APPLIED SOFTWARE ENGINEERING

Hamza Bhatti (21223241)

Contents

Introduction

For this assignment a case study was provided. Appropriate UML, implementation in Java, along with testing was produced based on one chosen use case. The report will also cover critical analysis of all the previously mentioned elements of the assignment.

Use Cases Found

*Appendix_ Saxon Use Case Diagram

Before one use case was chosen to be implemented, a variety of use cases were found within the case study. Many different actors where found, ranging from different staff, members of Saxon Heritage and even the system itself.

The use cases found were presented in an abstract format, but still had meaning when associated with the actors themselves. If documentation is kept for each of the use cases in the future, along with more detailed use case descriptions, there would be room for maintainability for new developers of the system. This would be the case for all the use cases presented.

Class Diagram

*Appendix_ Saxon Class Diagram

After the use cases where found, a class diagram was also created that covered all of the use cases. This was useful as it would aid developers to implement different parts of the system. It also helped in showing the different relationships that classes would have once a whole system was implemented.

The class diagram was very abstract. This was the case as no other classes, attributes or operations were identified other than the useful ones that related to the use cases in the diagram.

Naturally, most of the operations were present in the SaxonSystem class. This was because other classes acted as the domain model. The SaxonSystem class would be the one that would handle all of the business logic. Classes that are behind the system boundary would be safe from direct access.

Architecture Diagram

*Appendix_ Architecture Diagram

An architecture diagram was also created for the system as a whole. The architecture that was used was a generic layered architecture. Though this architecture was not adopted for the use case that was chosen to be implemented, it can still be used when the whole system is developed in the future.

The system proposed is split into multiple layers. The top most layer is where actors would be able to interact with the system through a user interface. The next layer is where authentication would take place. This would involve checking if the user can access certain areas of the system. It would also allow for error handling of entered information. The layer below this deals with the business logic. This is where all the use cases are situated. This is because the front end that the actor can see would be where all the business processes would occur, keeping other components away from direct access. The lowest layer is the data bases layer where data is stored and can be retrieved. This would keep the data away from direct use. As a database would be used, crucial data would still be saved in the event of the system crashing.

As mentioned this architecture was not fully implemented, but can be implemented when the whole system is put together. For the use case that will be shown in the next section, the implementation uses ArrayLists instead of databases as a means of storing data.

Requirements Engineering

The use case that was chosen to be implemented was “Update site popularity” which also included “Prioritise site for marketing”. A set of requirements were found within the case study which had to be followed to ensure that the implementation would be successful.

Requirement for the Use Case

The requirements understood where:

1) The check will occur on the 30th December every year.
2) 6 regions are currently present, all of which have sites.
3) Site have ratings which are Bronze, Silver and Gold.
4) Site visitors will determine the new rating. If visitors are below 10,000 a Bronze is given, if between 10,000 and 30,000 a Silver is given and when above 30,000 a Gold is given.
5) While the rating is given, the site will be checked if it needs a marketing campaign.

To ensure that the implementation would be successful, these requirements had to be reflected in the use case description, class diagram and sequence diagrams. This would ensure that all documentation would be consistent with one another.

For maintainability of the system in the future, new developers that may have to handle the system will have these requirements as a guide. They would have an idea of what the system should include and have basic knowledge of how the system would work.

UML Modelling

To aid in implementing the requirements of the use case, a series of UML diagrams where created. The diagrams provide a description of what should occur, the components that should be present, their interactions and sequence of events that take place to meet the requirements.

Use Case Description

* Appendix _ Use case diagram

Talk about the functionality of the use case diagram – What’s going to happen in the software

The use cases that are show in the diagram were, “Update site popularity” which included the use case “Prioritise site for marketing”. The actor for these use cases was the System itself as there was no outside interaction by a physical user.

The use case description produced described how the system that would implemented will function. It covered the goal, pre-conditions, post-condition and how the use cases would be triggered along with the steps needed to reach the goal.

The requirements that had been identified in the previous section of the report are embedded in the whole use case diagram. One requirement was that the date needed to be 30th of December for the popularity to be changed. In the use case description, it is described that the date will be checked against a target. If the target was reached, the use case could carry on, otherwise nothing would happen.

Another requirement was for the popularity ratings that would be determined by the amount of visitors of a site. The description states that the rating will change to either Bronze, Silver or Gold depending on if the visitor count meets a certain threshold.

Finally, the check to see if marketing was needed for a site was identified in the description. This would occur before the "Update site popularity" use case has been completed as it was included alongside the use case. The process itself checks if the site needed marketing based on whether the site had half of the Bronze rating threshold of visiors.

Class Diagram

* Appendix _ Class diagram

What are the elements in the class diagram – classes (attributes, operations), associations, pattern and abstraction?

To ensure that the use case and requirements could be met in the implementation, an appropriate class diagram had to be produced. The class diagram provided, contains the different classes, their attributes, operations and their associations with one another. A design pattern is also provided here, which will be further described later in the implementation section.

The classes created were based on the requirements. The SaxonSystem class was made to handle the business logic of the use cases "Update site popularity" and "Prioritise site for marketing". The Region class was created to hold a number of sites and this adhering to the use case precondition.

The Site class was created with operations that would help retrieve the amount of visitors and set a new rating. Again, these operations would be accessed through the SaxonSystem and not directly used anywhere else in the program.

All classes presented have interfaces which act as contracts and so made the diagram abstract. This is the case because classes that implement the interfaces only show what attributes are needed for the implementation. The contract themselves have no bearing on the implementation, but show the method headings that need to be used to achieve the use cases. Methods identified in the contracts are named in a way that will help future developers understand what should be going on within them. This aids in maintainability of the system when handled by new teams.

The abstraction of the diagram helps remove clutter, such as extra classes that do not need to be there. It also only shows the attributes and operations that are needed to fulfil the goal of the use case. This removes any confusion and leaves that system open and less rigid for any type of implementation for future development.

Sequence Diagram

* Appendix _ Sequence diagram

The sequence diagram shows the process that would occur in the implementation by following the requirements, description and the components from the class diagram. The classes that are included in the sequence diagram are the SaxonSystem which also is the actor, the Site class were information will be retrieved and changed, and the Region classes that will have changes from the Site reflected on it.

Behaviour of the system – Talk about what happens at each step.

The first operation that is used in the sequence diagram is the updateSitePopularity() which is in line with the use case “Update Site Popularity”. This sends a message from the actor to itself. From here the date is checked to see if it is 30th of December. This is in line with the requirements and the use case description. If the date has not been reached, a message will be sent back to the system saying the date has not been reached. Otherwise the operation would carry on working.

Next, the SaxonSystem will interact with the Site class as it has an association with it, found in the class diagram. Though the System has access to the Regions which also contains Sites, it is much quicker to interact with the Sites directly. It would take more steps to go through the Region, retrieve the Sites, and then retrieve the information needed. The Site method getVisitors() was used to retrieve the siteVisitors. This value is needed to determine what the new sitePopularity value would be set to.

The Site operation setSitePopularity() would then be used to set a new sitePopularity rating. As mentioned in the requirements, this depends on the amount of visitors that the Site gets. Alternate flows are used here, showing what the new ratings would be, based off the siteVisitors. The Site rating would be set accordingly. After the new sitePopularity is set, this will then reflect onto the Region objects that each have a collection of Sites. The changes that are directly made to the Site, will reflect onto the Regions also.

Following the use case diagram, the included use case was “Prioritise site for marketing” which occurs before the “Update Site Popularity” use case and method have been completed. In the sequence diagram, the method prioritiseSiteForMarketing() is called. Another alternate flow is identified which falls in line with the use case description. If the Site had a threshold of visitors that was half of the Bronze rating, it would need marketing, otherwise nothing will occur.

Finally a confirmation is given. By the end of the sequence of events, the Site would have been updated and checked to see if it needs marketing. The latter would occur before the update had finished.

Implementation

* Appendix _ Source Code

The implementation in Java was created with all previous sections in mind. The source code follows the requirements and use case description. It also takes all the classes identified in the class diagram, along with methods and follows the sequence of events identified in the sequence diagram.

Using Interfaces

Many of the classes implement interfaces. These interfaces act as contracts, identifying what methods need to be present in the classes that implement them. The abstract nature of the interface allows future developers to decide how they would want to implement certain methods in the future if requirements change. Below are the interfaces and an indication of what classes implemented them. These are also reflected in the class diagram.

