Analysis of the Trade-Offs and Benefits of Using the Publisher-Subscriber Design Pattern

Technical Paper

Srihitha Yerabaka
Advisor: Prof. Ivan Marsic

Abstract

Design patterns describe a proven successful solution to problems that occur repeatedly. Using them promotes reuse of code and can help programmers get the right solution faster. In this paper, we apply cohesion and coupling metrics to two versions of a software application implemented with and without the Publisher-Subscriber design pattern to establish the benefits of using design patterns.

We apply the SCOM cohesion metric, a complexity measure - Structure101 and a Coupling Analyzer to our programs. We find that while using the Publisher-Subscriber design pattern does result in low coupling, it does not produce the best cohesion.

Table of Contents

1. Introduction

Design patterns describe a proven successful solution to problems that occur repeatedly. Using them promotes reuse of code and can help programmers get the right solution faster. This paper attempts to prove that programs written with design patterns are much more well-structured and efficient compared to code written without design patterns.

We implement a version of a home access control system described by Dr. Marsic in [2] with the publisher-subscriber pattern, and another version with out any design patterns but with the same exact functionality. We then apply cohesion and coupling metrics to these two versions of the home access control application to analyze the benefits and tradeoffs. We also apply Structure101 [18] to calculate the Structural Complexity of our code.

2. Background

2.1.Design Patterns

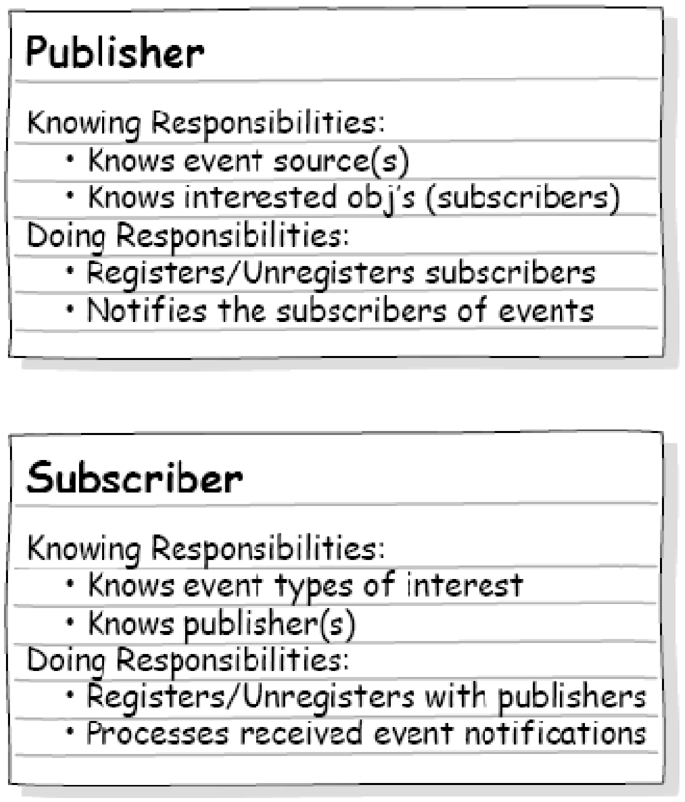
In their book [1], *Gamma et.al.* define a design pattern by four main components, the name, the problem, the solution and the consequences. The name refers to the name of any given pattern, the problem describes the situation in which the pattern is useful, the solution describes the design and how the elements in the design interact with each other and the consequences describe the results and tradeoffs that occur from using the pattern.

There are three types of design patterns, Creational, Structural and Behavioral. Creational design patterns describe object creation mechanisms. They try to create objects in a way that is suitable to the situation and does not result in increased complexity or create more problems. Structural patterns ease the design by identifying a simple way to realize relationships between entities. Behavioral patterns identify common communication patterns between objects and realize these patterns. This increases the flexibility in carrying out this communication.

Since we use the publisher-subscriber design pattern in our experiment, this paper will go more in-depth about this particular pattern. The publisher-subscriber pattern is a behavioral design pattern. In behavioral patterns, the assignments of responsibilities between objects are defined along with how different objects communicate with each other and a chain of responsibility is established.

With this pattern, individual modules do not need to know how other modules function or how they use the information provided by the other modules. For example, lets say that one module in a system gets information which is required for other modules to do their work. So, once this module acquires the information, it is expected to pass it on to the other modules. This means that the first module must know what the functionality of the other module is and send the information according to it. This is not desirable because changing either module will affect the other and hampers code reusability.

In order to eliminate this problem, we can use the Publisher-Subscriber pattern. With this pattern, the original module is coded as the publisher which just publishes the information it gets to its subscribers which in this case would be the other modules that need that information to their work. This way, neither module knows how nor what the other module does with this info. The responsibilities of publishers and subscribers are described in *fig.1* below.



Publisher

Knowing Responsibilities:
- Knows event source(s)
- Knows interested obj's (subscribers)

Doing Responsibilities:
- Registers/Unregisters subscribers
- Notifies the subscribers of events

Subscriber

Knowing Responsibilities:
- Knows event types of interest
- Knows publisher(s)

Doing Responsibilities:
- Registers/Unregisters with publishers
- Processes received event notifications

Fig. 1: Responsibilities of Publishers and Subscribers [2]

Since the publishers and subscribers are not needed to know the functionality or behavior of each other and they each only care about either publishing the information, or subscribing for information, they are not hindered if the other is changed or not working properly. This makes it a very loosely coupled system.

The pattern also makes the design easily scalable for larger systems, i.e., there is no limit on how many publishers or subscribers exist. The disadvantage of this design pattern however stems from the same feature that makes it desirable. For example, if the subscriber had crashed and did not receive the message the publisher published, there is no way for the publisher to know this since they are so decoupled.

To summarize, we define the Publisher-Subscriber design pattern in Table 1 below according to the definition provided in [1] as described above.

Component	Description

Name:	Publisher-Subscriber
Problem:	The Publisher-Subscriber design pattern is used to define a “one - to many” dependency between objects, so that when one object changes its state, all of its dependents are notified and updated immediately [1].
Solution:	Instead of allowing different modules in the system from learning and depending upon the inner workings of other modules, we define publishers and subscribers.
Consequences:	*Advantages:* 1. Decreases coupling in the software system 2. Easily scalable to bigger systems. *Disadvantages:* 1. Publisher just assumes that the subscriber is listening and has no way to verify if the subscriber actually received the information published. If the subscriber crashes, the published message could be lost for ever since no one receives or acts on it.

Table 1: Defining the four components of the Pub-Sub design pattern

2.2. Coupling & Cohesion Metrics

2.2.1. Coupling

Coupling metrics are a measure of how interdependent different modules are on each other. High coupling occurs when one module modifies or relies on the internal workings of another module. Low coupling occurs when there is no communication at all between different modules in a program. Coupling is contrasted with cohesion. Both cohesion and coupling are ordinal measurements and are defined as "high" or "low". It is most desirable to achieve low coupling and high cohesion.

We use a coupling analyzer [19] developed at Rutgers to calculate the coupling metrics for our programs. Given the main() class of an application, the analyzer parses through entire code and builds a class dependency graph.

2.2.2. Cohesion

Cohesion is defined as a measure of relatedness or consistency in the functionality of a software component. In an object-oriented paradigm, a class can be a component, the data can be attributes and the methods can be parts. Modules with high cohesion are usually robust, reliable, reusable, and easy to understand while modules with low cohesion are associated with undesirable traits such as being difficult to maintain, test, reuse and understand. Cohesion is an ordinal type of measurement and is usually expressed as "high cohesion" or "low cohesion".

We can find many formulas to calculate the cohesion metrics in literature. However, many of them either define the class in question as either "cohesive" or "not cohesive", they do not make a comparative analysis of the degree of cohesiveness. This makes it hard to compare two cohesive, or two non cohesive classes. Since we want to compare the cohesion of the two different versions of our home access control system program, it is necessary for us to select a metric that can calculate not just whether a module is cohesive or not cohesive but also the degree of its cohesiveness. Assuming both the versions of our home access control system are cohesive, this would enable us to fairly judge which version is better designed and more cohesive.

Many of these proposed metrics were compared by a few papers. However, as opposed to most of them, which only looked for weaknesses in the existing metrics and tried to improve them by eliminating their points, Joshi and Joshi [4] focus on the strengths of these metrics besides the weaknesses and find metrics that are suitable for different purposes such as detecting the disparate classes needing refactoring and detecting the degree of cohesiveness.

They compare a total of thirteen cohesion metrics that they classify into five types based on the methods of quantification of cohesion. The definitions of these types of classification are listed in Table 2.

Classification Name	Description
Disjoint component-based metrics	These metrics count the number of disjoint sets of methods or attributes in a given class.
Pairwise connection-based metrics:	These metrics compute cohesion as a function of number of connected and disjoint method pairs.
Connection magnitude-based metrics	This category does not compute direct sharing between methods, but it instead uses the count of accessing methods per attribute and indirectly finds a sharing index in terms of the count.
Decomposition-based metrics:	These metrics compute cohesion in terms of recursive decompositions of a given class. The decompositions are generated by removal of pivotal elements that keep the class connected.
Interface-based metrics	Metrics that compute class cohesion based on information gathered from method signatures are grouped under this category.

Table 2: Types of classification of cohesion metrics as described in [4]

Among the thirteen metrics that the authors look at, *LCOM1* [5], *LCOM3* [6] and *LCOM4* [7] are classified as Disjoint Component-Based metrics. *LCOM2* [8], *Relative Lack of Cohesion (RLCOM)* [9], *Cohesion Ratio (CR)* [10], *Tight Class Cohesion (TCC)* and *Loose Class Cohesion (LCC)* [11] are classified as Pairwise Connection-Based metrics. *LCOM** [12] and

SCOM [13] fall under Connection Magnitude-Based metrics while *Cohesion Based on Member Connectivity (CBMC)* [14] and *Improved Cohesion Based on Member Connectivity (ICBMC)* [15] are classified as Decomposition-Based metrics. Finally, *Cohesion Among Methods of Class (CAMC)* [16] and *Cohesion Metric (CM)* [17] are classified as Interface-Based metrics.

