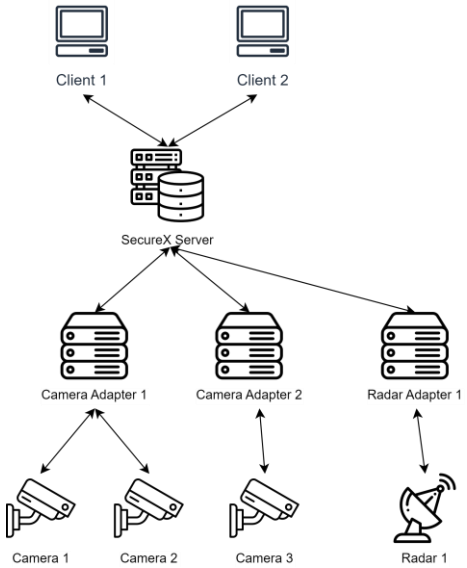

Figure 2. Deployment model of SecureX.

When a user wants to perform some action with a sensor, after pressing a button in the SecureX GUI Client, a message will be sent to the SecureX Server. Then, the server delegates this message to the adapters and other servers that are hierarchically under that server. The message arrives at the sensor's adapter and, according to the Interface Control Document (ICD) used in its integration, a message would be sent to the sensor to perform the desired action. Events and detections caught by the sensors would follow the reverse route and find their way to the SecureX GUI Clients.

SecureX is developed using the OSGi framework, which is a Java framework to develop modular software [10]. It is a platform in which manufacturers and developers can use as a software component framework. It is a versatile deployment API that can manage the life cycle of applications.

A. OSGi

The OSGi framework, based on its specifications, is a framework that can be used for creating highly modular Java systems. With its component model, it is a very reasonable candidate to be used in software product line development. It provides a simple way to change software components not only without a need to rebuild the entire system, but also dynamically changing them at the runtime. This shows the main capability of OSGi that simplifies the development of variation points, which is a crucial aspect of software product line architecture. The components are called "bundles" in the OSGi world, and the framework provides methods for installing, uninstalling and updating those bundles [11]. The life cycle that each bundle undergo in the OSGi framework can be seen in the Figure 3.

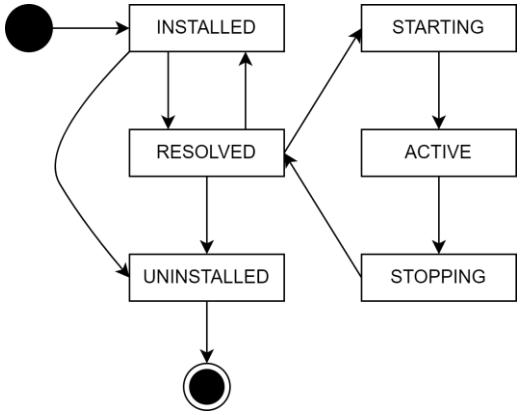

Figure 3. OSGi bundle life cycle. [11]

Every application that runs on the OSGi framework is expected to be able to immediately respond to the component changes at runtime. Any component might get an update or gets installed at runtime and the application that uses the component must properly react to this change and migrate to the new component. OSGi is a dynamic environment that expects applications to catch up to its changes.

Every bundle in an OSGi application has a start level that is defined in the bundle configuration files. When an application that runs on the OSGi framework starts, its bundles get initialized in the order of their start levels. SecureX uses this ordered initialization procedure and runtime bundle installation capabilities to optimize its initialization time by only installing the key bundles at first and installing the remaining bundles at the runtime.

IV. Existing Adapter Architecture

There are several adapter applications developed for different types of sensors such as Camera Adapter, Radar Adapter or Seismic Adapter etc. Their working principles are quite similar. The SecureX Server connects to the adapters and the adapter connects to the sensors. To segregate the sensor integration, we must first analyze the existing adapter architecture.

A few of the bundles in the Camera Adapter program can be seen in Figure 4. SecureX uses this framework to take advantage of its service architecture. We use the Camera Adapter application to describe the adapter architecture, but all adapter applications of SecureX are quite similar.

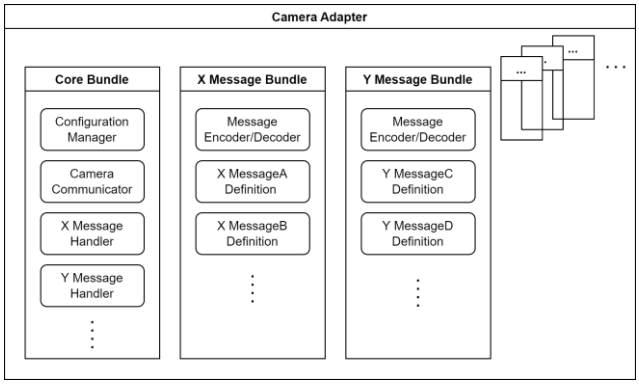

Figure 4. Simplified Camera Adapter model in the existing architecture.