Region implemented by classes of type Region (6 Regions from the case study)

```java
public interface Region {
    // All classes implementing this interface should have addSite
    public void addSite(Site siteToAdd);
    // Only needed for testing purposes
    public ArrayList<Site> getSites();
}
```

SiteInterface implemented by Site

```java
public interface SiteInterface {
    // Methods to be implemented
    public int getSiteVisitors();
    public void setSitePopularity(String newSitePopularity);
}
```

SaxonSystemInterface implemented by SaxonSystem

```java
public interface SaxonSystemInterface {
    // Methods that must be implemented
    public String updateSitePopularity();
    public void prioritiseSiteForMarketing(Site siteToPrioritise);
}
```

The methods used in the interfaces are all consistent with the class diagram. The methods names also help understand what should be going on within them to avoid confusion. Along with this, the SaxonSystemInterface methods share the names of the two use cases, clearly identifying what should occur within the method implementation.

Using interfaces also ensures reusability of code. This is because any class that may be added to the system, for example more Sites and Regions, they can use the same methods from the interface and developers would not need to start from scratch.

Factory Design Pattern

From the requirements, it was mentioned that there were 6 regions. From the class diagram there was use of a factory pattern. The pattern helped create the 6 regions from the case study and where of type Region. All of the regions created had regionNames and sites. Below is a code snippet from the RegionFactory class that returns region objects.

```
public Region makeRegion(String regionName) {
    if (regionName.equalsIgnoreCase("LONDON")) {
        return new London();
    }
```

The benefit of using this factory pattern is that loose coupling can be achieved. The code that is created around the objects only interact with the interface. The interface itself means that any class in a system can use it. The objects created will only depend on the interface (JavaTPoint 2014).

Fulfilling the Use Case

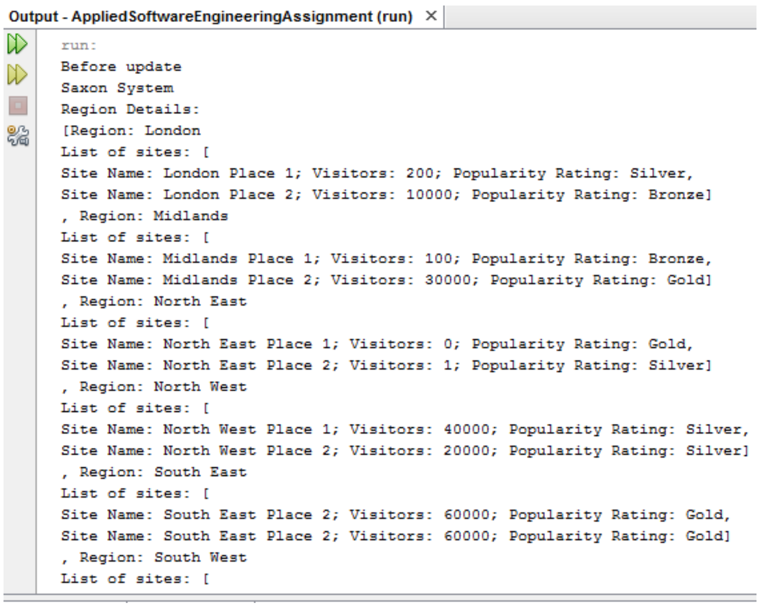
The requirements of the use cases can be seen as a whole in SaxonSystem class. The updateSitePopularity() method is used to check the date, retrieve the Site visitor details, then update the site popularity depending on the details. As that occurs, the Site is checked to see if it needs marketing when the prioritiseSiteForMarketing() method is used. The prioritising adhered to the use case diagram as it occurs before the "Update site popularity" method finishes and uses a threshold to determine if the site needs marketing. Below are the methods showing how the use case was fulfilled.

```java
// USE CASE - Update site popularity
// Checks date, loops through Sites, gets the siteVisitors,
// sets a new sitePopularity based on visitors,
// Checks if marketing needed, finishes
public String updateSitePopularity() {
    if (currentDate.equals(date)) {
        Site s;

        for (int i = 0; i < sites.size(); i++) {
            s = sites.get(i);

            int visitors = s.getSiteVisitors();

            if (visitors < 10000) {
                s.setSitePopularity("Bronze");
            } else if (visitors >= 10000 && visitors < 30000) {
                s.setSitePopularity("Silver");
            } else {
                s.setSitePopularity("Gold");
            }

            prioritiseSiteForMarketing(s);
        }
        return ("All site popularity ratings have been updated"
                + "\n" + regions);

    } else {
        return ("Cant Update Popularity yet. Wait till 30th Dec");
    }
}

// USE CASE - Prioritise site for marketing
// Site as argument, check siteVisitors, add to martketting
public void prioritiseSiteForMarketing(Site siteToPrioritise){
    if(siteToPrioritise.getSiteVisitors() < 5000) {
        // Add it to a list
        prioritsedSites.add(siteToPrioritise);
    }
}
```

Proof of Execution

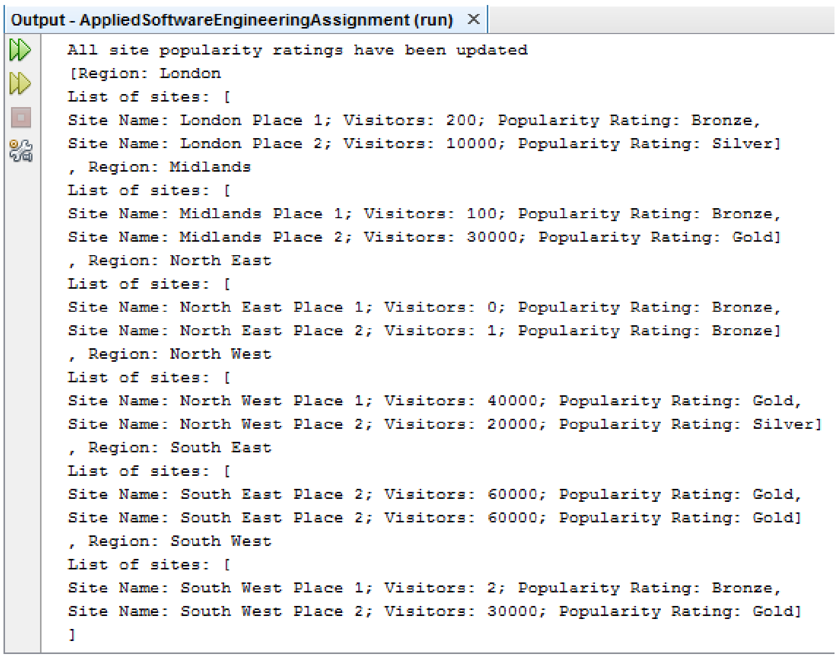
First, all the objects where created. This involved creating Regions with the factory, adding a series of Site to them. Then the updateSitePopularity() method was called.

The screenshot below shows a few Sites that belong to Regions. This information was presented before the updateSitePopularity() method was called.



```
run:
Before update
Saxon System
Region Details:
[Region: London
List of sites: [
Site Name: London Place 1; Visitors: 200; Popularity Rating: Silver,
Site Name: London Place 2; Visitors: 10000; Popularity Rating: Bronze]
, Region: Midlands
List of sites: [
Site Name: Midlands Place 1; Visitors: 100; Popularity Rating: Bronze,
Site Name: Midlands Place 2; Visitors: 30000; Popularity Rating: Gold]
, Region: North East
List of sites: [
Site Name: North East Place 1; Visitors: 0; Popularity Rating: Gold,
Site Name: North East Place 2; Visitors: 1; Popularity Rating: Silver]
, Region: North West
List of sites: [
Site Name: North West Place 1; Visitors: 40000; Popularity Rating: Silver,
Site Name: North West Place 2; Visitors: 20000; Popularity Rating: Silver]
, Region: South East
List of sites: [
Site Name: South East Place 2; Visitors: 60000; Popularity Rating: Gold,
Site Name: South East Place 2; Visitors: 60000; Popularity Rating: Gold]
, Region: South West
List of sites: [
```

