CS 578 – Software Architectures
Spring 2011
Course Project

Due: *Friday, April 29th,2011*

– *see course websites for submission details* –

Introduction

For this project, you will form teams of **two or three persons**. The first part of the project will require you to define your own architectural smells. For the second part of the project, you will be detecting architectural smells from an XML representation of the recovered architecture of the OODT File Manager.

You must come up with **two new smells NOT** defined in any of the following papers:

> Joshua Garcia, Daniel Popescu, George Edwards and Nenad Medvidovic, Toward a Catalogue of Architectural Bad Smells, *Proceedings of the 5th International Conference on the Quality of Software Architectures (QoSA): Architectures for Adaptive Software Systems*, June 2009.
>
> Formalization of architectural concepts for smells and smell definitions - http://softarch.usc.edu/~josh/arch_and_smells_defs.pdf

You will need to define each architectural smell and describe **why each smell is architectural**. Just like the previous assignment, you will need to describe how the smell affects maintainability, provide an example of the smell, describe how it may be refactored, and explain what non-functional properties besides maintainability may be affected by refactoring the smell.

Here are some strategies that may help you find new architectural smells:

- Take an existing architectural pattern or style and break it in a way that satisfies the definition of "architectural smell''
- Consider the different ways good design principles (e.g. separation of concerns, isolation of change, abstraction, coupling and cohesion, simplicity, etc.) can be violated in an architecture
    - You need not constrain yourself to software engineering principles focused on in this course. For example, other software engineering principles include the principle of incrementality, which prescribes that a system be constructed by building a small piece of it and then verifying and validating that piece before building the next piece.
- Take an existing code smell or anti-pattern and see if you can find an analog of it at the architectural level
- Consider the guidelines for designing for NFPs that you learned in the course and see if those guidelines can give you ideas about how failure to follow one or more of them can result in architectural smells
- Consider connector dimensions, sub-dimensions, or types that can be combined in a way that negatively affects maintainability

For the second part of the assignment, we will provide you three separate XML representations of a recovered architecture. You will need to implement **two** smell detection algorithms. **One of those smell detection algorithms must detect a previously defined smell in the two papers above. The other smell detection algorithm must detect one of the smells your team came up with.**

The architecture for the file manager was recovered using a hierarchical clustering technique for component recovery called the Weighted Combined Algorithm. This technique hierarchically creates clusters, groupings of software entities (e.g., functions, classes, variables, etc.), and uses the clusters to identify components. In this specific case, we clustered classes to obtain clusters that either implement primarily components or connectors. Architectural associations between components and connectors are obtained through the call and reference relationships between classes in this technique. If you are interested, more detail about this technique can be found here:

> Onaiza Maqbool, Haroon Babri, "Hierarchical Clustering for Software Architecture Recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759-780, Aug. 2007, doi:10.1109/TSE.2007.70732

The produced clusters and the associations are represented as nodes and edges, respectively, in a graph in the file named **oodt-filemgr_smellArchGraph.xml**. This XML file describes each cluster and its constituent classes. The file contains a list of edges between clusters indicating the source cluster and destination cluster of the edge.

Part of this recovered architecture includes a representation of "concerns.'' A concern is a concept, role, responsibility, or purpose in a software system. For example, concerns can be "presentation of data," "matrix manipulation," "weather analysis," "event interaction," etc. To represent concerns, we use a statistical language model that models text by obtaining a probability distribution over words from it called "topics." These models are called topic models. One of the simplest of these topic models is represented using a language model called Latent Dirichlet Allocation (LDA). For example, a topic can be about events and the highest probability words for the events topic may include words such as "event," "send," "receive," "filter," etc. Probabilities associated with words can look like the following for the event topic:

Word	**Probability**
Event	0.3
Send	0.1
Receive	0.1

Note that the above example does not show lower probability words as indicated by the fact that the probabilities do not add up to 1.