The Camera Adapter application consists of many OSGi bundles whose purposes vary from providing network connection interfaces or utility tools, to message definition of sensors. These message definition bundles contain the methods for encoding and decoding messages to and from the sensor. Generally, the message formats for each sensor are different. They have different data types, header types, checksum calculation methods, big or little endian formats. Some sensors accept JSON formatted string messages, and some require encoding messages in a certain length byte array and sending them. Information about how to communicate with a sensor is given in its ICD. A message bundle is basically an implementation of the related ICD.

The *Configuration Manager* class in the *Core* bundle is mainly responsible for opening a Transmission Control Protocol (TCP) port to accept incoming server connections and initializing the *Message Handlers*. Each sensor's type, model, unique identifier key and required information about establishing a connection to it is written in a configuration XML file. The *Configuration Manager* constantly iterates over these files, creating a *Camera Communicator* and a specific *Message Handler* for every new or updated file. Messages are received by the TCP server and forwarded from there to the *Camera Communicator* and lastly to the sensor's *Message Handler*.

A *Camera Communicator*, which extends from the *Sensor Communicator* class as in every other sensor family, is the class where the processing of messages that came from the server starts. It handles generic messages or preprocesses them before the messages arrive at the *Message Handler*. When a message is received from the server, it is added to the message buffer of every active *Camera Communicator* in that adapter. *Camera Communicators* takes this message and decide if this message is meant for their sensor. To do this, they use the sensor identifiers in the messages. If the identifier is the same as the *Message Handler* they have, the message gets processed as will be explained in the subsequent paragraph, otherwise it is discarded.

The processing of the messages starts at the *Camera Communicator* level. Some messages are not specific to different sensor integrations and can be handled at the *Camera Communicator* level. Alternatively, some messages require a preprocessing step such as transforming some variables before they get forwarded to the *Message Handler*. After the initial processing is done, the *Camera Communicator* sends the message to the *Message Handler*.

The *Message Handler* is where the connection to the sensor is established using the protocol the sensor uses, which could be TCP, User Datagram Protocol (UDP), WebSocket, serial port, (Representational State Transfer) REST or any other network connection method that is stated in its ICD. The *Message Handler* knows how the connection should be established and how the incoming and outgoing messages should be processed. It receives the incoming message from the communicator and sends necessary commands to the sensor. The *Message Handler* needs a utility bundle to do the message conversions. When it needs to encode/decode messages to/from the sensor, it uses the Message bundle of that sensor that contains the message types, formats, checksum methods and the information of exactly how a message should be generated. After a message is generated, the *Message Handler* sends it to the sensor using the connection interface.

V. The Integration Problem

To keep up with the new and updated sensors to be integrated, and changing customer needs regarding sensor types and capabilities, sensor integration must be segregated and can be developed and updated independently. After analyzing the adapter architecture in the previous chapter, we can focus on what makes it difficult to integrate sensors in the current architecture.

When the adapter starts, the *StartLevelEventDispatcher* thread in the OSGi framework initializes all bundles that are marked for auto-start in the bundle configuration file. In Figure 5, initialization of the *Core* bundle is shown. The *Core* bundle is the one that starts the main Camera Adapter process with its thread "*ConfigurationMonitor*". In the initialization of the *Core* bundle, a single *Configuration Manager* instance gets created. The *Configuration Manager* then opens a port to listen to incoming SecureX Server connections. After that, it starts a thread that periodically checks sensor configuration files to find new or updated configurations. If there is such a file, then the *Configuration Manager* creates a *Camera Communicator* and the *Message Handler* for that sensor. In the existing architecture, in order to create a *Message Handler* instance, the *Configuration Manager* has to know which *Message Handler* needs to be used for which sensor configuration. In the configuration file, the identifier of the correct *Message Handler* is given, and the *Configuration Manager* uses that identifier to construct the *Message Handler*. But these *Message Handler* classes are inside the *Core* bundle and the *Configuration Manager* has a class dependency for them. This is the root problem in the current architecture.

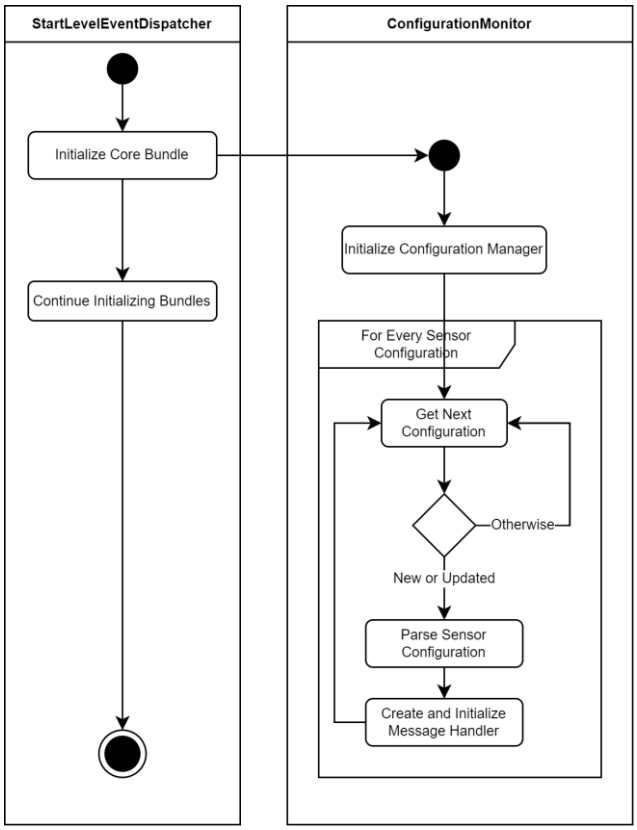

Figure 5. Message Handler initialization in the existing architecture.