A set of four example cases, shown in figure 2, were used to capture the degree of cohesiveness. “m1”, “m2”, “m3” represent methods 1, 2 and 3 while “a1”, “a2” and “a3” represent three different attributes.

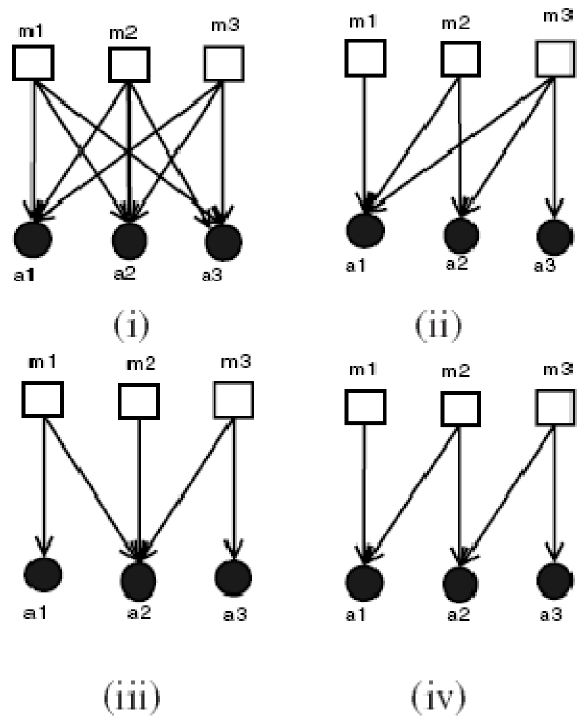


Fig. 2: Example cases for capturing Degree of Cohesion. [4]

The most cohesive class is shown in case (i); all of its methods use all three attributes. Case (ii) depicts the second most cohesive class. The class is still connected, but all methods do not use all the attributes. In case (iii) the cohesion is even lesser. In this class, the methods are connected through only one of the three attributes. Case (iv) is the least cohesive. None of its attributes are used by all three methods.

LCOM2	RLCOM TCC, CR	ICBMC LCOM*	SCOM	CBMC
i, ii, iii, iv	i,ii,iii	i	i	i
-	iv	ii	ii	ii, iii
-	-	iii, iv	iii	iv
-	-	-	iv	-

Fig. 3: Case ranks based on the Degree of Cohesion. [4]

Since Interface-Based metrics like *CAMC* and *CM* require the description of the interface and not the implementation, they were not tested for Degree of Cohesiveness. The LCOM metrics were also avoided since they only determine whether a class is cohesive or not and do not calculate the degree of cohesiveness. Among the eligible metrics that were tested, the authors found that *SCOM* was the only one to successfully differentiate between all four cases. Figure 3 shows how the cases were ranked on the degree of cohesiveness by the applied metrics. Based on the results shown in this paper, we choose the SCOM metric to calculate the cohesiveness of our programs.

Sensitive Class Cohesion Metric or *SCOM* [13] is defined as,

$$SCOM = \frac{2}{m(m-1)} \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} C_{i,j} * \alpha_{i,j}$$

where $\{\mathbf{A_1},\ldots,\mathbf{A_a}\}$ and $\{\mathbf{M_1}\ldots.\mathbf{M_m}\}$ are the set of attributes and methods of the class, respectively, and $C_{i,j}$ is the connection intensity and $\alpha_{i,j}$ is the weight factor . The coefficient of the summation, 2/(m(m-1)) is the inverse of the total number of method pairs and is used for normalization.

The connection intensity formula is defined as,

$$C_{i,j} = \begin{cases} 0, & if\ I_i \cap I_j = \phi \\ \dfrac{card(I_i \cap I_j)}{\min(card(I_i), card(I_j))} & otherwise \end{cases}$$

where $\mathbf{I_k}$ is denoted for the subset of attributes used by a given method $\mathbf{M_k}$ and "*card*" is the cardinality. The cardinality of a set is defined as the measure of the number of elements of the set. So, the connection intensity is 0 if there are no common attributes between methods "i" and "j" and the ratio of the total number of common attributes divided by the total number of attributes for methods "i" or "j", whichever is smaller.

The weight factor is defined as,

$$\alpha_{i,j} = \frac{card(I_i \cup I_j)}{a}$$

Where “*a*” is the total number of instance attributes in the class.

The SCOM metric is normalized to the range [0..1], where the two extreme values are 0 and 1. Value zero represents no cohesion at all and the value one represents full cohesion.

2.3 Structure101

According to [18] structural complexity is not a problem, in and of itself. They point out that while complexity may be inherent in the problem domain and require a complex solution, such a solution would have a rational structure and is understandable. A complicated solution, however is difficult to understand and analyze. The authors argue that when working with large, complex code bases, traditional object-oriented metrics, such as Coupling and Cohesion only offer a limited snapshot of the system complexity. They present Structure101 measurement framework which provides a comprehensive view of structural complexity of a software system.

This measurement framework is based on two aspects of excessive structural complexity, XS: Tangles and fat. It calculates the XS by multiplying the degree of violation due to tangles and fat by its size. The degree of violation due to tangles, which are cyclic dependencies between packages, is calculated by identifying the minimal feedback set (MFS) which is the minimum set of edges that must be removed from a dependency graph of a software system to eliminate cyclic dependencies. Structure101 divides the number of code-level references in the MFS by the total number of dependencies to obtain the degree of tangle violation.
The fat metric represents how easy it is to understand a given application. It is calculated as the number of dependencies in higher-level packages that contain multiple packages. For lower level packages, which contain classes, fat is computed by counting the total number of dependencies between classes. Degree of tangle violation and the degree of fat violation are then multiplied by the item’s size.

Structure101 then calculates the cumulative XS by recursively adding the local XS values for a particular item’s descendants. This gives us the percentage of the code base that is excessively complex.

3. Design & Implementation

We designed a Home Access Control System application to conduct our experiments. The system will control all aspects of a typical home security system such as lock control, lights, alarm and key control.

When a user wants to enter a house secured by this access control system, he presses a button which allows him to enter a password with a prompt. After the user enters a key code, the “KeyChecker” checks it. If the password is correct, the alarm is disabled and the door is unlocked. Also depending on the time of the day, the light turns on automatically. The user can be given a preset number of tries to enter the correct key code. If he/she has exhausted all tries and enters a wrong key code, the alarm stays armed and the alarm bell goes off.

Figure 4 shows the original design of the Home Access Control System as described in [2].

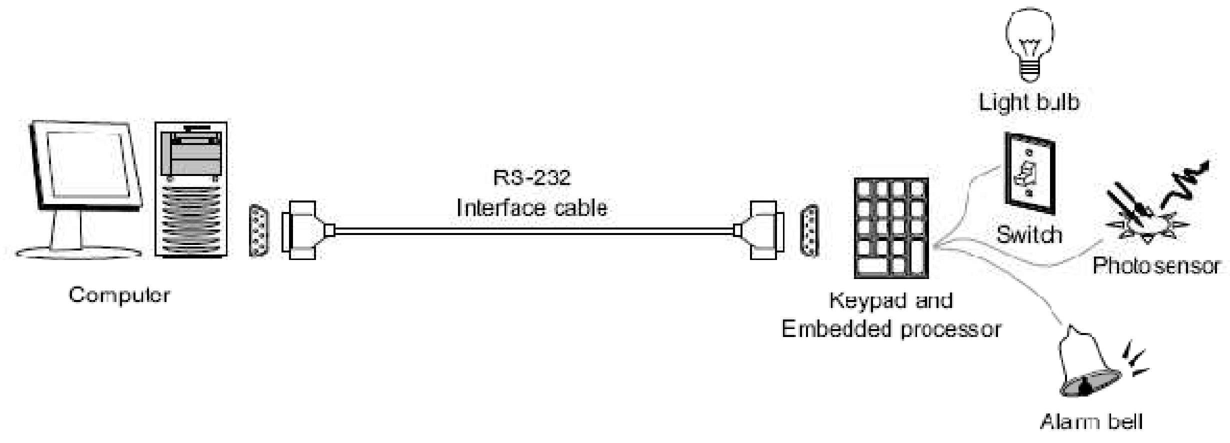


Fig. 4: The original Home Access Control System described in [2]

For practical reasons, we use a relay switch board to show the status of the light, alarm and key controls. Since we cannot implement an actual Photo Sensor, we instead use the system time to determine if the light should be turned on when the correct alarm key is entered. The embedded processor with the keypad is actually a java program run on a computer. We use a USB connection to the relay switch as a virtual COMM port. The Relay switch board will be discussed further in a subsection.

Figures 5 and 6 depict flowcharts describing what will happen when the key code entered is correct and when it is incorrect, respectively.