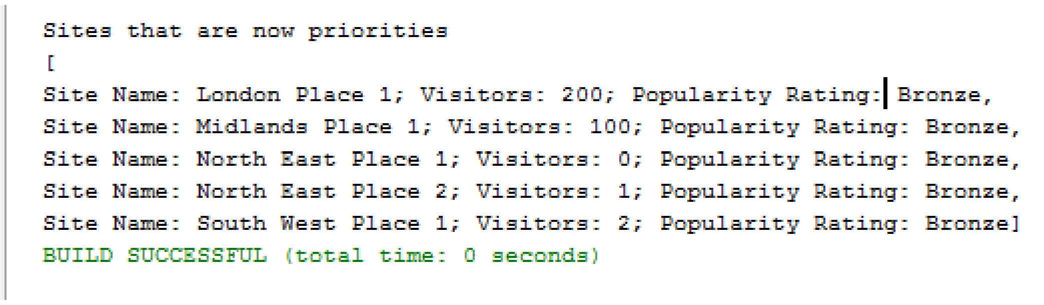
Looking at the above image, London Place 1 for example, has a rating of Silver, but after the update will have a Bronze rating as the amount of visitors is below 10000. The image below shows the sites after the update.



```
All site popularity ratings have been updated
[Region: London
List of sites: [
Site Name: London Place 1; Visitors: 200; Popularity Rating: Bronze,
Site Name: London Place 2; Visitors: 10000; Popularity Rating: Silver]
, Region: Midlands
List of sites: [
Site Name: Midlands Place 1; Visitors: 100; Popularity Rating: Bronze,
Site Name: Midlands Place 2; Visitors: 30000; Popularity Rating: Gold]
, Region: North East
List of sites: [
Site Name: North East Place 1; Visitors: 0; Popularity Rating: Bronze,
Site Name: North East Place 2; Visitors: 1; Popularity Rating: Bronze]
, Region: North West
List of sites: [
Site Name: North West Place 1; Visitors: 40000; Popularity Rating: Gold,
Site Name: North West Place 2; Visitors: 20000; Popularity Rating: Silver]
, Region: South East
List of sites: [
Site Name: South East Place 2; Visitors: 60000; Popularity Rating: Gold,
Site Name: South East Place 2; Visitors: 60000; Popularity Rating: Gold]
, Region: South West
List of sites: [
Site Name: South West Place 1; Visitors: 2; Popularity Rating: Bronze,
Site Name: South West Place 2; Visitors: 30000; Popularity Rating: Gold]
]
```

As can be seen, the London Place 1 had updated accordingly. Comparing the images, it is evident that all sites had been updated appropriately.

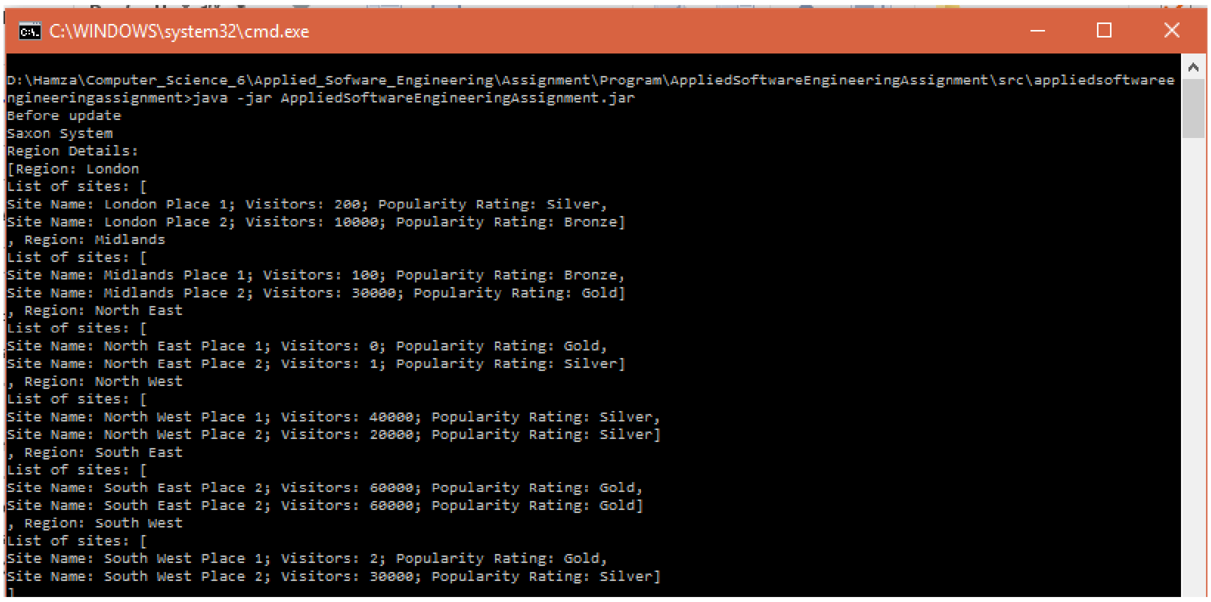
As the prioritiseSiteForMarketing() method occurs from inside the updateSitePopularity() method, it is important to show that this method works accordingly also. As previously mentioned, when the Site visitors are below half of the Bronze threshold (in this case below 5000), it would have been added to a prioritised list. Below is an image of all the sites that had been added. These fall in line with the updated popularity ratings.



```
Sites that are now priorities
[
Site Name: London Place 1; Visitors: 200; Popularity Rating: Bronze,
Site Name: Midlands Place 1; Visitors: 100; Popularity Rating: Bronze,
Site Name: North East Place 1; Visitors: 0; Popularity Rating: Bronze,
Site Name: North East Place 2; Visitors: 1; Popularity Rating: Bronze,
Site Name: South West Place 1; Visitors: 2; Popularity Rating: Bronze]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Running Instruction

Navigate from src >> appliedsoftwareengineeringassignment and you will see many files including a .jar file. In the windows document viewer, hold shift and press right click on the moue. You will see an option to "Open with command prompt", click this. A terminal will open. Type in the command: java –jar AppliedSoftwareEngineeringAssignment.jar and press enter this will run the file.



```
D:\Hamza\Computer_Science_6\Applied_Sofware_Engineering\Assignment\Program\AppliedSoftwareEngineeringAssignment\src\appliedsoftwaree
ngineeringassignment>java -jar AppliedSoftwareEngineeringAssignment.jar
Before update
Saxon System
Region Details:
[Region: London
List of sites: [
Site Name: London Place 1; Visitors: 200; Popularity Rating: Silver,
Site Name: London Place 2; Visitors: 10000; Popularity Rating: Bronze]
, Region: Midlands
List of sites: [
Site Name: Midlands Place 1; Visitors: 100; Popularity Rating: Bronze,
Site Name: Midlands Place 2; Visitors: 30000; Popularity Rating: Gold]
, Region: North East
List of sites: [
Site Name: North East Place 1; Visitors: 0; Popularity Rating: Gold,
Site Name: North East Place 2; Visitors: 1; Popularity Rating: Silver]
, Region: North West
List of sites: [
Site Name: North West Place 1; Visitors: 40000; Popularity Rating: Silver,
Site Name: North West Place 2; Visitors: 20000; Popularity Rating: Silver]
, Region: South East
List of sites: [
Site Name: South East Place 2; Visitors: 60000; Popularity Rating: Gold,
Site Name: South East Place 2; Visitors: 60000; Popularity Rating: Gold]
, Region: South West
List of sites: [
Site Name: South West Place 1; Visitors: 2; Popularity Rating: Gold,
Site Name: South West Place 2; Visitors: 30000; Popularity Rating: Silver]
```

Software Testing

To check if the implementation was working accordingly, based on requirements and general running of the system, JUnit testing was used.

Purpose of Testing

Unit tests where created to ensure that the implementation was working accordingly. The testing environment used was JUnit. JUnit itself is widely used in industry to test java applications and is now a standard in most IDEs (Wide Skills 2015).

Using Junit allows all the methods to be tested. This is the case when modifications to the methods are made (Gorav, Ankush 2012). The testing of these methods can happen any time during the development of software, but earlier is better to ensure methods are working accordingly.

Testing the system before deployment is a necessary step. This would reduce the chances of having a handed over system being turned down because operations do not work well or do not meet requirements. This would cause a huge loss in money and increase development time.

Tests Carried Out

Tests were created to ensure that the requirements were fulfilled, along with the UML diagrams that had been created. The methods that where presented in the class and sequence diagrams where focussed on, as those were the elements that would ensure that the goal of the two uses could be met. Testing these ensured that the methods were successful.

Testing Default Site

One of the tests that was made was testDefaultSiteRating(). As the name suggests, this test was created to see what siteRating value is given to a Site object created with default values. The test can be seen below, along with the instantiation of a default Site.

```
siteTest2 = new Site();

public void testDefaultSiteRating() {
    String siteRating = siteTest2.getSitePopularity();
    assertEquals(siteRating, "Bronze");
}
```

This test was created to ensure that the precondition that new Sites should have a default rating of Bronze. This test passed when checking if the siteRating was the same as Bronze.

Testing if Regions Had Sites

This test was created to ensure another precondition was fulfilled. This was to ensure that Region objects had Sites within them. The test can be seen below.