A. *Difficulties with the Existing Architecture*

In order to carry out a new sensor integration, the message definition bundle has to be added in the Camera Adapter product file and its *Message Handler* has to be included in the *Core* bundle. The *Configuration Manager* class needs to know with which configuration identifier the new *Message Handler* should be constructed beforehand, hence the dependency. Because of this design, integrating or updating the integration of a sensor requires updating the *Core* bundle in the adapter. The components in the *Core* bundle, such as *Configuration Manager* and *Camera Communicator*, are used in every *Message Handler* and need to be compatible with all of them too. Therefore, any change in those components in the integration of a sensor could affect the already integrated sensors and cause them not to function as intended. Alarms detected by the sensor might start not to be forwarded to the server or changing the orientation of the sensor becomes difficult because of a change in some movement speed calculations.

In the current design, to update an already deployed system, a complete new build needs to be generated and tested. But the regression testing of the previous sensor integrations is not always easy or even possible. These sensors could be produced in very limited numbers, and they can only be found in the customer's facilities, working with the previous SecureX version. The location of these facilities might be difficult to access too and trips to these locations are not only costly, but sometimes, also dangerous. Because these sensors are almost always used in closed networks, the only way to test them is by going to these facilities, increasing the cost of testing. Also, customers would not want testers to separate these sensors from the PSIM system to test with the new version, creating a window of vulnerability.

Even if the tests are somehow completed, the update procedure has its own problems. To quickly update systems used in remote locations with little to no network access, or used in thousands of mobile locations without stable internet access, the update size must be minimal. But, with the current architecture, the whole adapter build needs to be updated, rather than just a couple of bundles.

Also, to catch up with new and updated sensors or security systems, 3rd party companies are employed for integrations. But this process is done through signing a Non-Disclosure Agreement (NDA) and sharing substantial parts of the adapter code with them to be used to integrate the sensors. Any one of them could expose the code at any point and this indeed is a security vulnerability.

Because of these reasons, there is a need for an architecture that ensures that the new integrations will not affect the existing ones. The main problem with the current design is, for every new integration, it has a need to update the *Core* bundle. The reason for that is the *Configuration Manager* class needs to know all available *Message Handlers* and for what kind of sensor they need to be used beforehand via class dependencies. In the new architecture, this problem is targeted with the aim to reduce testing overhead, reducing the amount of code that is shared with 3rd parties and also enables updating the deployed systems with small amount of data.

VI. New Adapter Architecture

To solve the problems with the existing architecture, a new adapter architecture shown in Figure 6 is developed. With this new architecture, all *Message Handler* classes moved to their message definition bundles and an OSGi service called *IMessage Handler Provider Service* that

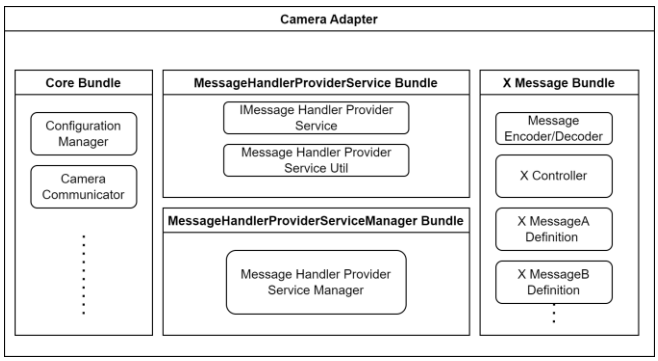

Figure 6. Simplified Camera Adapter model in the new architecture.

provides a *Message Handler* constructor for a given configuration identifier is developed. With that change, now the *Core* bundle does not depend on the *Message Handlers* or message bundles, but it depends on the *Message Handler Provider Service* bundle. Message bundles also depend on this service bundle too. This fixes the problem of the *Core* bundle depending on *Message Handlers* and its need to be updated to include a dependency with every new sensor integration. These message bundles, similar with every other OSGi bundle, can be extracted as a compiler .jar file and be installed externally.

Figure 7 shows the new classes and their hierarchies while Figure 8 shows the new message handler initialization procedure. The *Message Handler Provider Service Manager* implements the *IMessage Handler Provider Service* interface and when it is initialized by the *StartLevelEventDispatcher*, it reads a directory in which the new sensor integration bundles are placed as .jar files. The manager installs those new integrations and after the initialization of every new bundle, it registers itself as an instance that implements the *IMessage Handler Provider Service* interface to the OSGi context.