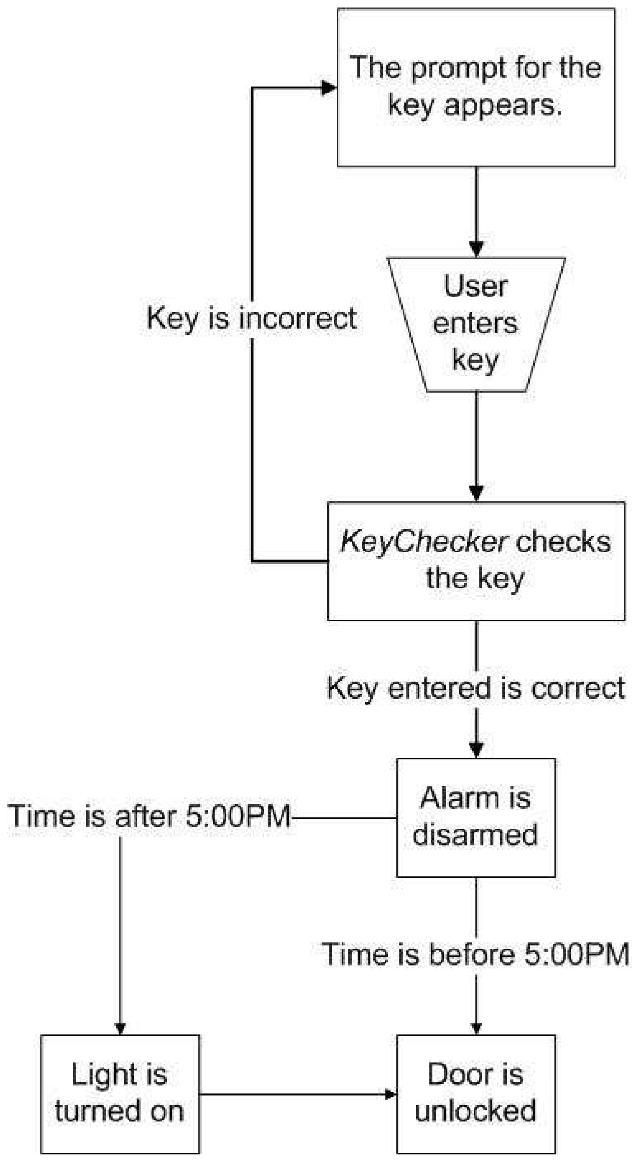


Fig. 5: Flow of the system when the Key is correct

We preset the Light Control cut off time to 5PM as a substitute for the Photo Sensor. The figure above shows the control going back to the beginning where the user is prompted for a key, however this only depicts the first wrong key entered. The figure below, solely concentrates on the system flow when the alarm key code being entered is incorrect.

The flow diagrams are a high level depiction of how the system acts and remain the same for both the version with the design pattern and the one without.

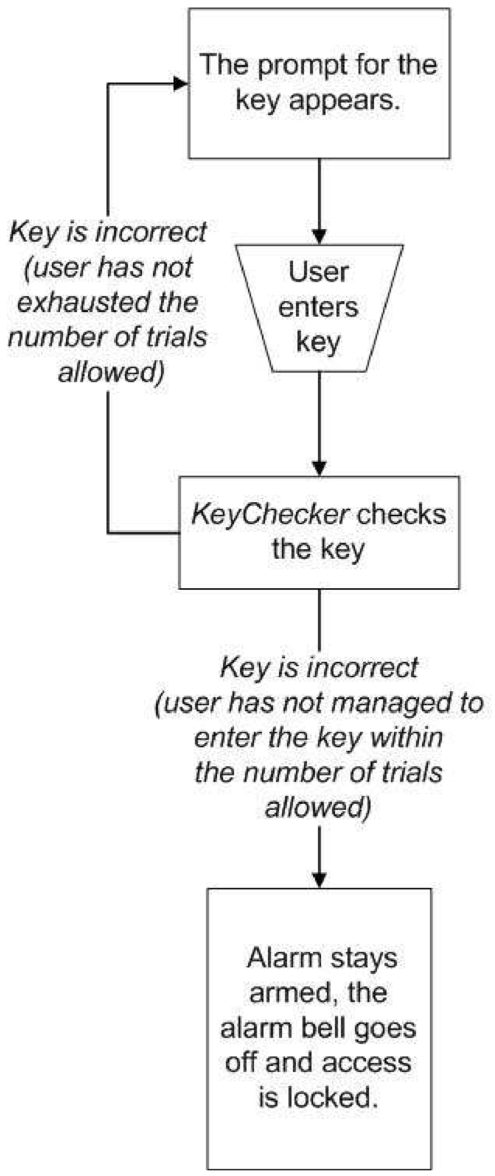

The prompt for the key appears.
User enters key
KeyChecker checks the key
Key is incorrect (user has not exhausted the number of trials allowed)
Key is incorrect (user has not managed to enter the key within the number of trials allowed)
Alarm stays armed, the alarm bell goes off and access is locked.

Fig. 6: Flow of the system when the Key is incorrect

3.1 Without Design Patterns

The Home Access Control System application contains 6 classes, The main class is *HomeAccessControlSystem*, this instantiates the *Controller*, *KeyChecker*, *AlarmCtrl*, *LightCtrl* and *LockCtrl*. The *HomeAccessControlSystem* establishes connection with the relay switch through the USB virtual COMM port and then calls the *enterKey()* method from the *Controller* to prompt the user for a Key.

The *Controller* is the entry point to the main domain logic. It calls the *checkKey()* method from the *KeyChecker* class which returns a boolean value representing whether the Key entered is correct or incorrect. If the Key is correct, the controller sets the *setArmed()* method from *lockCtrl* to false to disarm the alarm. It also checks the system time and sets the *setLit()* method from *lightCtrl* to true if it is after 5PM. The *openDoor()* method from *lockCtrl* is also set to true to indicate access is permitted.

If the Key entered is incorrect, the controller sets the *setArmed()* method from *lockCtrl* to true to keep the alarm armed and also sets the *soundAlarm()* method from *alarmCtrl* to true, to ring the alarm bell.

Code Snippets
```
public HomeAccessControlSystem( String key, OutputStream outputStream )
throws IOException
      {
            LockCtrl lkc = new LockCtrl(outputStream);
            LightCtrl lic = new LightCtrl(outputStream);
            AlarmCtrl ac = new AlarmCtrl(outputStream);
            KeyChecker kc = new KeyChecker (key);
            contrl_ = new Controller(kc, lkc, lic, ac);
      }
```
```
public class Controller
{
      protected KeyChecker checker_;
      protected LockCtrl lockCtrl_;
      protected LightCtrl lightCtrl_;
      protected AlarmCtrl alarmCtrl_;
      public static int maxNumOfTrials_ = 2;
      protected long numOfTrials_ = 0;
      String userkey;
      Scanner in = new Scanner(System.in);
      Calendar calendar = new GregorianCalendar();
      String am_pm;

      public Controller(KeyChecker kc, LockCtrl lkc, LightCtrl lic, AlarmCtrl
ac)
      {
            checker_ = kc;
            lockCtrl_ = lkc;
            alarmCtrl_ = ac;
            lightCtrl_ = lic;
      }

      public void enterKey()
      {
            while (numOfTrials_ < maxNumOfTrials_)
            {
                  System.out.println("Enter Alarm Password:");
```
```
public class KeyChecker
{
      private String validKey;
```

```
	private boolean isvalid = false;


	public KeyChecker(String OriginalKey)
	 {	validKey = OriginalKey; }

	public boolean KeyCheck(String userKey)
	{
```

```
public class AlarmCtrl
{
	protected boolean alarmRing;
	protected OutputStream sendout;

	public AlarmCtrl(OutputStream outputStream)
	{
		sendout = outputStream;
		alarmRing = false;
	}
	public void soundAlarm(boolean val)
	{
```

```
public class LockCtrl
{
	protected boolean armed_;
	protected boolean unlocked_;
	protected OutputStream sendout;

	public LockCtrl(OutputStream outputStream)
	{
		sendout = outputStream;
		armed_ = true; //alarm in armed by default
		unlocked_ = false; //door is locked by default
	}

	public void setArmed (boolean val)
	{
```

```
public class LightCtrl
{
	protected boolean Lighted;
	protected OutputStream sendout;

	public LightCtrl(OutputStream outputStream)
	{
		sendout = outputStream;
		Lighted = false; // light is turned off by default
	}

	public void setLit (boolean val)
	{
```

3.2 With Design Patterns

Since most operations are a based on the result from the KeyChecker, we design it to implement the publisher, and Controller, lockCtrl, alarmCtrl and lightCtrl implement subscribers. To use the publisher-subscriber pattern, we first need to define two types of class interfaces called Publisher and Subscriber. Since our subscribers will either need to subscribe to the event where the Key is valid or invalid, we define two subscriber interfaces, *SubscribeKeyIsValid* and *SubscribeKeyIsInvalid*. The Publisher interface has methods to subscribe to either of the two types of events, namely *KeyIsValid* and *KeyIsInvalid*.

The *Controller* and *alarmCtrl* implement the *SubscribeKeyIsInvalid* subscriber and register for the *KeyIsInvalid* event from the publisher. The C*ontroller* needs it to prompt the user again for a key when it is incorrect and the *alarmCtrl* needs it to ring the alarm bell.

lockCtrl and *lightCtrl* implement the *SubscribeKeyIsValid* subscriber and register for the *KeyIsValid* event from the publisher to disarm and provide access and to turn on the lights respectively.

Code Snippets (Implementation with Publisher-Subscriber)

```
public HomeAccessControlSystem( String key, OutputStream outputStream )
throws IOException
      {
            KeyChecker kc = new KeyChecker (key);
            LockCtrl lkc = new LockCtrl(kc, outputStream);
            LightCtrl lic = new LightCtrl(kc, outputStream);
            AlarmCtrl ac = new AlarmCtrl(kc, outputStream);
            contrl_ = new Controller(kc);
      }
```

```
public interface LockPublisher
{
      public void subscribeKeyIsInvalid(KeyIsInvalidSubscriber subscriber);
      public void subscribeKeyIsValid(KeyIsValidSubscriber subscriber);
}
```

```
public interface KeyIsValidSubscriber
{
      void KeyIsValid (int boolval);
}
```

```
public interface KeyIsInvalidSubscriber
{
      void KeyIsInvalid (int boolval);
}
```

```
public class Controller implements KeyIsInvalidSubscriber
{
	protected KeyChecker checker_;
	String userkey;
	Scanner in = new Scanner(System.in);

	public Controller(KeyChecker kc)
	{
		kc.subscribeKeyIsInvalid(this);
		checker_ = kc;
	}

	public void KeyIsInvalid(int boolval)
	{
		enterKey();
	}

	public void enterKey()
	{
```

```
public class LightCtrl implements KeyIsValidSubscriber
{
	protected boolean Lighted;
	protected OutputStream sendout;
	Calendar calendar = new GregorianCalendar();
	String am_pm;

	public LightCtrl(KeyChecker keyChecker, OutputStream outputStream)
	{
		sendout = outputStream;
		keyChecker.subscribeKeyIsValid(this);
		Lighted = false; // light is turned off by default
	}

	public void KeyIsValid(int boolval)
	{
			if ((calendar.get(Calendar.HOUR)>= 5) &&
(calendar.get(Calendar.AM_PM) != 0))
					setLit(true); //if time is after 5PM.
				else
					setLit(false); //if time is before 5PM.
	}

	public void setLit (boolean val)
	{
```

3.3 Relay Switch

We use a USB relay switch board from National Control Devices to function as the Home Access Control Hardware. We only use four of the available eight switches. The first switch represents if the alarm is armed. The second switch signifies the ringing of the alarm bell. The third switch represents the lights, and the fourth switch shows if access has been granted.