```
londonTest.addSite(siteTest1);
londonTest.addSite(siteTest2);

public void testCheckForSites() {
    ArrayList<Site> sitesTest = londonTest.getSites();
    int i;
    for (i = 0 ; i < sitesTest.size() ; i++) {
        i = i+1;
    }
    assertTrue(i > 0);
}
```

Sites where added to the London region. The arrayList of sites was gathered and a loop was used to make a count of how many sites were found at each iteration. The test was a success as more than 0 sites where present in the London region.

Testing Getting Site Visitors

From the class diagram and sequence diagram, it is identified that the Site class must have the getSiteVisitors() method to determine how to set a new popularity rating. The test can be seen below.

```
siteTest1 = new Site("Test site 1" , 10000, "Silver");

public void testGetVisitors() {
    int visitors = siteTest1.getSiteVisitors();
    assertEquals(visitors, 10000);
}
```

The test passed as true. This is because the value returned from getSiteVisitors() was 1000, and was equal to the assertEquals value. This showed that the method worked correctly and that the siteVisitors are retrieved correctly.

Testing Setting Site Popularity

The requirement of the use case “Update site popularity” required the site to be changed accordingly. To ensure that this worked correctly, the method setSitePopularity() was tested. This method is found in the class diagram and sequence diagram. The test is seen below.

```
public void testSetPopularity() {
    Site st = new Site("Temp site", 2, "Gold");
    st.setSitePopularity("Bronze");
    assertTrue(st.getSitePopularity() == "Bronze");
}
```

The test involved creating a new site with parameters, giving the site details. The setSitePopularity() method was used to change the popularity rating from Gold to Bronze. The assertTrue method was used to check if the new rating was equal to Bronze. The test passed, showing that the method worked accordingly and another requirement was fulfilled.

Testing Update Site Popularity

The method updateSitePopularity was tested. This was to ensure that all the requirements of the use case “Update site popularity” would work. In the SaxonSystem, where the method comes from, the date was set to 30th of December and would be compared with another date attribute. The test is shown below.

```
siteTest1 = new Site("Test site 1" , 10000, "Silver");
siteTest2 = new Site();
siteTest3 = new Site("Test site 2" , 3, "Gold");

public void testUpdateSitePopularity() {
    // In case of siteTest1, end up as Silver
    // In case of siteTest2, stays as Bronze
    // In case of siteTest3, end up as Bronze

    String update = saxonTest.updateSitePopularity();
    //assertEquals(siteTest1.getSitePopularity(), "Silver");
    //asserEquals(siteTest2.getSitePopularity(), "Bronze");
    assertEquals(siteTest3.getSitePopularity(), "Bronze");
}
```

After the updateSitePopularity() method is used to change the sitePopularity rating, checks are made for each of the sites that were instantiated. In each case, the tests pass because they are compared with the value that they should have been set to. This shows that the method worked correctly. It also shows the getSiteVisitors() and setSitePopularity() methods work in the same method together.

Testing Prioritise Site for Marketing

The included use case “Prioritise site for marketing” was also tested. The method prioritiseSiteForMarketing() was used in different cases. The test is shown below.

```
public void testPrioritiseSiteForMarketing() {
    // Site 1 should fail
    saxonTest.prioritiseSiteForMarketing(siteTest1);
//  // Site2 should pass
//  saxonTest.prioritiseSiteForMarketing(siteTest2);
//  // Site3 should pass
    saxonTest.prioritiseSiteForMarketing(siteTest3);

    ArrayList<Site> prioritised = saxonTest.getPrioritisedSites();