While those bundles are initialized, they register themselves with the *IMessage Handler Provider Service* in the OSGi context using the configuration identifier to indicate the sensor they should be used for. Accessing the registered *IMessage Handler Provider Service* is made possible through the *Message Handler Provider Service Util* class. This access technique blocks the requester thread until a service instance registers. The *Message Handler Provider Service Manager* registers itself after it initializes every integration file. Because *Message Handlers* access this manager using the same blocking technique, they can only register themselves after the service manager finishes its job. This causes all *Message Handlers* to register almost simultaneously.

While this process continues, the *Core* bundle also starts by the *StartLevelEventDispatcher* thread and continues its regular processes. But this time, the *Configuration Manager* class does not know any *Message Handler* itself. The dependencies for *Message Handler* classes are removed. When the *Configuration Manager* reads a sensor configuration, it uses its configuration identifier and asks a *Message Handler* constructor from the registered *IMessage Handler Provider Service*. It uses the *Message Handler Provider Service Util* class to access the service, so it also waits until an *IMessage Handler Provider Service* finishes its initializations and registers itself. After that, if a *Message Handler* for a given configuration identifier exists in the application, the *Configuration Manager* uses its constructor to create an instance and initialize it. The initialized *Message Handler* connects to the sensor and starts its regular processes. If a *Message Handler* does not exist for that identifier, the *Configuration Manager* skips that configuration for this iteration.

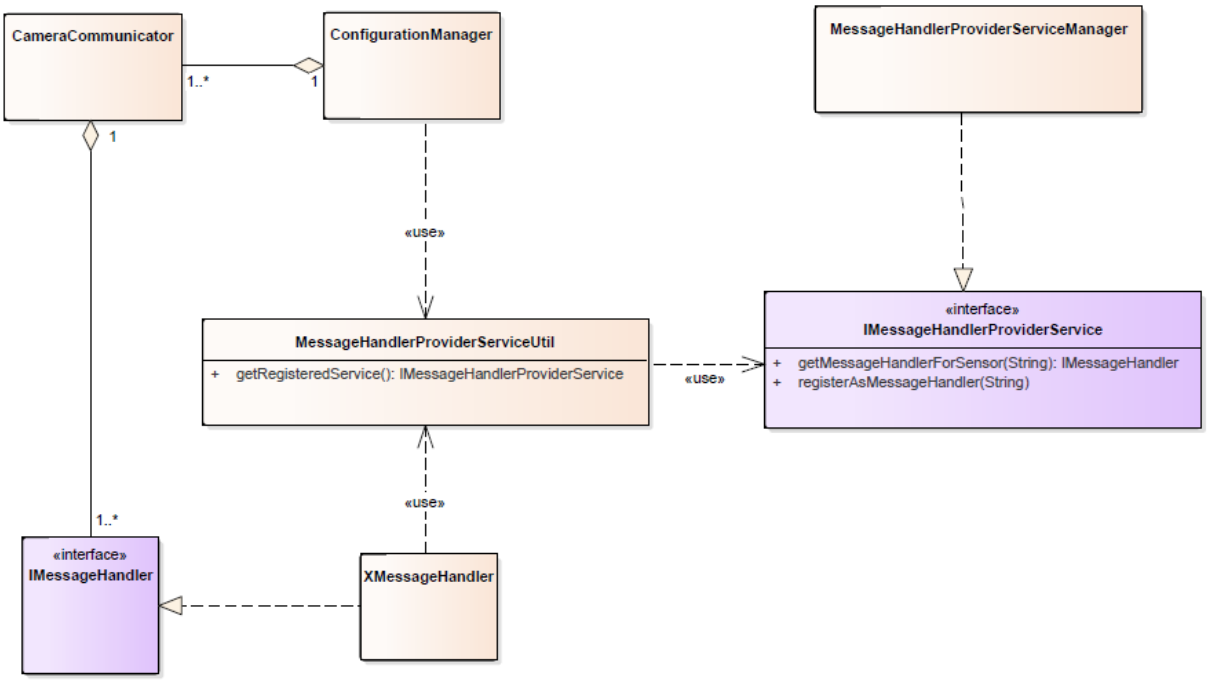

Figure 7. Camera Adapter Class Diagram (Simplified).