<u>*Scenario*</u>	***Relay 1 (Alarm system)***	***Relay 2 (Alarm bell)***	***Relay 3 (Lights)***	***Relay 4 (Access to home)***

Initial state	On (alarm system armed)	Off (Bell not ringing)	Off (Lights off)	Off (No access granted)
Correct Key -before 5PM	Off (system disarmed)	Off (Bell not ringing)	Off (Lights off)	On (Access granted)
Correct Key -after 5PM	Off (system disarmed)	Off (Bell not ringing)	On (Lights on)	On (Access granted)
Wrong Key	On (alarm system armed)	On (Bell ringing)	Off (Lights off)	Off (No access granted)

Table 3: State of the Relay Switches for possible scenarios.

The following figures show the status of the relay switches through a sample flow. The system is in the initial state and changes to scenario 2 when a correct key is entered before 5PM. The last few will show scenario 4, where the incorrect key is entered.

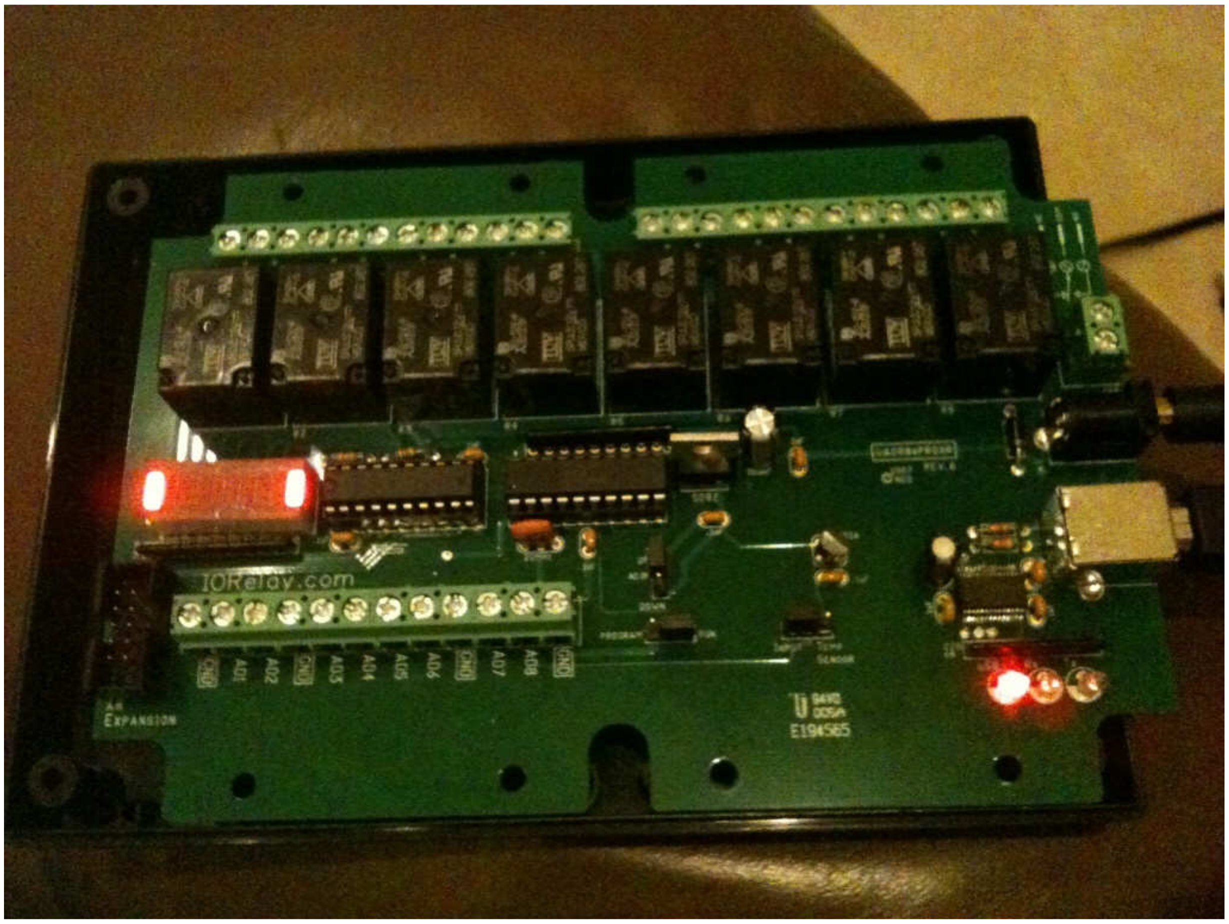


Fig. 7: Picture of the Relay Switch Board. The bank of switches can be seen in red on the left hand side. The LED on the right bottom corner indicates that it is a USB connection.

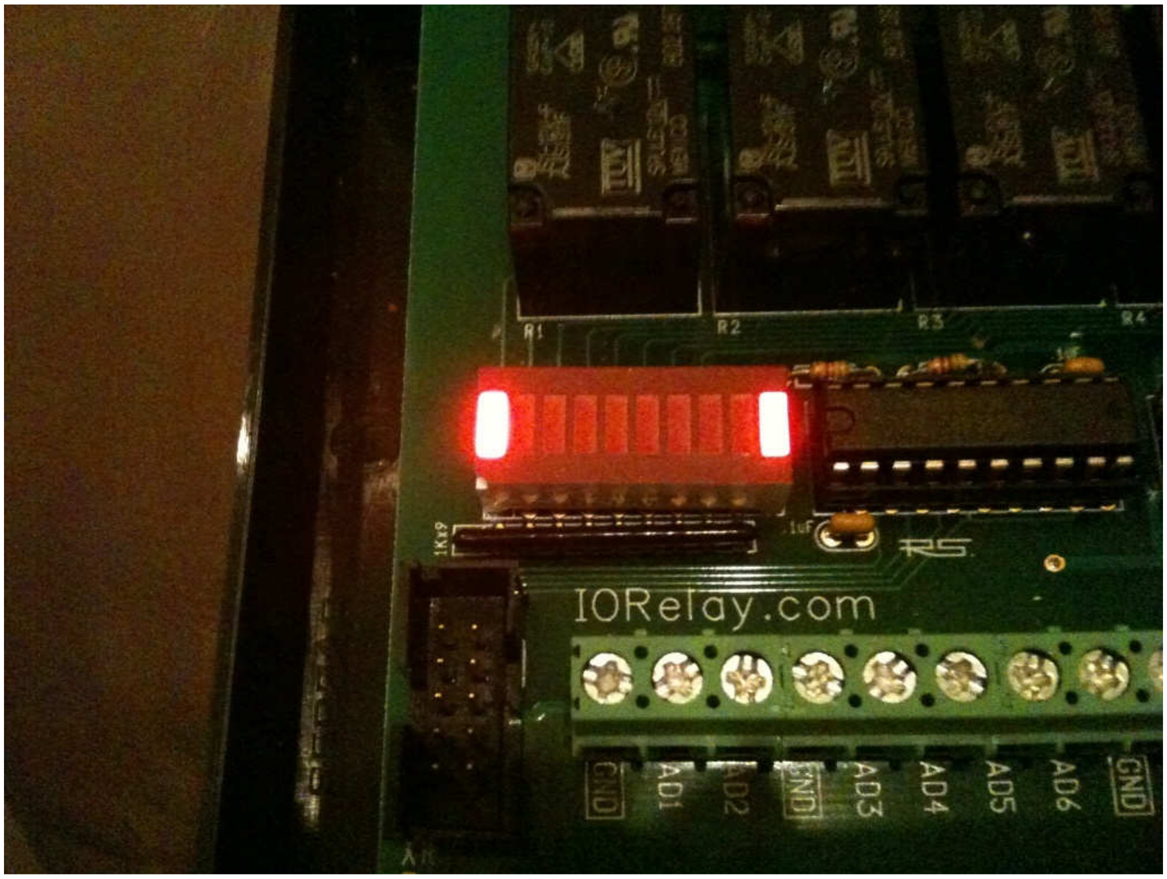


Fig. 8: In the Initial State. The red light on the left most side represents Relay 1, i.e. "Alarm is armed." A total of 10 LEDs can be seen, The first 8 represent the 8 Relay Switches. The 9th lights up when the board is receiving information and the 10th LED indicates whether this bank of Relays is operating correctly.