    int j;
    for(j = 0 ; j < prioritised.size() ; j++) {
        j = j + 1;
    }
    assertEquals(j, 0);
}
```

When testing the method, the list of prioritised sites was retrieved. This is because the method itself adds the appropriate Site an array when the siteVisitors were less than 5000. In the case above, siteTest1 had 10000 visitors. With the logic applied to the method, it would not be prioritised and put into an arrayList. When the array was looped through, a counter was set to check how many objects where present. As the visitors of the siteTest1 was above

5000, the test passed. This was because the site was not added. When testing siteTest2 and siteTest3, the test failed as these were added to the arrayList and the counter reached 1.

Give instructions on how to run

Evaluation

In this section, an evaluation of all the UML, implementation and testing will be presented.

UML

The UML itself was very abstract. In the use case description for the “Update site popularity” and “Prioritise site for marketing”, specific details were not given. This would allow room for requirement changes. Future developers would only know how the system should work. Although that is the case, specific detail was given in the sequence diagram for methods. If there will be requirement changes, these details would have to be changed.

The class diagram promoted abstraction. No unnecessary classes, attributes or operations where present. This was the fact for both versions of the class diagram. All relationships where given. This would help maintainability for future development, as relationships identified here would help determine how to change certain features, like adding new classes or refactoring classes.

The sequence diagram was in line with the use case description and class diagrams. This was the case as the sequence followed the description and used the same classes and methods from the class diagram.

The class diagram for the whole case study could have included design patterns. The approach used for the case study class diagram was to be more abstract than the one used for the chosen use cases. Including patterns here could have given the system more depth.

Implementation

The code that was created, followed all of the requirements and the UML produced. This ensured that the system was consistent with these. It would also mean that when the system is maintained, there would be no confusion about certain components of the program not falling in line with requirements or the UML.

The code is reusable. This is because of the interfaces that where provided. These interfaces could be used anywhere in the system. As they act as contracts, these interfaces would not need to be changed. Only the implementation would have to change.

Software Testing

The tests provided where beneficial to the project as a whole. Carrying out the tests ensured that the methods presented in the class diagram worked accordingly with the processes in the use case description. It also ensured that the logic presented in the use case description worked affectively. If tests had failed because of lack of understanding of the case study, the UML and implementation could have been reworked to ensure for a better system.

Conclusion

Bibliography

Gorav, A. (2012) *Benefits of Using JUnit.* Available at: https://www.gontu.org/benefits-of-using-junit-framework/ (Accessed: 30/11/2016).

JavaTPoint (2014) *Factory Method Pattern.* Available at: http://www.javatpoint.com/factory-method-design-pattern (Accessed: 28/11/2016).

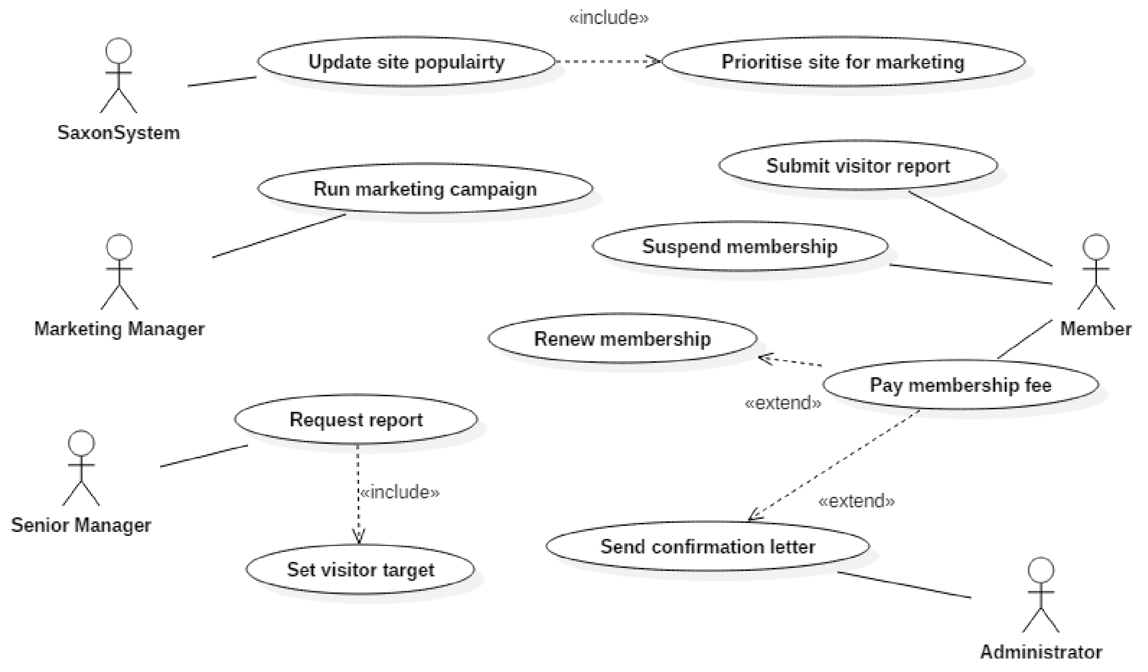
Nevreker, A. (2009) *Exploring Factory Pattern.* Available at: https://www.codeproject.com/articles/37547/exploring-factory-pattern (Accessed: 28/11/2016).

