VOC Montor Process QA

Sentient

11/30/2013

Table of Contents

State Definition………………………………………………………………………………………………………………………………..3-5

Transition Definition………………………………………………………………………………………………………………………...5-6

Coding Standard ………………………………………………………………………………………………………………………………7-10

State Definition

There are six states in the VOC Monitor Process QA: Define Requirements, Analysis, Design, Code, Testing, and Fix. Following is a description of each state and a definition of each artifact related to that state. The quality of artifacts are tested by peer review from each member of the development team. Once reviewed by all team members final analysis is executed by the team lead.

Phase 1: Define Requirements

In this state the functionality of the VOC Monitoring System is documented by dividing the system into conceptual groups. Those groups are further divided into specific features. In this state the market that the monitor will appeal to is also defined. The artifacts that result from this state are as follows:

Functional Requirements Document

This document is what results from dividing the system into conceptual pieces and breaking those pieces into extremely specific functionality.

Initial Business Case

This document is an argumentative document that depicts the users of the system, their motivation, competing systems, a cost analysis, and an overall argument for why the system should be created.

Phase 2: Analysis

In this state the overall functionality of the system is analyzed. This analysis is executed through written documents that describe a user in the process of using the system and how the system would be tested. The analysis state is used to comb through the system to ensure necessary requirements were not missed. The artifacts that result from this state as follows:

Functional Architecture

This model of the system is used to visually represent the functional requirements document and derive dependencies between features.

Test Scripts (Black Box )

This document describes the steps that would be taken in order to test the functionality of the system. These tests are not code based.

User Stories

This document describes a user implementing the system at different levels of the system. User stories are also written in preparation of the Functional Requirements Document but are solidified as a single document in this phase.

Phase 3: Design

In this state the underlying functionality of the system is organized into classes and dependencies are realized. Unlike Analysis, this phase works with the code and hardware that runs the VOC Monitoring System. The system is divided into objects that will act as classes. The functionality of these classes is defined. Scripts to test the functionality behind each procedure are also defined. The artifacts associated with this state are as follows:

Test Scripts (White Box)

Black box test scripts are similar to white box test scripts except white box test scripts are code based and test specific features. Black box test scripts rely on many working pieces where white box tests those many features one at a time.

Logical Architecture

The Logical Architecture is a lower level visual representation of the VOC Monitoring System that describes the classes and the functions within those classes.

Class Models

Class Models are visual representations of the system that depict requirements being used through system actors.

Phase 4: Code

This state deals with the actual creation of the source code for the system. The Logical Architecture created in the design phase is used as a blue print to build piece by piece. The artifacts involved in this state are as follows:

Source Code

The source code is the bread and butter of the system and is written in C,

C++, and C#.

Phase 5: Testing

This state deals with testing that the source code created in the previous phase is correct and up to the VOC Monitoring coding standards. The use of tests in this system will reveal bugs in the system and those bugs will be repaired. The artifacts created during this phase are as follows:

Unit Test

Unit Tests are the code implementation of white box test. Each unit test checks a single system process for accurate results.

Phase 6: Fix

This state deals with realizing the errors that have been made throughout the last four phases. These mistakes are found through testing results, inability to fulfill requirements with current hardware or design, and scope miscalculations. The artifacts involved in this state are the artifacts of all the previous states that will be modified based on issues.

Transition Definition

There are six transitions in the VOC Monitoring System Process QA: Define Requirements to Analysis, Analysis to Design, Design to Code, Code to Testing, Testing to Fix, and Fix back to Define Requirements. While some transitions are simultaneous others require artifact and implementation completion before moving on. The description of these transition rules follow.

Transition 1: Define Requirements to Analysis

In order to transition from Defined Requirements to Analysis the customers and development team have to agree on the specific functionality of the system. The development team must ensure that the functionality the customer wants is viable. Therefore, in order to transition from Defined Requirements to Analysis the Functional Requirements Document must be in functional order and signed off by both parties. Customers need to attempt to confirm that the system will do what they need it to. The Initial Business Case must then appeal to the same market that the customers intend the product for.

Transition 2: Analysis to Design

In order to transition from Analysis to Design the entire system needs to be described in writing from the perspective of a user implementing the system. The Design phase cannot commence until the development team is confident that their every feature has been describe in fine detail in order to ensure that new functional requirements will appear during following states. There will be a User Story describing each step of the system as well as Test Scripts for each element in the User Stories. The Functional Architecture will reflect the current Functional Requirements Document.

Transition 3: Design to Code

In order to transition from the Design state to the Code state there must be a complete blueprint for the entire system. The Logical Architecture is like the pseudo code for the system. It describes how classes interact down to the functions that will be called. To ensure a consistent system the Logical Architecture must be in functioning order; a complete version with no outlining classes or unused functionalities.