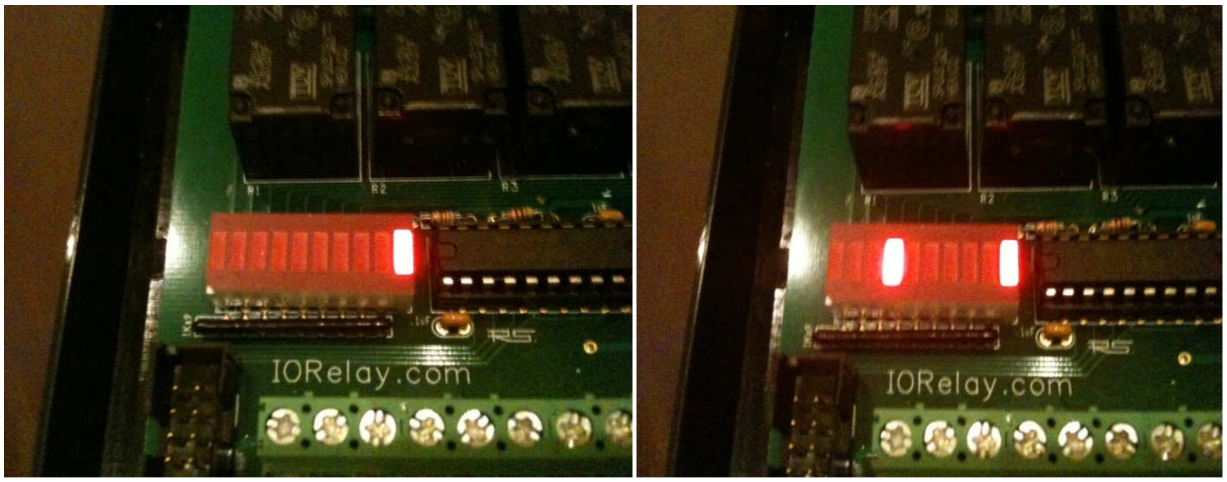


Fig. 9: State for scenario 2. The correct Key is entered before 5PM. The 1st relay turns off to indicate that the alarm has been turned off and relay 4 turns on to indicate access is granted.

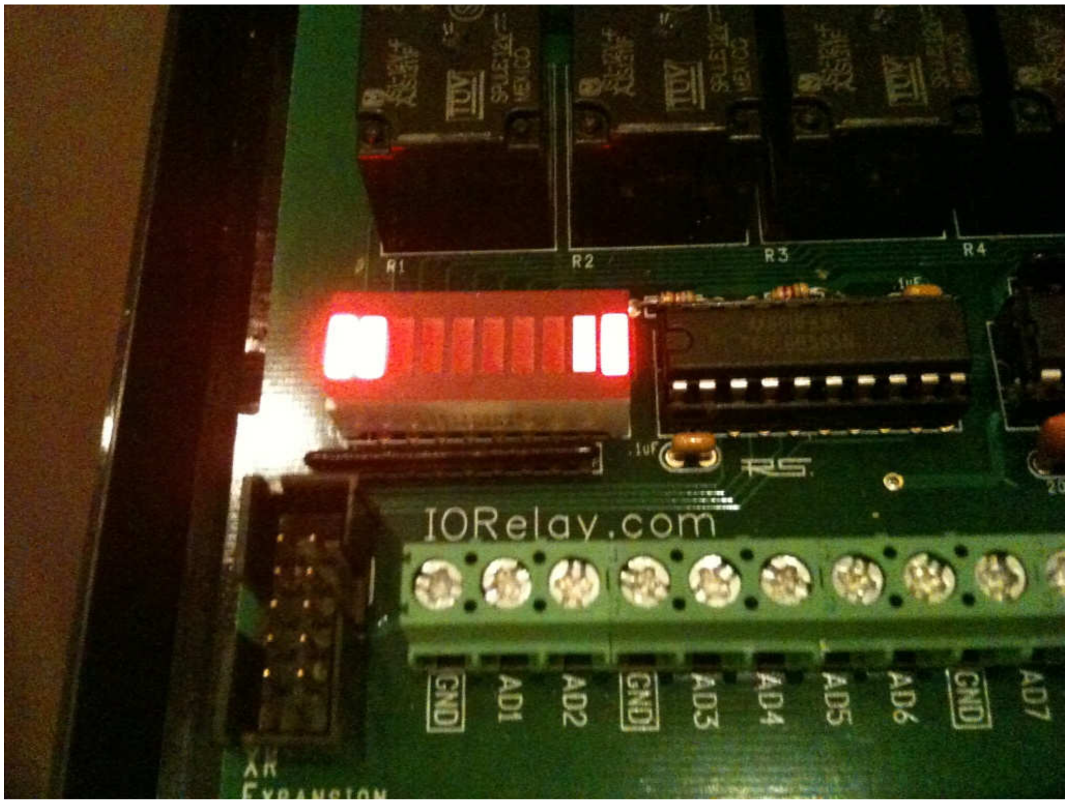


Fig. 10: State for scenario 4. The incorrect key is entered more than the allowed number of times while the system was in the initial state. Relay 1 stays on to indicate Alarm is still armed, Relay 2 turns on to indicate that the alarm bell has gone off.

The following code snippets show how the control is passed via the USB Virtual COMM port to the Relay Switch Board. The ASCII code "254" has to be sent first to setup the Relay board into Command mode when it will accept commands. Since there are eight relay switches, the ASCII codes 1 to 8 are used to turn off the eight relay switches and the ASCII codes 9 to 16 are used to turn on the eight relay switches [20].

Code Snippets

```
protected OutputStream sendout;
...
//System.out.println("Turning on Relay 2.");
System.out.println("Alarm bell is ringing.");
sendout.write(254);sendout.write(9);
...
//System.out.println("Turning off Relay 2.");
System.out.println("Alarm bell is not ringing.");
sendout.write(254);sendout.write(1);
...
//System.out.println("Turning on Relay 3.");
System.out.println("Lights are on.");
sendout.write(254);sendout.write(10);
...
//System.out.println("Turning off Relay 3.");
```

```
System.out.println("Lights are off.");
sendout.write(254);sendout.write(2);
...
//System.out.println("Turning on Relay 1.");
System.out.println("Alarm is armed.");
sendout.write(254);sendout.write(8);
...
//System.out.println("Turning off Relay 1.");
System.out.println("Alarm is disarmed.");
sendout.write(254);sendout.write(0);
...
//System.out.println("Turning on Relay 4.");
System.out.println("Door is unlocked. Access permitted.");
sendout.write(254);sendout.write(11);
...
//System.out.println("Turning off Relay 4.");
System.out.println("Door is locked. Access not permitted.");
sendout.write(254);sendout.write(3);
...
```

4. Evaluation & Results

4.1 Coupling Metrics

We can easily determine from observing the code that the version with the Publisher-Subscriber pattern has lower coupling compared to the original. In the original version, the *Controller* instantiates all the other classes and invokes their methods. This increases the dependency between these modules and thus increases coupling. With the Publisher-Subscriber, this dependency is avoided since the modules all subscribe to the *KeyChecker*. The *Controller* no longer needs or has access to the internal methods of the other modules. This reduces coupling.

The Coupling Analyzer [19] validates our claims. Figure 11 below shows the degree of connectivity for the classes in the Original code. One class has a degree of 4, another has a degree of 5 and the remaining four classes have a degree of 0. Comparing this with Figure 12, we see that the version with the Publisher-Subscriber pattern only has a degree of 1, another has a degree of 5 and the remaining are all 0. Figure 13, compares the class dependency trees for both versions. Because of the Publisher-Subscriber interface, The *Controller* is only connected to one other class, the *KeyChecker*.

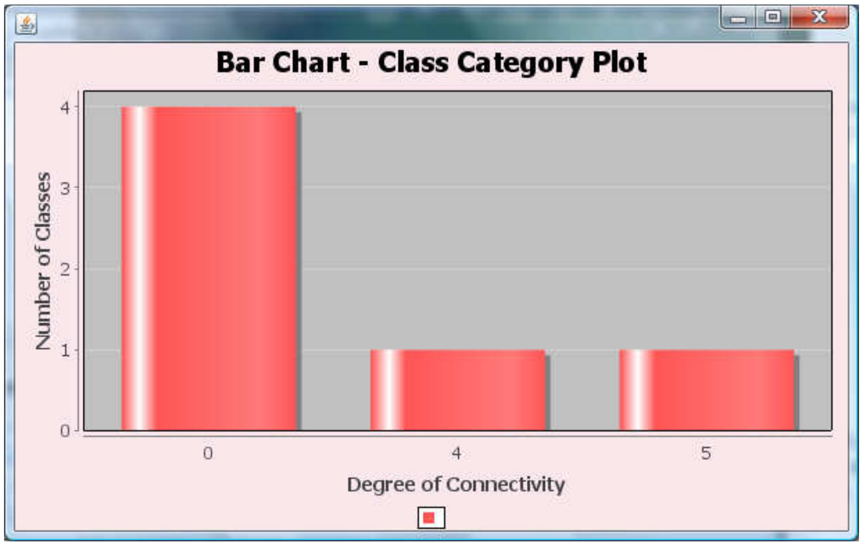


Fig. 11: Degree of Connectivity plot for original version without Design Patterns.

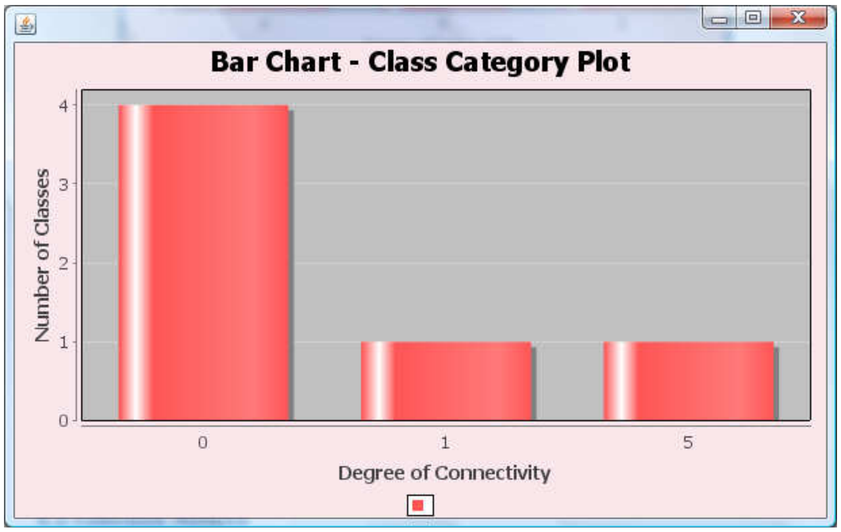


Fig. 12: Degree of Connectivity plot for the Publisher-Subscriber version.