Architectural concerns are represented using topics from LDA. Each cluster, which either implements a component or connector, also has a probability distribution over topics associated with it called a document-topic distribution. Therefore, if you had a topic about "events" and a topic about "weather" the following table can represent a component's document-topic distribution:

Component 1	
Topic	**Probability**
Events	0.1
Weather	0.5

Therefore, you can infer that Component 1 has a little to do with send and receiving events and a lot to do with weather. Note that the above example does not show all the topics since the probabilities do not add to 1.

You do not need to know details about LDA, but if you are interested you can read about it in the following papers:

> Steyvers, M. and Griffiths, T., "Probabilistic topic models," *Handbook of latent semantic analysis*, 2007.
>
> Blei, D.M. and Ng, A.Y. and Jordan, M.I., "Latent dirichlet allocation," *The Journal of Machine Learning Research*, 2003.

Lastly, each topic has been labeled as either application-specific or application-independent where the former label is associated with components and the latter is associated with connectors.

We provide you another XML file, **oodt-filemgr_smellArch_specified.xml** , that lists all the topics extracted using LDA, their most probable words, each of their types (application-specific or application-independent), and the probability of each topic for every cluster. We also automatically labeled the type (application-specific or application-independent) for each cluster. The type of the cluster tells you whether or not it is a component or connector.

The XML file contains a tag <topic> for each probabilistic topic with attributes "id" and "type." The id is a unique identifier for the topic. The type attribute can take the value "spec," which indicates that the topic is application-specific, or "indep," which indicates that the topic is application-independent. The <word> tags are the top occurring words in the topic in descending order of the probability that the word appears in the topic.

Each <Cluster> tag represents a general architectural element that can either be a component or a connector. The <Cluster> tag has an attribute "type" indicating whether it is application-specific, i.e., a component, or application-independent, i.e., a connector. It consists of a <name> tag, a <classes> tag that describes the classes that are part of the cluster, and a <doc-topic> tag that represents the document-topic distribution for the cluster. The <classes> tag comprises a set of <class> tags that each contain a set of <method> tags. The <doc-topic> tag comprises a set of <topic> tags whose "id" is taken from the set of topics listed at the beginning of the document and takes on a value between 0 and 1 representing the proportion of that topic for the cluster.

We also provide you a third xml file, **oodt-filemgr_methodInfo.xml**, that describes method names, the parameter and return value types for methods, what classes they belong to, and what clusters those classes belong to. **Note:** You can simply take the public methods listed in the clusters to be architectural interfaces for the smell detection portion of your project.

We ask that you implement two architectural smell detection algorithms using these three XML-based views of the OODT File Manager's architecture. You need to **write the pseudocode for your algorithms**, **implement them**, and **analyze their false positive rates**.

Deliverable

The deliverable of your assignment is a written assessment report and the code for your detection algorithms. The report should be written using the font Times New Roman in font size 12 and single line spacing for each paragraph. Please check your spelling and use proper grammar.

Please adhere to the following format when you write your report on architectural smells.

1. Name of your architectural smell
    a. Definition of your architectural smell
    b. Why is the smell architectural?
    c. How does the smell negatively affect maintainability?
    d. Provide an example as to how your smell may manifest itself in an existing system. This example should be similar to the explanation found in the smells paper above.
    e. How can the smell possibly be refactored?
    f. What non-functional properties can refactoring the smell affect besides lifecycle properties?
2. Detection of architectural smells
    a. Write out the algorithm you implemented using pseudocode
    b. Provide an explanation of the algorithm you wrote in pseudocode
    c. Show the false and true positives of your algorithm and discuss the false positive rate of your detection algorithm
        i. A table is a nice way to present this part of your assignment
        ii. Indicate for each instance positively detected as a smell
            1. Which architectural elements are involved
            2. Which specific type of smell was detected
        iii. Discuss the number of false positives compared to true positives
        iv. Discuss why the false positives occurred and how they may be possibly fixed
    d. Implementation of your detection algorithms in a ZIP file
        i. You must implement your detection algorithm in Java. Java provides many built-in classes that can help you read in XML files.
        ii. A description about how to compile and run your code
            1. Your code does need to be compilable and runnable