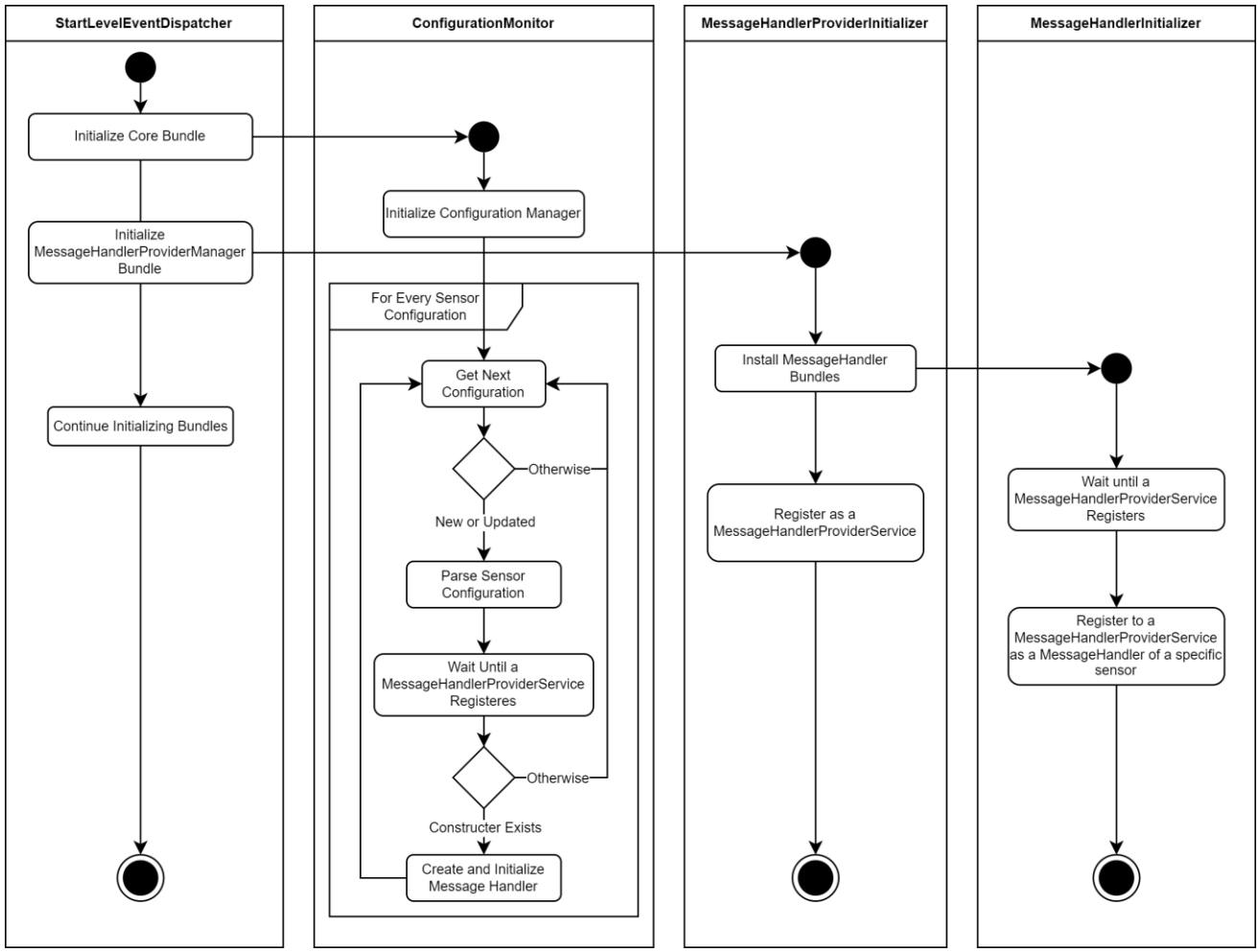

Figure 8. Message Handler initialization in the new architecture.

VII. Experiments And Survey

Similar to any other PSIM system, SecureX does not tolerate slow performance. It must provide a quick response capability for its users. Therefore, the architecture change must not cause a performance drawback. Also, to justify this architecture change, the new system must lower the test costs, as this was one of the promises of the new architecture.

A. Performance Tests

The main difference between the old and new architectures is the initialization of the adapter. As shown in Figure 7, the new initialization procedure is more complicated than the old one, which is shown in the Figure 4. Comparing those two diagrams, the difference mainly resides in how the *Configuration Manager* gets access to the *Message Handler* constructers. In the old architecture, *Configuration Manager* and all *Message Handlers* are in the Core bundle. Therefore, when *StartLevelEventDispatcher* initializes the Core bundle, every *Message Handler* class gets initialized along with the *Configuration Manager*. This enables *Configuration Manager* to access *Message Handlers* instantly without any additional dependency.

In the new architecture, *Message Handlers* are initialized in their separate bundles, and they register themselves to the *Message Handler Provider Manager*. *Configuration Manager* uses *Message Handler Provider Manager* to access the *Message Handler* constructers. This additional step causes a delay in the initialization phase of the adapter. But after the initialization is completed, any extra delay in other parts of the adapter is not expected. The tests confirm this hypothesis.