Sommerville, I. (2011) *Software Engineering.* 9th ed. edn. United States of America: Pearson.

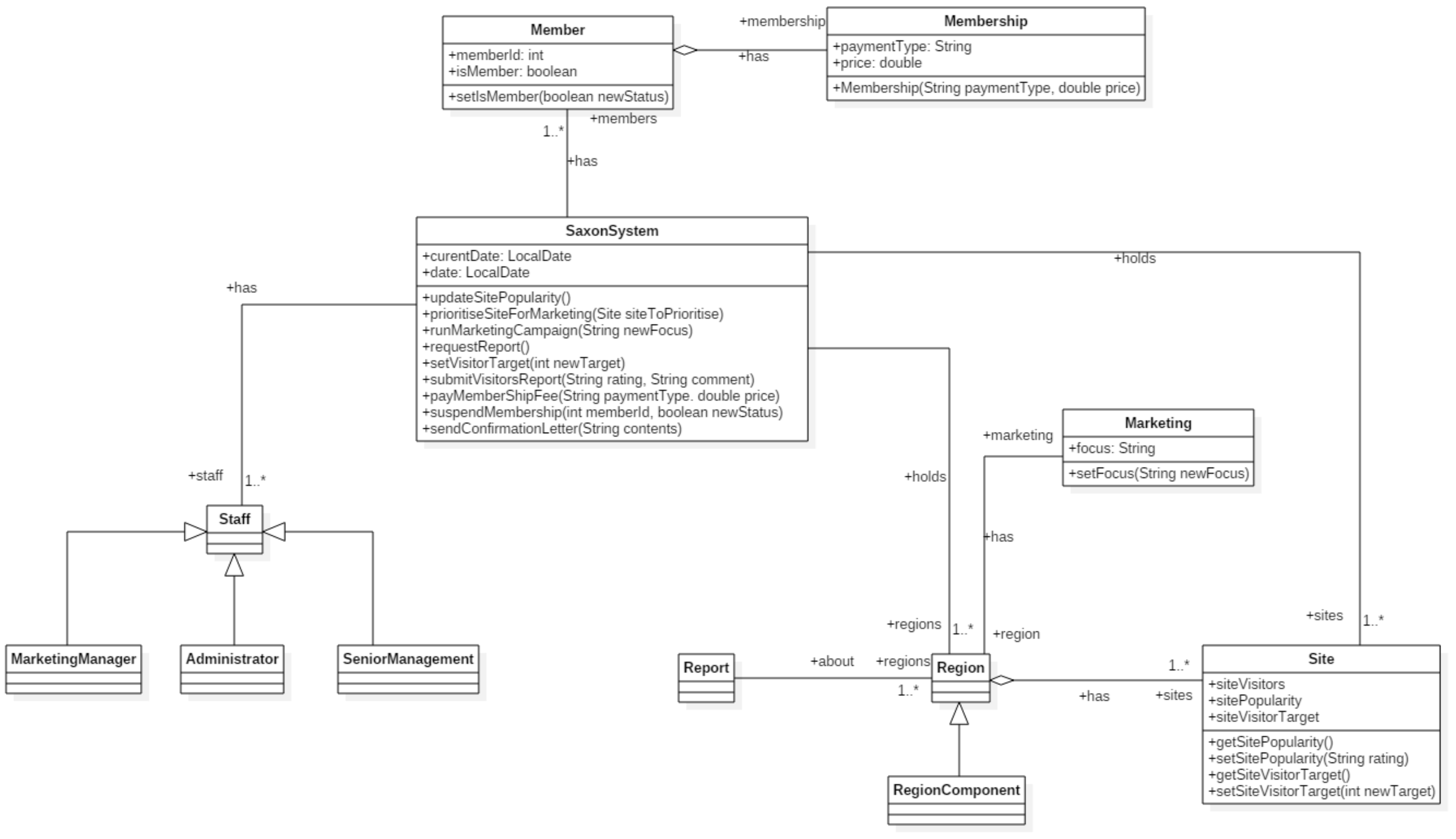
Wide Skills (2015) *Advantages of Using JUnit.* Available at: http://www.wideskills.com/junit/advantages-using-junit (Accessed: 29/11/2016).

Appendix

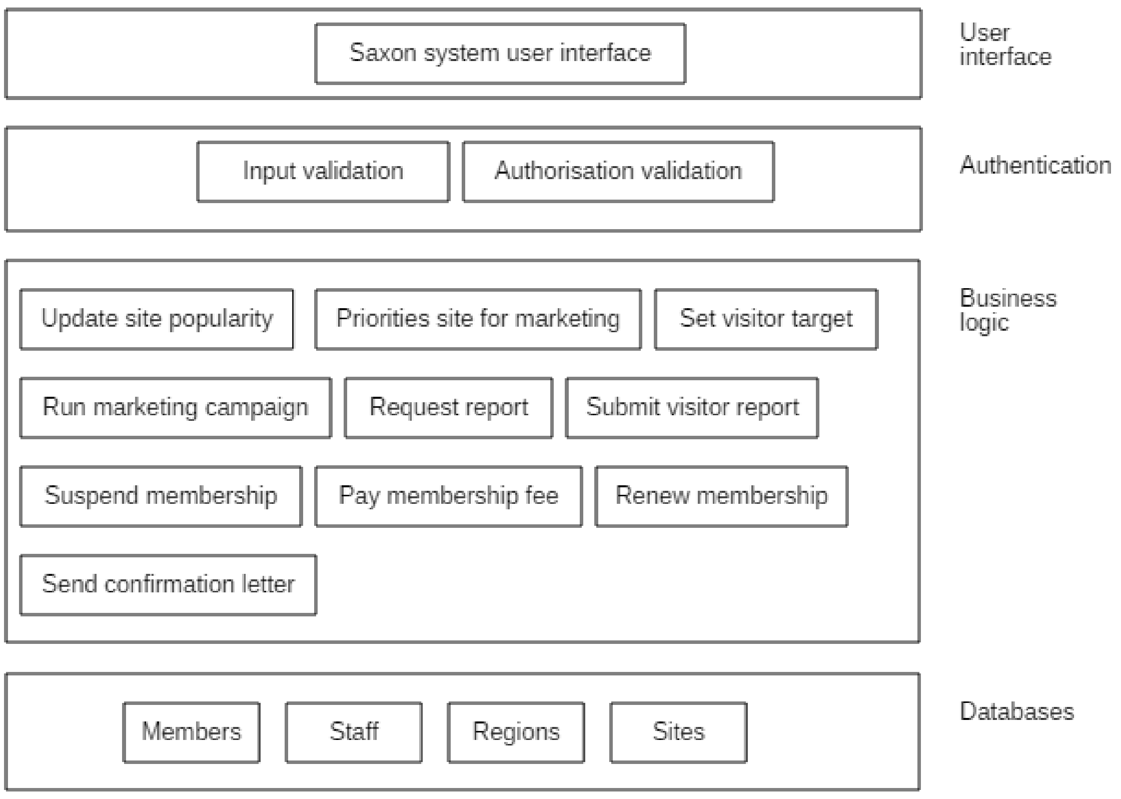
1 Use Case Diagram: Saxon Heritage



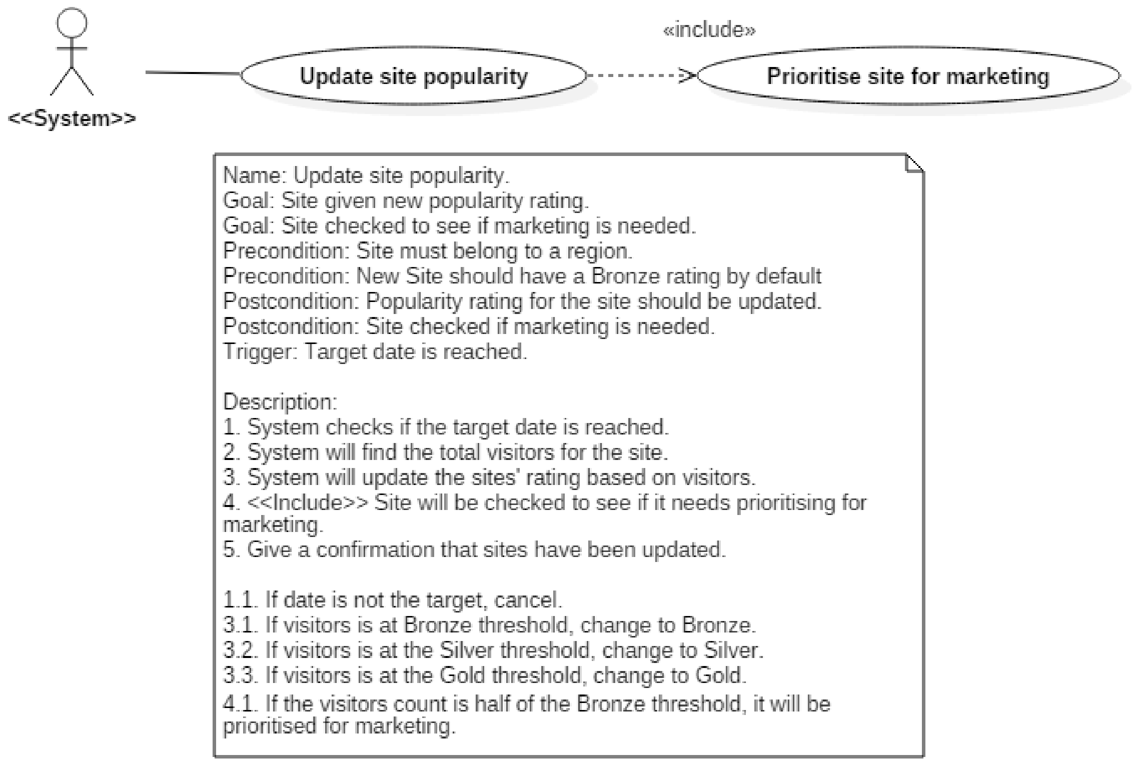
2 Class Diagram: Saxon Heritage



3 Architecture Diagram



4 Use Case Description: Update Site Popularity

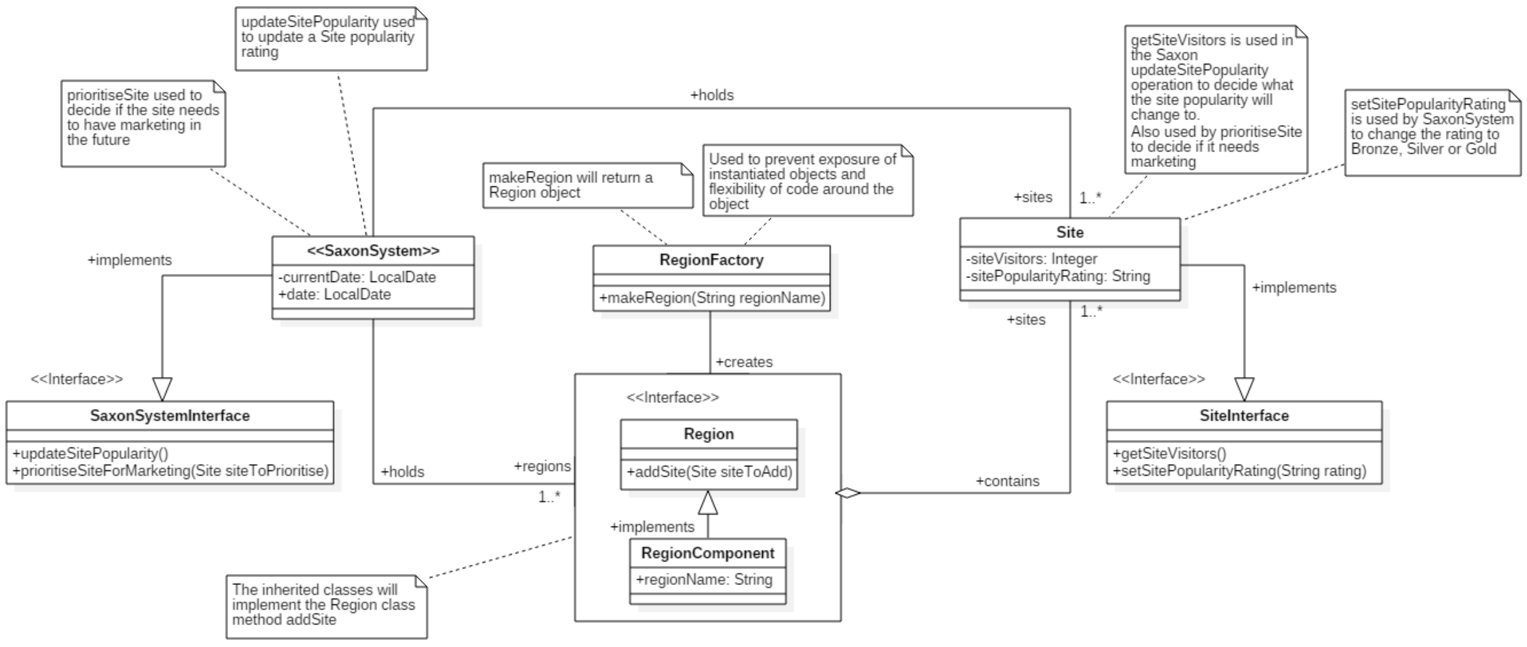


Name: Update site popularity.
Goal: Site given new popularity rating.
Goal: Site checked to see if marketing is needed.
Precondition: Site must belong to a region.
Precondition: New Site should have a Bronze rating by default
Postcondition: Popularity rating for the site should be updated.
Postcondition: Site checked if marketing is needed.
Trigger: Target date is reached.

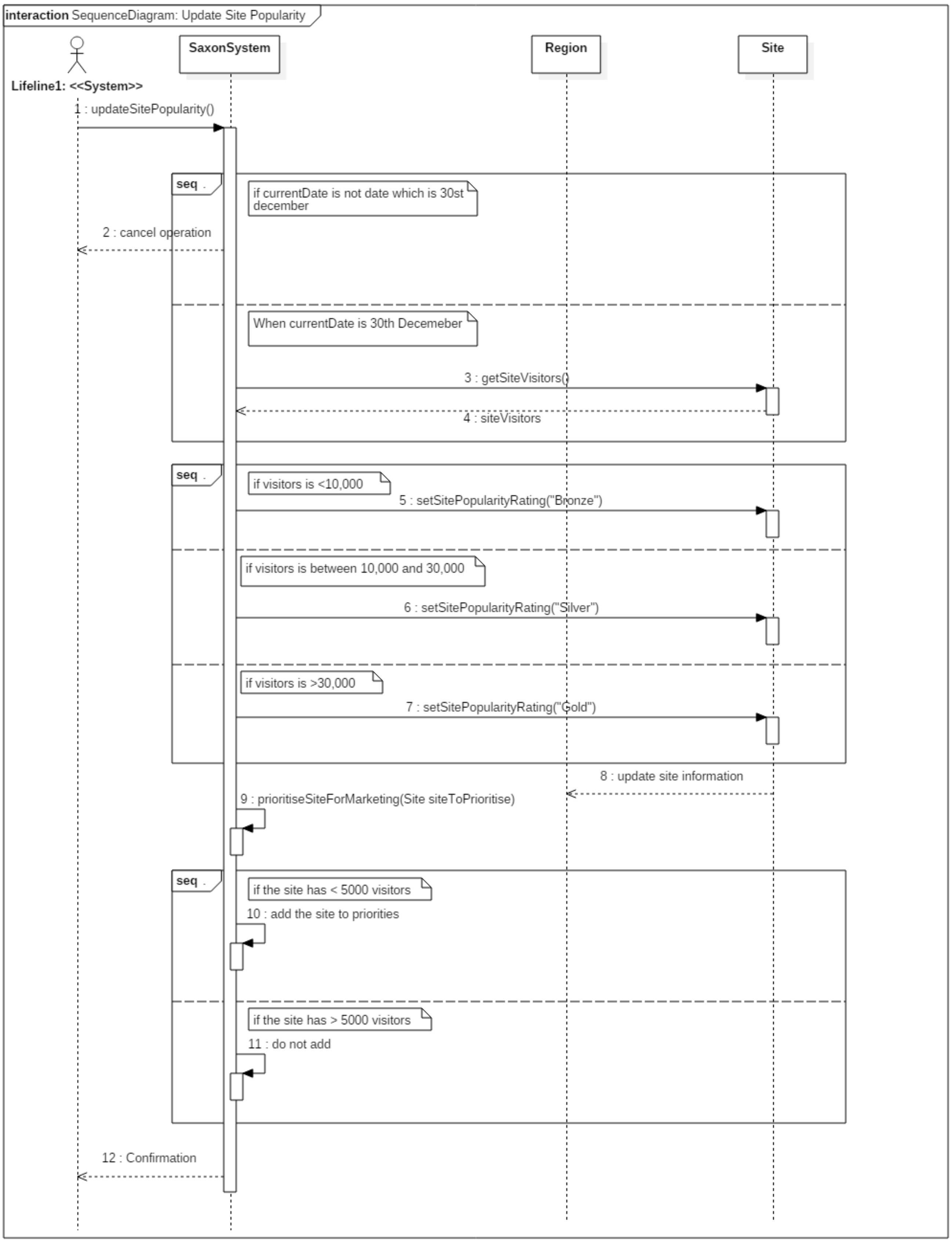
Description:
1. System checks if the target date is reached.
2. System will find the total visitors for the site.
3. System will update the sites' rating based on visitors.
4. <<Include>> Site will be checked to see if it needs prioritising for marketing.
5. Give a confirmation that sites have been updated.

1.1. If date is not the target, cancel.
3.1. If visitors is at Bronze threshold, change to Bronze.
3.2. If visitors is at the Silver threshold, change to Silver.
3.3. If visitors is at the Gold threshold, change to Gold.
4.1. If the visitors count is half of the Bronze threshold, it will be prioritised for marketing.

5 Class Diagram: Update Site Popularity


updateSitePopularity used to update a Site popularity rating
prioritiseSite used to decide if the site needs to have marketing in the future
+holds
getSiteVisitors is used in the Saxon updateSitePopularity operation to decide what the site popularity will change to. Also used by prioritiseSite to decide if it needs marketing
setSitePopularityRating is used by SaxonSystem to change the rating to Bronze, Silver or Gold
makeRegion will return a Region object
Used to prevent exposure of instantiated objects and flexibility of code around the object
+sites
1..*
+implements
<<SaxonSystem>>
-currentDate: LocalDate
+date: LocalDate
RegionFactory
+makeRegion(String regionName)
Site
-siteVisitors: Integer
-sitePopularityRating: String
+implements
+sites
1..*
+creates
<<Interface>>
SaxonSystemInterface
+updateSitePopularity()
+prioritiseSiteForMarketing(Site siteToPrioritise)
<<Interface>>
Region
+addSite(Site siteToAdd)
<<Interface>>
SiteInterface
+getSiteVisitors()
+setSitePopularityRating(String rating)
+holds
+regions
1..*
+contains
+implements
RegionComponent
+regionName: String
The inherited classes will implement the Region class method addSite

6 Sequence Diagram: Update Site Popularity


interaction SequenceDiagram: Update Site Popularity
Lifeline1: <<System>>
SaxonSystem
Region
Site
1 : updateSitePopularity()
seq
if currentDate is not date which is 30st december
2 : cancel operation
When currentDate is 30th Decemeber
3 : getSiteVisitors()
4 : siteVisitors
seq
if visitors is <10,000
5 : setSitePopularityRating("Bronze")
if visitors is between 10,000 and 30,000
6 : setSitePopularityRating("Silver")
if visitors is >30,000
7 : setSitePopularityRating("Gold")
8 : update site information
9 : prioritiseSiteForMarketing(Site siteToPrioritise)
seq
if the site has < 5000 visitors
10 : add the site to priorities
if the site has > 5000 visitors
11 : do not add
12 : Confirmation

7 Source Code

7.1 Region Interface and Classes