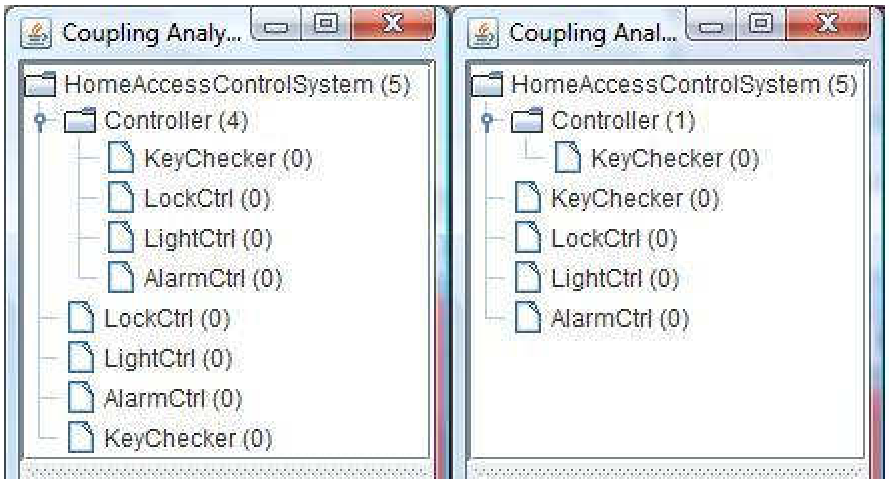


Fig. 13: Class dependency tree for the original and Publisher-Subscriber versions, respectively.

4.2 Cohesion Metrics

We calculate the cohesion metrics using the formulas for SCOM listed in section 2.2.2. According to [13], Abstract and Interface methods must only be considered in the classes they are implemented. Also, constructors and destructors methods and attributes must be included. Keeping these rules in mind, we list the results of these calculations for all the classes in our system in Table 4.

Class Name	*Original Version*	*Publisher-Subscriber*	*More Cohesive*
HomeAccessControlSystem	0.43	0.43	Equal
Controller	1	1	Equal
KeyChecker	0.5	0.38	Original
AlarmCtrl	0.667	0.238	Original
LockCtrl	0.462	0.325	Original
LightCtrl	0.667	0.296	Original

Table 4: Cohesion Metrics

Based on the Cohesion Metric numbers listed in table 4, we can conclude that for the most part, since the original design was already highly cohesive, there wasn't much scope to increase it further.

The reduced cohesion in Publisher-Subscriber version of '*KeyChecker*' is a result of including the implemented publisher interface methods. Since methods of this type in general only have attributes that do not interact at all or not much with the other local methods and the methods themselves use only very few of the other attributes in the class, it is understandable that the cohesion was slightly lower than the original. '*AlarmCtrl*', '*LockCtrl*', '*LightCtrl*' are also affected in a similar way.

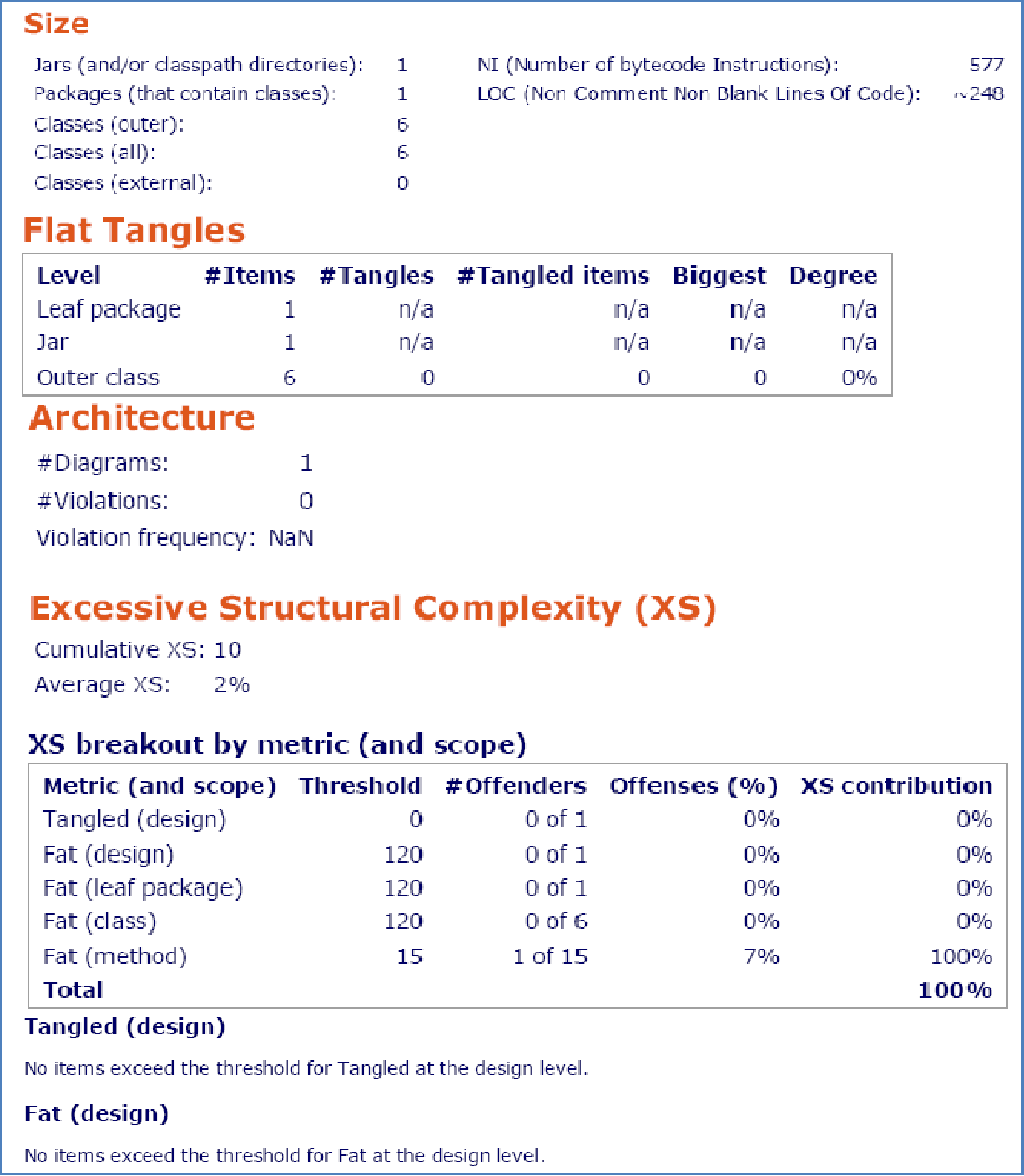
However, this scenario is not always true for Pub-Sub methods. In our case, we do not need to pass any required parameters in the publisher, subscriber interface methods which need to be used by other methods in the class which implements these interfaces. If a required parameter were to be passed from the publisher to the subscriber, which was used by other methods in the class, the cohesion could possibly improve when using the SCOM metric because that would create a stronger bond in the class, with more methods using the same attributes.

The reason this interface method, *KeyIsInvalid*, does not affect the *Controller* is because it does not use any attributes at all when implemented in the *controller*. SCOM requires that such methods and attributes not be counted in Cohesion measurements. In '*KeyChecker*', '*AlarmCtrl*', '*LockCtrl*' and '*LightCtrl*', the publisher-subscriber interface methods use one or two of the other attributes, but when implemented in the controller, the interface method only calls another method and does not use any attributes at all.

4.3 Structural Complexity Measurement Using Structure101

We also ran our code through the Structure101 framework to measure the structural complexity of our code. The results are shown below. Figures 14 and 15 show the dependency graphs generated by Structure101.

Structure101 Results for Original Code (No Design Patterns)



Size

Jars (and/or classpath directories):	1	NI (Number of bytecode Instructions):	577
Packages (that contain classes):	1	LOC (Non Comment Non Blank Lines Of Code):	~248
Classes (outer):	6		
Classes (all):	6		
Classes (external):	0		

Flat Tangles

Level	#Items	#Tangles	#Tangled items	Biggest	Degree
Leaf package	1	n/a	n/a	n/a	n/a
Jar	1	n/a	n/a	n/a	n/a
Outer class	6	0	0	0	0%

Architecture

#Diagrams:	1
#Violations:	0
Violation frequency:	NaN

Excessive Structural Complexity (XS)

Cumulative XS:	10
Average XS:	2%

XS breakout by metric (and scope)