Transition 4: Code to Test

The transition from Code to Test is more lucrative than any of the transitions before it. Code must be tested piece by piece. If the entire code base was written before any testing was committed, then any bugs that were introduced early would cascade through the entire system. The transition from Code to Test is done many times throughout the Code state. Each benchmark that is coded must be tested before moving forward to the next benchmark. Test Scripts are therefore designed for each process and implemented as that Code is completed.

Transition 5: Test to Fix

This transition is also lucrative. The Fix state relies on the results of the Test state. The Test state is intermittently entered by the Code phase. The Fix phase is therefore entered throughout the Code state. These three states are tightly woven together and rely on the continuous changes to artifacts.

Transition 6: Fix to Define Requirements

While the Fix state is intermittently entered, the Define Requirements State is not. In order to transition from the Fix State to the Define Requirements State, an issue with the overall design, a specific requirement, or the implementation design of a requirement must be revealed by the Fix state. From there the issue is sent through the process again until the system is complete.

Coding Standard

In order to ensure code consistency a coding standard is necessary to provide a blue print for Sentient development members to refer to. This code standard addresses the following: Naming Conventions for Variables, Classes, and Functions, Code Layout including fonts, indentation, function, and classes, and Documentation covering header blocks, function comments, and code comments.

Naming Conventions

Variables will not include verbs but consist of nouns and adjectives. No part of a variable will hold a key or reserved word for the language in use. Similar variables will not be discerned through numbering without an underscore before the integer. All variables will be named in camel case, meaning the first letter of the name will always be lowercase, and any subsequent words within the variable name begin with a capital letter.

For Example:

Not Acceptable: Reading\_File, TrueFlag, Writing\_File, File2

Acceptable: arduinoFile, flag, file\_2, storeSensorData

Classes will not include verbs or adjectives but consist only of a single noun. No numbers, underscores, or like characters will be used to name a class. The first character will be capitalized and the rest lowercase.

For Example:

Not Acceptable: Class Ardunio\_Sensor, Class Sensor2, Class VOC

Acceptable: Class Sensor, Class Formaldehyde, Class Voc

Functions will start with a verb. Each new word will be capitalized. No numbers, underscores, or the like will be used to name a function.

For Example:

Not Acceptable: File(), Arduino(), Reading\_From\_File()

Acceptable: ReadingArduinoFile(), PrintVocLevel()

Code Layout

All code will be kept in the default font set by Visual Studio.

A tab will be inserted after each bracket. Brackets occupy their own line.

For Example:

Not Acceptable

int ReadFile(){

int flag = 0;

}

Acceptable

int ReadFile()

{

int flag = 0;

}

Function will consist of no more than fifty lines. All variables will be instantiated at the start of the function. There will be one exit for each function.

For Example:

Not Acceptable:

int CheckingExit( int choice)

{

if( choice == 5)

return 5;

else

return 0;

}

Acceptable:

int CheckingExit( int choice)

{

int flag = 0;

if( choice == 5)

flag = 5;

else

flag = 0;

return flag;

}

Classes will depict public and private regardless of attributes or members within them. Manager functions will be included and defined in all classes as well as described first in both the header and dot file. Protected only need be used when implementing inheritance.

Not Acceptable:

Class Sensor

{

Int ReadSensor();

}

Acceptable:

Class Sensor

{

public:

void Sensor();

void Sensor( const Sensor\* copy );

void ~Sensor();

Sensor& Operator=(const Sensor\* equals);

int ReadSensor();

private:

}

Documentation

Class headers will be in the following format:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* void ReadInputData( int plantArray[] );

\*

\* Purpose: This function reads the input data for each plant.

\*

\* Entry: LastPlantNumber is the declared size of the array

\* plantArray.

\*

\* Exit: For plantNumber = 1 through lastPlantNumber:

\* plantArray[ plantNumber - 1 ] equals the total

\* production for plant number plantNumber.

\*

\* void GetTotal( int &totalSumPerPlant );

\*

\* Purpose: This function reads non-negative integers from the

\* keyboard and accumulates the total sum per plant.

\*

\* Entry: Must have a valid plant number.

\*

\* Exit: Accumulate the total units produced by each

\* department of a plant and end input with a

\* sentinel number ( -1 ).

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Functions will be documented with a comment:

//This function reads in a file containing VOC levels

void ReadVocLevels()

{

…

}

Comments will be used to describe a complexity of the code:

Not Acceptable:

flag = ReadFile(); //Something in this function isn’t working

Acceptable:

flag = ReadFile(); //Returning if file exists