```
import java.util.ArrayList;

public interface Region {

    // All classes implementing this interface should have addSite
    public void addSite(Site siteToAdd);
    // Only needed for testing purposes
    public ArrayList<Site> getSites();
}

import java.util.ArrayList;

public class London implements Region {

    //// Properties ////
    // Name of the Region
    private String regionName;
    // The list of sites
    private ArrayList<Site>sites;

    //// Constructor ////
    // No need for parameterised
    public London() {
        regionName = "London";
        sites = new ArrayList<Site>();
    }

    // Method from RegionInterface
    @Override
    public void addSite(Site siteToAdd){
        sites.add(siteToAdd);
    }

    // Get the arrayList of sites
    @Override
    public ArrayList<Site> getSites() {
        return sites;
    }

    //// toString ////

    // To output object as string
    public String toString() {
        return "Region: London\nList of sites: " + sites + "\n";
    }
}

import java.util.ArrayList;
public class Midlands implements Region {

    //// Properties ////
    // Name of the Region
    private String regionName;
    // The list of sites
    private ArrayList<Site>sites;

    //// Constructor ////
    // No need for parameterised
    public Midlands() {
        regionName = "Midlands";
        sites = new ArrayList<Site>();
    }

    // Method from RegionInterface
    @Override
    public void addSite(Site siteToAdd){
        sites.add(siteToAdd);
    }

    // Get the arrayList of sites
    @Override
    public ArrayList<Site> getSites() {
        return sites;
    }

    //// toString ////

    // To output object as string
    public String toString() {
        return "Region: Midlands\nList of sites: " + sites + "\n";
    }
}

import java.util.ArrayList;

public class SouthWest implements Region{

    //// Properties ////
    // Name of the Region
    private String regionName;
    // The list of sites
    private ArrayList<Site>sites;

    //// Constructor ////
    // No need for parameterised
    public SouthWest() {
        regionName = "South West";
        sites = new ArrayList<Site>();
    }

    // Method from RegionInterface
    @Override
    public void addSite(Site siteToAdd){
        sites.add(siteToAdd);
    }

    // Get the arrayList of sites
    @Override
    public ArrayList<Site> getSites() {
        return sites;
    }

    //// toString ////

    // To output object as String
    public String toString() {
        return "Region: South West\nList of sites: " + sites + "\n";
    }
```

```
}

import java.util.ArrayList;

public class NorthWest implements Region {

    //// Properties ////
    // Name of the Region
    private String regionName;
    // The list of sites
    private ArrayList<Site>sites;

    //// Constructor ////
    // No need for parameterised
    public NorthWest() {
        regionName = "North West";
        sites = new ArrayList<Site>();
    }

    // Method from RegionInterface
    @Override
    public void addSite(Site siteToAdd){
        sites.add(siteToAdd);
    }

    // Get the arrayList of sites
    @Override
    public ArrayList<Site> getSites() {
        return sites;
    }

    //// toString ////

    // To output object as string
    public String toString() {
        return "Region: North West\nList of sites: " + sites + "\n";
    }
}

import java.util.ArrayList;

public class SouthEast implements Region {

    //// Properties ////
    // Name of the Region
    private String regionName;
    // The list of sites
    private ArrayList<Site>sites;

    //// Constructor ////
    // No need for parameterised
    public SouthEast() {
        regionName = "South East";
        sites = new ArrayList<Site>();
    }

    // Method from RegionInterface
    @Override
    public void addSite(Site siteToAdd){
        sites.add(siteToAdd);
    }
    // Get the arrayList of sites
    @Override
    public ArrayList<Site> getSites() {
        return sites;
    }

    //// toString ////

    // To output object as string
    public String toString() {
        return "Region: South East\nList of sites: " + sites + "\n";
    }
}

import java.util.ArrayList;

public class SouthWest implements Region{

    //// Properties ////
    // Name of the Region
    private String regionName;
    // The list of sites
    private ArrayList<Site>sites;

    //// Constructor ////
    // No need for parameterised
    public SouthWest() {
        regionName = "South West";
        sites = new ArrayList<Site>();
    }

    // Method from RegionInterface
    @Override
    public void addSite(Site siteToAdd){
        sites.add(siteToAdd);
    }

    // Get the arrayList of sites
    @Override
    public ArrayList<Site> getSites() {
        return sites;
    }

    //// toString ////

    // To output object as String
    public String toString() {
        return "Region: South West\nList of sites: " + sites + "\n";
    }
}
```

7.2 Region Factory

```java
public class RegionFactory {

    // To return different regions as Region type based on string argument
    public Region makeRegion(String regionName) {

        if (regionName.equalsIgnoreCase("LONDON")) {
            return new London();
        }
        else if (regionName.equalsIgnoreCase("MIDLANDS")){
            return new Midlands();
        }
        else if (regionName.equalsIgnoreCase("SOUTH EAST")){
            return new SouthEast();
        }
        else if (regionName.equalsIgnoreCase("SOUTH WEST")){
            return new SouthWest();
        }
        else if (regionName.equalsIgnoreCase("NORTH EAST")){
            return new NorthEast();
        }
        else if (regionName.equalsIgnoreCase("NORTH WEST")){
            return new NorthWest();
        }
        // Otherwise return nothing
        return null;
    }
}
```

7.3 SiteInterface and Site