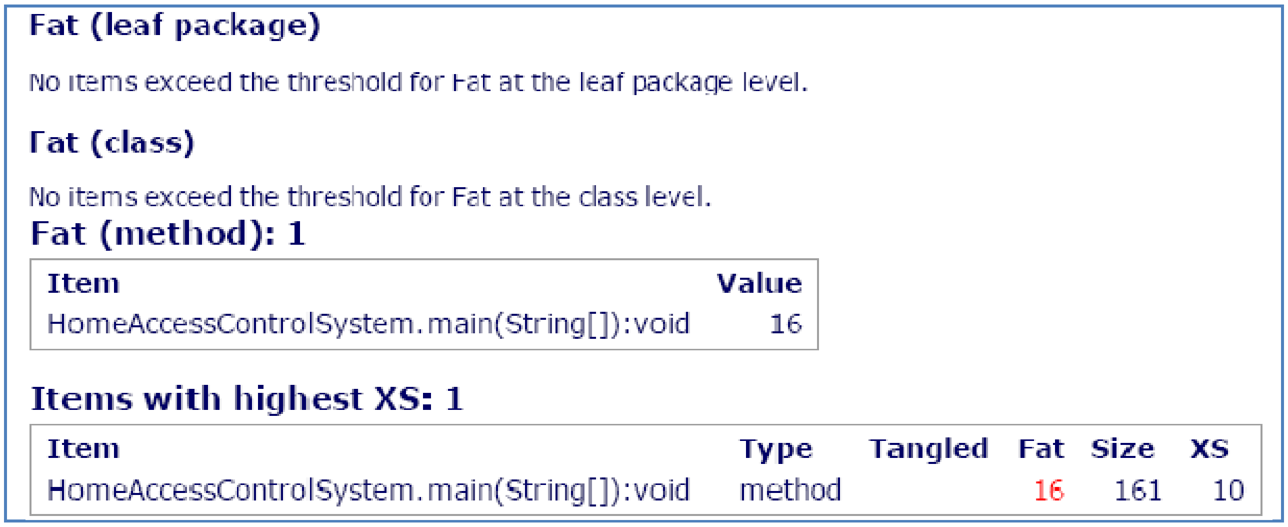
Metric (and scope)	Threshold	#Offenders	Offenses (%)	XS contribution
Tangled (design)	0	0 of 1	0%	0%
Fat (design)	120	0 of 1	0%	0%
Fat (leaf package)	120	0 of 1	0%	0%
Fat (class)	120	0 of 6	0%	0%
Fat (method)	15	1 of 15	7%	100%
Total				**100%**

Tangled (design)

No items exceed the threshold for Tangled at the design level.

Fat (design)

No items exceed the threshold for Fat at the design level.



Fat (leaf package)

No items exceed the threshold for Fat at the leaf package level.

Fat (class)

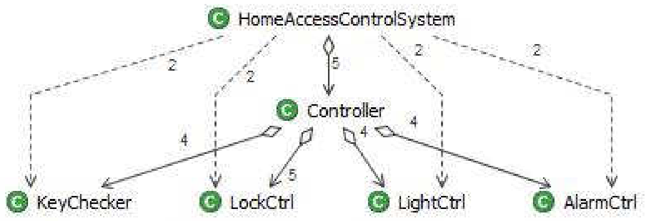
No items exceed the threshold for Fat at the class level.

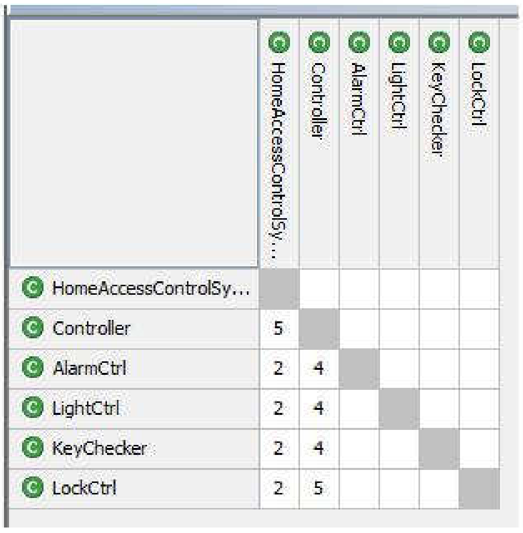
Fat (method): 1

Item	Value
HomeAccessControlSystem.main(String[]):void	16

Items with highest XS: 1

Item	Type	Tangled	Fat	Size	XS
HomeAccessControlSystem.main(String[]):void	method		16	161	10



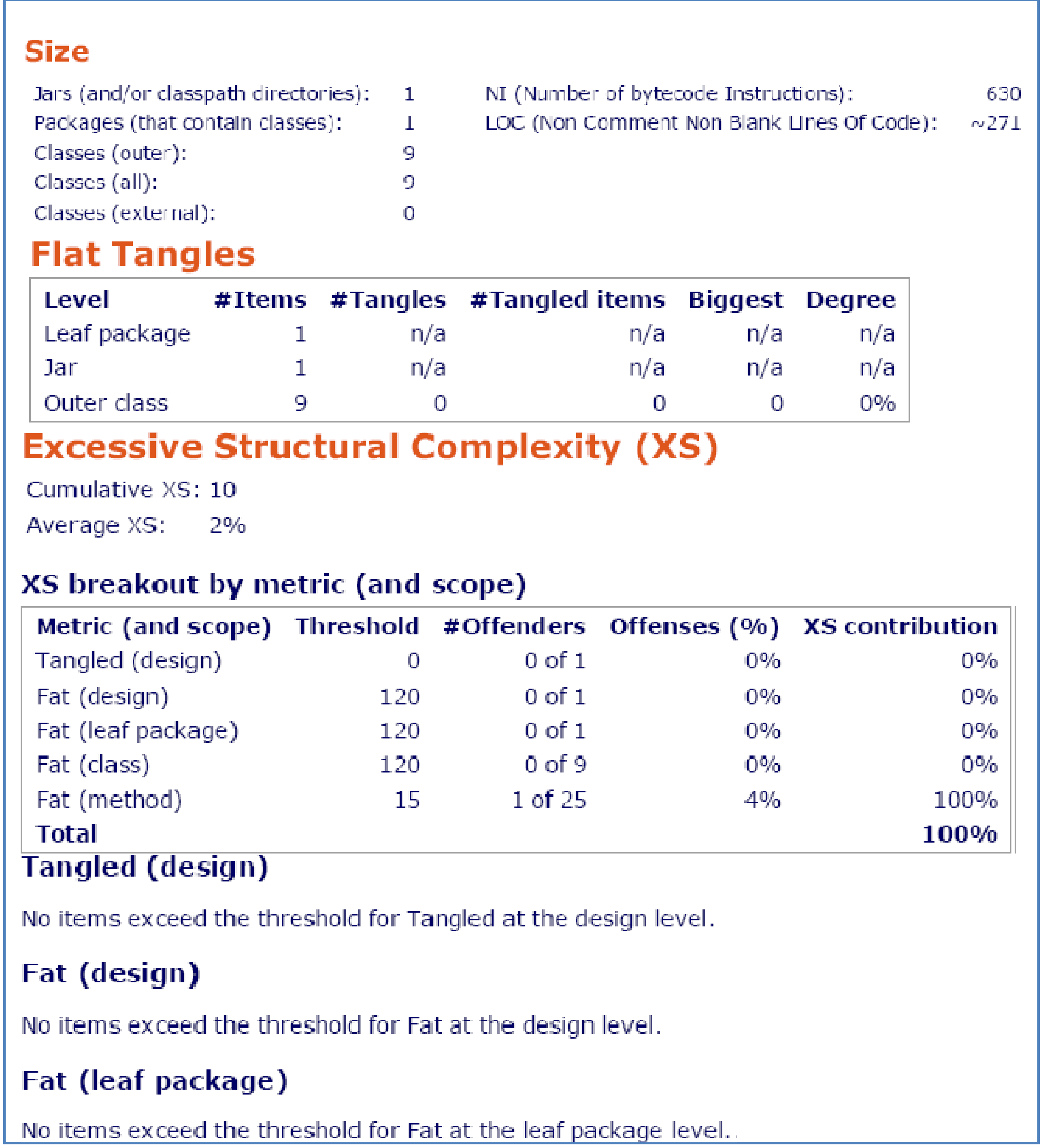


	HomeAccessControlSy…	Controller	AlarmCtrl	LightCtrl	KeyChecker	LockCtrl
HomeAccessControlSy…						
Controller	5					
AlarmCtrl	2	4				
LightCtrl	2	4				
KeyChecker	2	4				
LockCtrl	2	5				

Fig. 14: Dependency graph for code without Design Patterns.

According to the Structure101 framework results provided above for the original code, there are no *Tangles* or *Fat* except for the *main()* method in the *HomeAccessControlSystem*. This class only exists to setup the USB virtual COMM port connection with the Relay Switch Board. The code for this also does not change in the version implemented with the Publisher-Subscriber Design Pattern. The results for that code provided below are the same. The *main()* method in the *HomeAccessControlSystem* is the only one with fat.

Structure101 Results for Publisher-Subscriber Pattern



Size

Jars (and/or classpath directories):	1	NI (Number of bytecode Instructions):	630
Packages (that contain classes):	1	LOC (Non Comment Non Blank Lines Of Code):	~271
Classes (outer):	9		
Classes (all):	9		
Classes (external):	0		

Flat Tangles

Level	#Items	#Tangles	#Tangled items	Biggest	Degree
Leaf package	1	n/a	n/a	n/a	n/a
Jar	1	n/a	n/a	n/a	n/a
Outer class	9	0	0	0	0%

Excessive Structural Complexity (XS)

Cumulative XS: 10

Average XS: 2%

XS breakout by metric (and scope)

Metric (and scope)	Threshold	#Offenders	Offenses (%)	XS contribution
Tangled (design)	0	0 of 1	0%	0%
Fat (design)	120	0 of 1	0%	0%
Fat (leaf package)	120	0 of 1	0%	0%
Fat (class)	120	0 of 9	0%	0%
Fat (method)	15	1 of 25	4%	100%
Total				**100%**

Tangled (design)

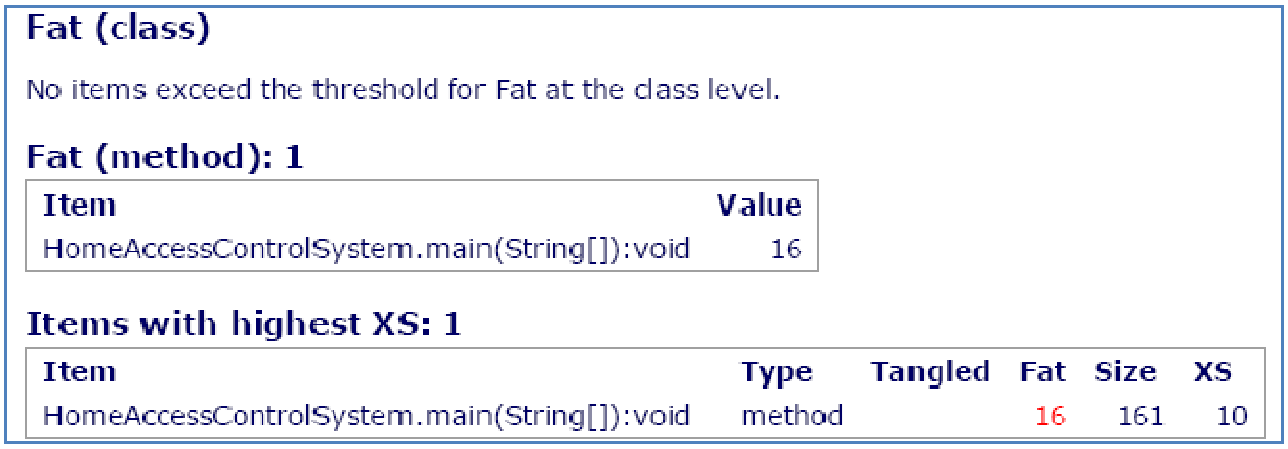
No items exceed the threshold for Tangled at the design level.