With the old architecture, time it takes to start the *ConfigurationManager* thread and for it to generate a *Message Handler* instance is on average 30 milliseconds, ranging between 29 and 31 milliseconds. In the new architecture, the average time for the same part of the initialization phase takes about 96 milliseconds, ranging from 88 to 104 milliseconds. This increase in time is the obvious result of not accessing the *Message Handler* constructers from within the same bundle and using an OSGi

service to do so. Both the *Message Handler*'s registration to the *Message Handles Provider Service*, and the service's own registration to the OSGi context takes time. But *Message Handler Provider Service Manager* keeps the *Message Handler* constructers in a map to easily access them if the *ConfigurationManager* needs it again. Therefore, this increased initialization time only happens on the first access of the constructer of a *Message Handler*. If there is another sensor of the same type in the system, the previous constructer gets used for the initializing of its *Message Handler*. But for a different type of sensor, another constructer must be generated.

This one time per sensor type increase in *Message Handler* initialization is trivial and it has little to no effect on SecureX's effectiveness.

After the *Message Handler* initializes, its operations such as processing, sending and receiving messages do not change between old and new architectures. On both architectures, typical message processing took about 3 milliseconds. This duration is the time between receiving a message from the SecureX server and after processing it, sending a notification to the server. So, the runtime performance of the adapter seems to be unaffected by this architecture change.

These tests show that the new design does not come with a significantly low performance. Increase in the initialization time is insignificant and hard to notice in the everyday use.

B. Survey for Cost of Testing

Another claim of the new design is that the test costs for a new sensor integration is high and the reason for this is the new bugs of the previously tested systems. We surveyed the testers who participated SecureX sensor integration tests to find out if that claim is true.

We have surveyed testers using in-depth interviews to understand the challenges in the SecureX tests. All nine testers who took the survey have at least one year, four of them has over three years of experience testing the SecureX sensor integrations. All participants had tested different cameras, radars, acoustic and seismic detectors. The used question set can be seen in Table 1. Based on their responses, on average, testing a camera or acoustic sensors takes about two hours, while a radar or seismic detector takes four hours. These test durations are not to be expected to be reduced by the proposed architecture change. New design does not provide a way to test one sensor faster, but it reduces the number of sensors to be tested after each new integration.

Table 1. Survey Questions

ID	Question
1	How long have you been testing software?
2	How long have you been testing SecureX?
3	What type of sensors did you test?
4	How many different cameras did you test?
5	How many different radars did you test?
6	How many different seismic detectors did you test?
7	How many different acoustic detectors did you test?
8	How long does it take to test a camera?
9	How long does it take to test a radar?
10	How long does it take to test a seismic detector?
11	How long does it take to test an acoustic detector?
12	In the last year, how many times was it necessary to go to the test site or the location where the system is installed to perform the test?
13	In the last year, how many times an intercity travel was necessary to reach to the test site or the location where the system is installed to perform the test?
14	How long does it take to go to the test site or the location and where the system is installed to perform the test?
15	In the last year, when a new sensor integration is tested, how often was it necessary to test other sensors of the same type as well? Ex: After testing a newly integrated camera, testing the other cameras in the system.
16	In the last year, when a new sensor integration is tested, how often a new bug from other sensors of the same type is detected?
17	How long does it take to fix and re-test the new bugs of the previously integrated sensors?
18	How much the development and test cost increase if new bugs of the previously integrated sensors were to be detected?
19	What was the worst-case scenario you experienced about bugs in previous integrations or increased test iterations like?

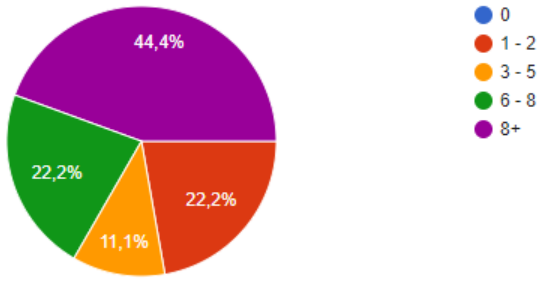

Figure 9. Distribution of the answers to Question 12.

Because the participants had worked on different projects that SecureX is used, their answers to the questions shown in Figure 9 and Figure 10 depend on those projects' test locations and configurations. If tests can be performed in house, and travel is unnecessary, the only test cost is the time spent testing the integrated sensors. When intercity travel is needed, the transportation and sometimes accommodation costs are added to the overall test cost.

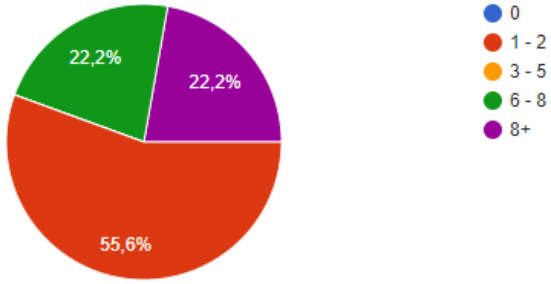

Figure 10. Distribution of the answers to Question 13.

But after testing the integrated sensor, tests for previously integrated sensors are needed; test durations for each of these sensors are also be considered when calculating the test cost.