```java
public interface SiteInterface {
    // Methods to be implemented
    public int getSiteVisitors();

    public void setSitePopularity(String newSitePopularity);
}

public class Site implements SiteInterface{
    //// Properties ////
    private String siteName;
    private int siteVisitors;
    private String sitePopularity;

    //// Constructor ////
    //Default
    public Site() {
        siteName = "New Site";
        siteVisitors = 0;
        sitePopularity = "Bronze";
    }
    // Parameterised
    public Site(String siteName, int siteVisitors, String sitePopularity) {
            this.siteName = siteName;
            this.siteVisitors = siteVisitors;
            this.sitePopularity = sitePopularity;
    }
    //// Getters ////

    // Get the name
    public String getSiteName() {
        return siteName;
    }

    // Get the amount of visitorts
    public int getSiteVisitors() {
        return siteVisitors;
    }

    // Get the popularity of the site
    public String getSitePopularity() {
        return sitePopularity;
    }

    //// Setters ////

    // Set a new name for the site
    public void setSiteName(String newSiteName) {
        this.siteName = newSiteName;
    }

    // Set a new visitor count for the site
    public void setSiteVisitors(int newSiteVisitors) {
        this.siteVisitors = newSiteVisitors;
    }

    // Set a new popualrity rating
    public void setSitePopularity(String newSitePopularity) {
        this.sitePopularity = newSitePopularity;
    }

    //// toString ////

    // Output object as string
    public String toString() {
        return ("\nSite Name: " + siteName + "; Visitors: " + siteVisitors
                + "; Popularity Rating: " + sitePopularity );
    }
}
```

7.4 SaxonSystemInterface and SaxonSystem

```java
public interface SaxonSystemInterface {
    // Methods that must be implemented
    public String updateSitePopularity();
    public void prioritiseSiteForMarketing(Site siteToPrioritise);
}

import java.time.LocalDate;
import java.util.ArrayList;

public class SaxonSystem implements SaxonSystemInterface{

    //// Properties ////
    private ArrayList<Region> regions;
    private ArrayList<Site> sites;
    private ArrayList<Site> prioritsedSites;
    private LocalDate currentDate;

    // Will be used as a way of checking if the current data is 30th
    private LocalDate date = LocalDate.parse("2016-12-30");

    //// Constructor ////

    // Default, no need for parameterised
    public SaxonSystem() {
        regions = new ArrayList<Region>();
        sites = new ArrayList<Site>();
        prioritsedSites = new ArrayList<Site>();
        currentDate = LocalDate.parse("2016-12-30");
    }

    // USE CASE - Update site popularity
    // Checks date, loops through Sites, gets the siteVisitors,
    // sets a new sitePopularity based on visitors,
    // Checks if marketting needed, finishes
    public String updateSitePopularity() {
        if (currentDate.equals(date)) {
            Site s;

            for (int i = 0; i < sites.size(); i++) {
                s = sites.get(i);

                int visitors = s.getSiteVisitors();

                if (visitors < 10000) {
                    s.setSitePopularity("Bronze");
                } else if (visitors >= 10000 && visitors < 30000) {
                    s.setSitePopularity("Silver");
                } else {
                    s.setSitePopularity("Gold");
                }

                prioritiseSiteForMarketing(s);
            }
                return ("All site popularity ratings have been updated"
                    + "\n" + regions);

        } else {
            return ("Cant Update Popularity yet. Wait till 30th Dec");
        }
    }

    // USE CASE - Prioritise site for marketing
    // Site as argument, check siteVisitors, add to martketting
    public void prioritiseSiteForMarketing(Site siteToPrioritise){
        if(siteToPrioritise.getSiteVisitors() < 5000) {
            // Add it to a list
            prioritsedSites.add(siteToPrioritise);
        }
    }

    //// Getter ////
    // Return the currentDate
    public LocalDate getCurrentDate() {
        return currentDate;
    }

    public ArrayList<Site> getPrioritisedSites() {
        return prioritsedSites;
    }

    //// Setter ////
    // Set the currentDate to compare
    public void setCurrentDate(String d) {
        this.currentDate = currentDate.parse(d);
    }

    //// Add methods ////

    // Adding region objects
    public void addRegion(Region regionToAdd) {
        regions.add(regionToAdd);
    }

    // Adding site objects
    public void addSite(Site siteToAdd) {
        sites.add(siteToAdd);
    }

    // BUSINESS LOGIC - Add a site to priority
    public void addToPriorities(Site siteToAdd) {
        prioritsedSites.add(siteToAdd);
    }

    //// toString ////

    // To print out the list of prioritised sites
    public String printPrioritisedSites() {
        return ("Sites that are now priorities\n" + prioritsedSites);
    }

    // To print out the object as string
    public String toString() {
        return "Saxon System\nRegion Details:\n" + regions;
    }
}
```

7.5 SaxonApplication

```
public class SaxonApplication {

    public static void main(String[] args) {
        //// Make the SaxonSystem object ////
        SaxonSystem saxon = new SaxonSystem();

        //// Making Sites ////
        // Make 2 sites to go in each Region
        // SITES FOR LONDON
        Site londonSite1 = new Site("London Place 1", 200, "Silver");
        Site londonSite2 = new Site("London Place 2" , 10000, "Bronze");

        // SITES FOR MIDLANDS
        Site midlandsSite1 = new Site("Midlands Place 1", 100, "Bronze");
        Site midlandsSite2 = new Site("Midlands Place 2" , 30000, "Gold");

        // SITES FOR NORTHEAST
        Site northEastSite1 = new Site("North East Place 1", 0, "Gold");
        Site northEastSite2 = new Site("North East Place 2", 1, "Silver");

        // SITES FOR NORTHWEST
        Site northWestSite1 = new Site("North West Place 1", 40000,
"Silver");
        Site northWestSite2 = new Site("North West Place 2", 20000,
"Silver");

        // SITES FOR SOUTHEAST
        Site southEastSite1 = new Site("South East Place 1", 1000, "Gold");
        Site southEastSite2 = new Site("South East Place 2", 60000,
"Gold");

        // SITES FOR SOUTHWEST
        Site southWestSite1 = new Site("South West Place 1", 2, "Gold");
        Site southWestSite2 = new Site("South West Place 2", 30000,
"Silver");

        //// Using the factory to make regions ////
        RegionFactory regionFactory = new RegionFactory();
        // Making the regions
        Region london = regionFactory.makeRegion("LONDON");
        Region midlands = regionFactory.makeRegion("MIDLANDS");
        Region northEast = regionFactory.makeRegion("NORTH EAST");
        Region northWest = regionFactory.makeRegion("NORTH WEST");
        Region southEast = regionFactory.makeRegion("SOUTH EAST");
        Region southWest = regionFactory.makeRegion("SOUTH WEST");

        //// Add the sites to the regions, 2 sites for each regions ////
        london.addSite(londonSite1);
        london.addSite(londonSite2);
        midlands.addSite(midlandsSite1);
        midlands.addSite(midlandsSite2);
        northEast.addSite(northEastSite1);
        northEast.addSite(northEastSite2);
        northWest.addSite(northWestSite1);
        northWest.addSite(northWestSite2);
        southEast.addSite(southEastSite2);
        southEast.addSite(southEastSite2);
        southWest.addSite(southWestSite1);
        southWest.addSite(southWestSite2);
        saxon.addSite(londonSite1);
        saxon.addSite(londonSite2);
        saxon.addSite(midlandsSite1);
        saxon.addSite(midlandsSite2);
        saxon.addSite(northEastSite1);
        saxon.addSite(northEastSite2);
        saxon.addSite(northWestSite1);
        saxon.addSite(northWestSite2);
        saxon.addSite(southEastSite2);
        saxon.addSite(southEastSite2);
        saxon.addSite(southWestSite1);
        saxon.addSite(southWestSite2);


        // Add the Regions to the saxon System
        saxon.addRegion(london);
        saxon.addRegion(midlands);
        saxon.addRegion(northEast);
        saxon.addRegion(northWest);
        saxon.addRegion(southEast);
        saxon.addRegion(southWest);

        System.out.println("Before update");
        System.out.println(saxon);
        System.out.println("\n\n");

        System.out.println(saxon.updateSitePopularity());

        System.out.println("\n\n\n\n\n");
        System.out.println(saxon.printPrioritisedSites());
    }
}
```