Fat (design)

No items exceed the threshold for Fat at the design level.

Fat (leaf package)

No items exceed the threshold for Fat at the leaf package level.



Fat (class)

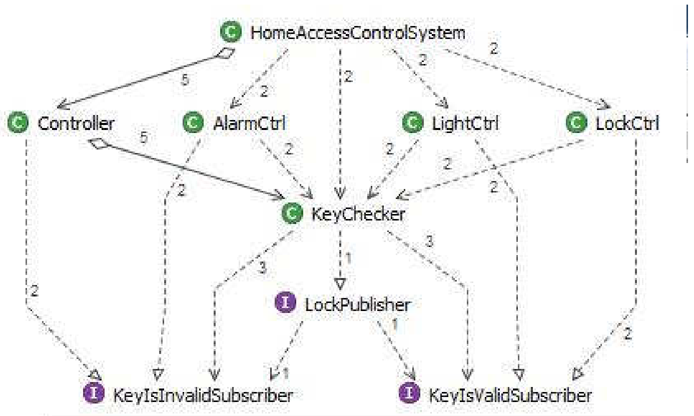
No items exceed the threshold for Fat at the class level.

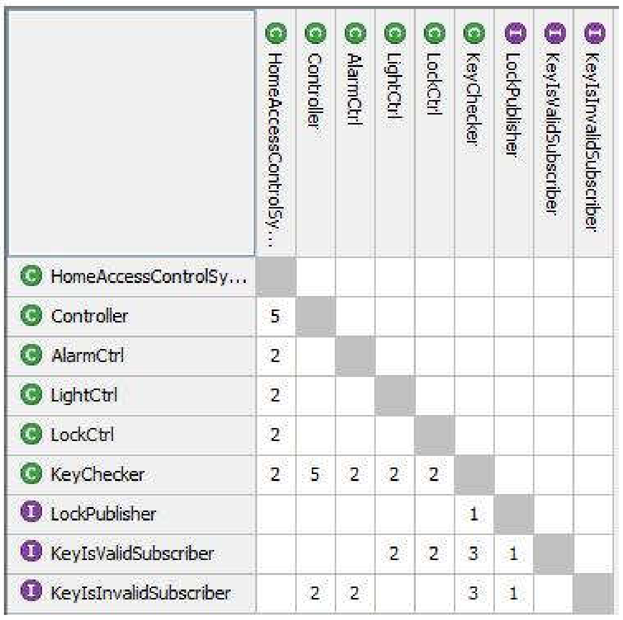
Fat (method): 1

Item	Value
HomeAccessControlSystem.main(String[]):void	16

Items with highest XS: 1

Item	Type	Tangled	Fat	Size	XS
HomeAccessControlSystem.main(String[]):void	method		16	161	10





	HomeAccessControlSy...	Controller	AlarmCtrl	LightCtrl	LockCtrl	KeyChecker	LockPublisher	KeyIsValidSubscriber	KeyIsInvalidSubscriber
HomeAccessControlSy...									
Controller	5								
AlarmCtrl	2								
LightCtrl	2								
LockCtrl	2								
KeyChecker	2	5	2	2	2				
LockPublisher						1			
KeyIsValidSubscriber				2	2	3	1		
KeyIsInvalidSubscriber		2	2			3	1		

Fig. 15: Dependency graph for code with Publisher-Subscriber Pattern.

Based on the dependency graphs and the Tangles and Fat metrics calculated by Structure101, we can conclude that introducing the Publisher-Subscriber design pattern does not introduce any unnecessary complexity into the code. The Tangles and Fat metrics are exactly the same for both versions. However, since this was a fairly small application compared to most real world applications, we cannot be entirely sure how structure 101 would rate a much bigger application. Looking at the current results though, since the Publisher-Subscriber pattern in itself did not result in any increased complexity, we could assume that the results will be similar regardless of the complexity of the original code. Introducing the Publisher-Subscriber pattern into the code will not be the reason for increased complexity.

Even though it is not evident in our case, since the original code did not have structural complexity, introducing the Publisher-Subscriber interface into an application could possibly reduce structural complexity if it was originally present. According to [18], for lower level packages, which contain classes, fat is computed by counting the total number of dependencies between classes. In a bigger real world application, introducing the publisher-subscriber interface would have more of an impact in improving cohesion and reducing fat according to the Structure101 framework by reducing dependencies between classes.

5. Conclusion & Future Work

From the results obtained, we can conclude that using the Publisher-Subscriber design pattern does result in low coupling and ease of scalability, making it a desirable choice. However, the cohesion metric results are not very encouraging since the classes implementing the publisher and the subscriber interfaces have to account for the interface methods. This brings the cohesion down, since they rarely interact with other methods or attributes in the class. The results from Structure101 however are more encouraging. There were no Tangles at all and the only fat present in both versions was in the *HomeAccessControlSystem* "*main()*" method. This shows us that the code is not unnecessarily complex even with the introduction of the Design Pattern.

In conclusion, we feel that the overhead and trade-offs created by implementing a Publisher-Subscriber interface are outweighed by the benefits it provides. According to the Structure101 results, there are no major trade-offs in terms of code complexity and the coupling and dependency go down. In future, more patterns could be tested to make a more thorough analysis and to see if this result is universal for all design patterns.

6. References

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object - Oriented Software.* Addison Wesley, 1995.

[2] I. Marsic, *Software Engineering*, Rutgers University, Electrical and Computer Engineering Department Course Textbook, 2008.

[3] Wikipedia, *http://en.wikipedia.org/*

[4] P. Joshi, R. K. Joshi, “Quality Analysis of Object Oriented Cohesion Metrics,” *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pp. 319 - 324, 2010.

[5] S. R. Chidamber and C. F. Kemerer, “Towards a metrics suite for object oriented design,” *OOPLSA*, pp. 197–211, 1991.

[6] W. Li and S. Henry, “Maintenance metrics for object oriented paradigm,” in *Proceedings of International software metrics symposium*, May 1993, pp. 52–60.

[7] M. Hitz and B. Montazeri, “Chidamber and kemerer’s metric suite: A measurement theory perspective,” *IEEE transactions on Software Engineering*, vol. 22, no. 4, pp. 267–271, April 1996.

[8] S. R. Chidamber and C. F. Kemerer, “A metric suite for object oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, pp. 476–493, Jun 1994.

[9] L. Xlinke, L. Zongtian, P. Biao, and X. Dahong, “A measurement tool for object oriented software and measurement experiments with it,” in *Proceedings of 10th International Workshop on New Approaches in Software Measurement*, 2000, pp. 44–54.

[10] N. Balasubramanian, “Object-oriented metrics,” in *International Proceedings of Asia-Pacific Conference on Software Engineering*, 1996, pp. 30–34.

[11] J. M. Bieman and B.-K. Kang, “Cohesion and reuse in an object-oriented system,” in *Proceedings of ACM symposium for software reusability*, 1995, pp. 259–262.

[12] B.Henderson-Sellers, *Object-Oriented Metrics Measures of Complexity*. Prentice Hall PTR, 1996.

[13] L. Fernández and R. Pena, “A sensitive metric of class cohesion,” *International Journal Information Theories and Applications*, vol. 13, pp. 82 – 91, 2006.

[14] H. S. Chae, Y. R. Kwon, and D. H. Bae, “A cohesion measure for object-oriented classes,” S*oftware- Practice and Experience*, vol. 30, pp. 1405–1431, 2000.

[15] Y. Zhou, B. Xu, J. Zhao, and H. Yang, “Icbmc: An improved cohesion measure for classes,” in *Proceedings of 21st IEEE Conference on Software Maintenance (ICSM)*, Montreal, Canada, October 2002, pp. 44–53.

[16] J. Bansiya, L. Etzkorn, C. Davis, and W. Li, “A class cohesion metric for object oriented designs,” *Journal of Object Oriented Programming*, vol. 11, no. 8, pp. 47–52, Jan 1999.

[17] J. Y. Chen and J. F. Lu, “A new metric for object-oriented design,” *Journal of Information and Software technology*, vol. 35, pp. 232–240, April 1993.

[18] R. S. Sangwan, P. Vercellone-Smith, P. A. Laplante, “Structural Epochs in the Complexity of Software over Time,” *IEEE software*, vol. 25(4), pp. 66-73, July-August 2008.

[19] P. Sreekumar and I. Marsic, “A Study of Structure of Java Programs Using Coupling Analyzer,” Rutgers University, Electrical and Computer Engineering Department Technical Paper, 2009.

[20] National Control Devices, *http://www.controlanything.com/manuals/ProXR.pdf*