Figure 11 shows that every participant stated that it was always necessary to test the previously tested sensors of the same type after a new sensor integration is completed. For example, after testing a newly integrated camera, testing other cameras in the system. Six of the testers said at least three times this was necessary and two of them said they had to test other sensors on more than eight occasions. Figure 12 shows the reasoning behind these additional tests. Often after

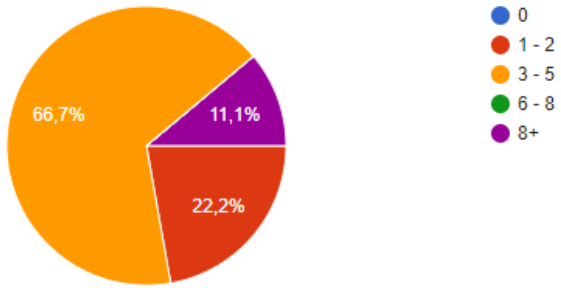

Figure 11. Distribution of the answers to Question 16.

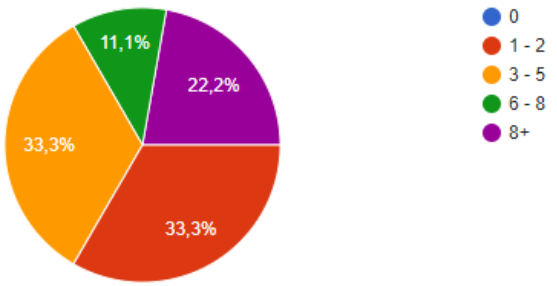

Figure 12. Distribution of the answers to Question 15.

a new sensor integration is completed, these additional tests reveal new bugs of the previously integrated sensors. Sensor integration in the old architecture does not segregate these integrations enough and provides an environment that is error prune.

These bugs extend the test and development duration as they need to be fixed and tested again. Also, the possibility of a bug occurring in the previous integrations cause testers to request testing those integrations whenever a new integration gets completed. As shown in Figure 13, this extra test and development process comes with an average cost increase of 30% to 50%, depending on the project configuration and location.

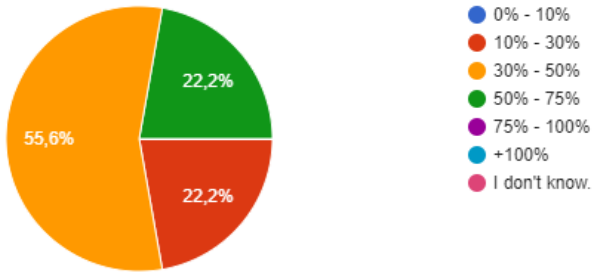

Figure 13. Distribution of the answers to Question 18.

The participants also asked what was the worst-case scenario that they experienced about sensor integration tests. Testers also point out that after the development and bug fixing processes for the bugs in the integration of sensors of same type, it was observed that the previously acquired and tested capabilities from other sensors were lost. This situation creates the need to review the integrations repeatedly and retest them after each bug fix. On one occasion, a SecureX system was installed at a remote location and is used by the operators when the customer wanted a new camera to be added for their changed security needs. The camera integration completed and tested at the company. But because the SecureX configuration the customer uses contains sensors not available at that moment during the tests of the new sensor, testers had to go to the location that SecureX system is installed. They, along with a developer, tested other sensors that the customer uses in order to verify that they still function as before. Testers found a couple of bugs and the developer fixed them and

after the re-test of the system, it is left to the operators again. During these tests, controls of the tested sensors are taken from the operators, and this lowers the PSIM system's availability.

As these tests take place in the customer's deployment, they may give customers a bad reputation about SecureX. Because for any bug that is found in those tests, there is a chance that it can be seen by the customer, or the security personnel at the location. Because it is not always possible to complete the tests without involving anyone from the customer's company. Usually, tests take place in the same room that the security personnel use and anyone would be curious to see what the new sensor can do. Even if the found bugs are minor or hard to reproduce in typical usage of the system, it would not matter if the customer realizes those bugs as well. If this situation keeps happening, reputation of SecureX would start to decline. Also, these security personnel or the customers themselves would mention their own requests from the PSIM system. Without anyone from the project management, the testers and developers are not always expected, to discuss the details of those requests.

Participants noted that for regression tests, having to go to remote locations where the existing systems are installed is costly and a way to reduce those costs are needed.

VIII. CONCLUSION

The proposed adapter architecture allows us to integrate additional sensors into the already deployed PSIM systems, without requiring to generate another complete build of an adapter software. Because previous integrations are not touched, integration tests of only the newly integrated sensors would be sufficient. When the sensor is integrated, it will most probably be available for testing as well and going to the field for using the sensor of a customer in order to conduct the tests will no longer be needed.

The survey with the testers has showed the extra work that needs to be done because of the new or reoccurring bugs in already tested sensor integrations. With the new architecture, these additional tests are no longer a regular requirement that takes place every time a new sensor gets integrated to the SecureX. Testing the sensor integrations only at their own integration times and keeping them bug free, even if new sensors or systems added to the project is crucial. An additional capability to a system should not take away or break the already existing and used capabilities. The cost of re-testing previously tested system after every integration is not something to be ignored. In addition to the amount of man hour being wasted for these tests, the financial cost also includes the logistic costs, which is depended on the test location. Therefore, the training that seemed to be required to work with this architectureas Almedia E. et al. found in their work [9], require a cost that is not worth mentioning of, when those additional test costs are considered.

The .jar files of the integration bundles are smaller than one MB. Thanks to these low sized components, system updates can be completed even with unstable or slow networks. Even if new sensor integrations have a problem working with previously integrated sensors, simply removing the .jar file would be enough to revert back to the previous deployment.

Segregating sensor integration also enables easily selecting and combining different integration bundles according to the project's requirement, as one could expect from a system developed with software product line principles. When starting a new project, depending on the sensors that are going to be used, only their integration files can be used. There is no need for adding every sensor integration to the project. The new design also enables employing 3rd party companies for integrations without sharing the bulk of the adapter code. Now, any integrator can develop an integration bundle only with the *Message Handler*, *IMessage Handler Provider Service* and the *Message Handler Provider Service Util* classes.

The proposed architecture is also shown to have similar performance with its predecessor with only a minimal delay at startup. Even if this startup duration increase was much more, if it's not extreme, it still might not be a problem. Generally, PSIM systems are not expected to shutdown and startup frequently. Due to high availability requirements, they tend to be designed as if they were expected to run continuously. So this minor increase in the initialization time can easily be ignored. Also, with the new architecture, stopping the system for adding a new senosr integration or changing an integration file is not required. New integrations can be added or updated while the system is running. This new capability also lowers the amount of times that the system had to be restarted. After the system is restarted and initialization is completed, the performance of the system was the same as it was with the old architecture. Only thing that the new architecture changes is the way sensor integrations are initialized.

The new architecture provides a helpful pattern towards transforming SecureX into a Software Product Line (SPL). An external .jar installer service could be used not only for sensor integrations, but also for features such as additional GUI views or in the server, new alarm evaluation algorithms. Because every feature is developed as an OSGi bundle, they all could be externalized. The sensor integration problem could be solved by developing an SDK, similar to the products given in Section II, but our design also eliminates the need of deploying the SecureX with a full feature set and stripping it off with configuration files at runtime. As this design gets implemented in other parts of SecureX, they could all be removed from the base build and can be added per customer demand.

The new design opens an evolutionary path for segregating such different aspects in SecureX architecture and is expected to be even more beneficial in the future. As such, the architectural change is not only applicable to PSIM systems like SecureX, but also any system that is developed with OSGi. Because at its core, our work can be described as a case study in how a software product line architecture can be implemented in an OSGi based product. What we achieve is within reach of any similar product.

We altered the existing architecture and took advantage of the OSGi framework to improve the modularity of our system. The modularity we achieve is a crucial requirement

for an SPL architecture, as SPL products are actually a combination of features selected from a feature set to satisfy particular requirements. These features can be developed in the same manner the sensor integration jar files are developed in the new architecture. And feature selection can be completed by using different feature jar files for different requirements. *Message Handler Provider Service Util* class and *IMessage Handler Provider Service* interface gives an example on how to select and use different features as well. Therefore, the architecture we proposed can be used in any software product line project.

REFERENCES

[1] O. Özçelik, M. H. S. Oğuztüzün, "Software Architecture Evolution of a Physical Security Information Management System". The Eighth International Conference on Advances and Trends in Software Engineering (SOFTENG), 2022, pp. 15-20.

[2] M. L. Garcia. The design and evaluation of physical protection systems. 2nd ed. Amsterdam: Elsevier, 2008.

[3] Genetec KiwiVision. [Online], retrieved May 2023 Available: https://www.genetec.com/products/

[4] Milestone XProtect. [Online], retrieved May 2023 Available: https://www.milestonesys.com/solutions/

[5] Nedap Aeos Access Control. [Online], retrieved May 2023 Available: https://www.nedapsecurity.com/solutions/

[6] Open Network Video Interface Forum (ONVIF). [Online], retrieved May 2023 Available: https://www.onvif.org/

[7] B. Tekinerdoğan, İ. Yakın, S. Yağız, and K. Özcan, "Product Line Architecture Design of Software-Intensive Physical Protection Systems". IEEE International Symposium on Systems Engineering (ISSE), 2020, pp. 1-8, doi: 10.1109/ISSE49799.2020.9272239.

[8] "The Java Language Specification, Java SE 8 Edition" J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. Apr. 2015. [Online]. retrieved May 2023 Available: https://docs.oracle.com

[9] E. Almeida, et al. "Domain Implementation in Software Product Lines Using OSGi". Seventh International Conference on Composition-Based Software Systems, 2008, doi: 10.1109/ICCBSS.2008.19

[10] R. S. Hall, K. Pauls, S. McCulloch, and D. Savage. "OSGi in Action - Creating Modular Applications in Java". Manning Publications, 2011

[11] "OSGi Service Platform, Core Specification, Release 8," The OSGi Alliance, April. 2018. [Online]. retrieved May 2023 Available: http://docs.osgi.org/specification/