Assignment 3

TI2206 Software Engineering Methods (2015-2016 Q1)
Date: 09/10/2015
Group Number: 0
Group Members:

Thomas Baars - tcbaars@gmail.com
Lars Ysla - yslars@gmail.com
Boris Schrijver - boris@radialcontext.nl
Adriaan de Vos - adriaan.devos@gmail.com
Dylan Straub - d.r.straub@student.tudelft.nl

Table of Contents

Exercise 1 - 20-Time, Reloaded

1. Requirements of Extensions

The planned extension for our project is an improved method of exception handling. Our current method of exception handling was to catch expected exceptions and print the stack trace. The planned improvements is to implement better exception handling that follows the "Fail-Fast"[1] approach. As well a better method of displaying the encountered exception. The following requirements outline the functional and non-functional requirements of the extension. These requirements are categorised by their priority, where critical requirements are necessary for the extension to be considered a success and non-critical requirements which will only be implemented if time permits.

Critical Requirements

- Errors which may occur during loading sound files shall be handled.
- Errors which may occur during playing sound files shall be handled.
- Errors which may occur during loading image files shall be handled.
- Errors which may occur during loading music files shall be handled.
- Errors which may occur during playing music files shall be handled.
- Errors which may occur during loading sprite sheets shall be handled.
- Errors which may occur during gameplay shall be handled.
- Where handling errors shall be defined as:
    - Exceptions encountered shall be logged.
    - A message describing the error encountered shall be displayed.
    - The message shall only be displayed once.
    - If the error is critical, then the game should be exited.
        - Where critical error means that the application is no longer playable.
    - If the error is non-critical, then the game should continue.
        - Where non-critical means the application is still playable.

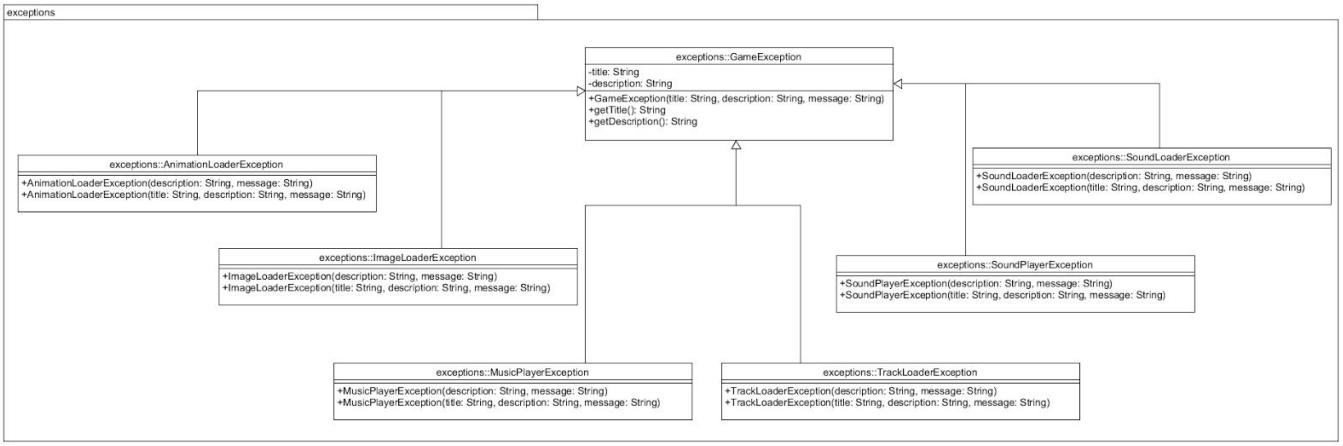
Non-Critical Requirements

- The message displayed shall be a dialogue box.
- The message shall display a description of the error.
- The message shall display a title indicating the related feature where the error originated from.
- The message shall display an icon which indicates how critical the error encountered is.

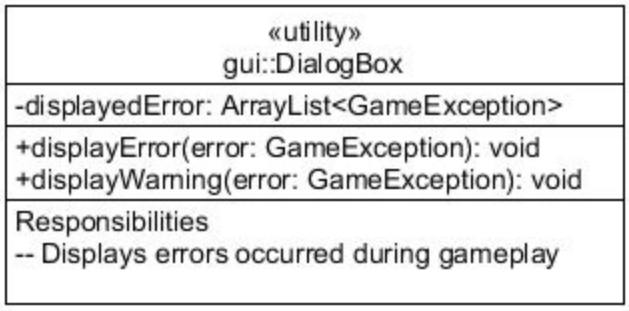
[1] Fail Fast - http://www.martinfowler.com/ieeeSoftware/failFast.pdf

2. Analysis and Design

The analysis and design documentation for the planned extension is as follows. The exception classes that are planned shall facilitate the error handling of errors which may occur during resource loading. The classes are described below:



The classes described are responsible for knowing a description of the error occurred. This is then displayed by a dialog box. The specification of this .dialog box is below:



The dialog box can display critical errors, which as mentioned before, are errors which leave the game in an unplayable state. After which the application is exited. Non-critical errors, or ‘warnings’, after which the game is still in a playable state, can be displayed after which the user shall be able to continue the game.

Exercise 2 - Design Patterns

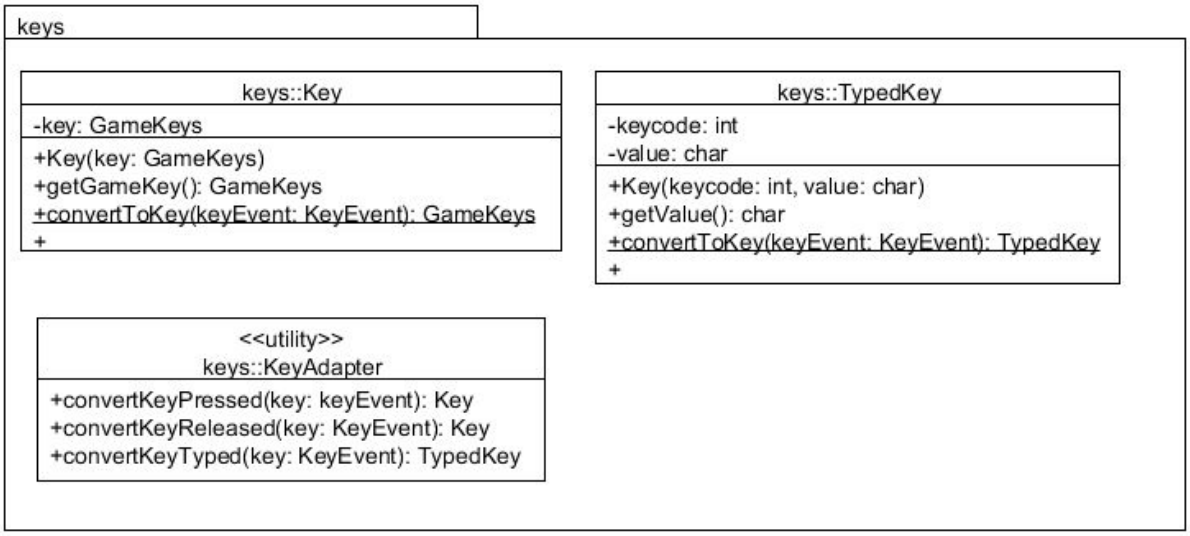
Design Pattern I - Adapter

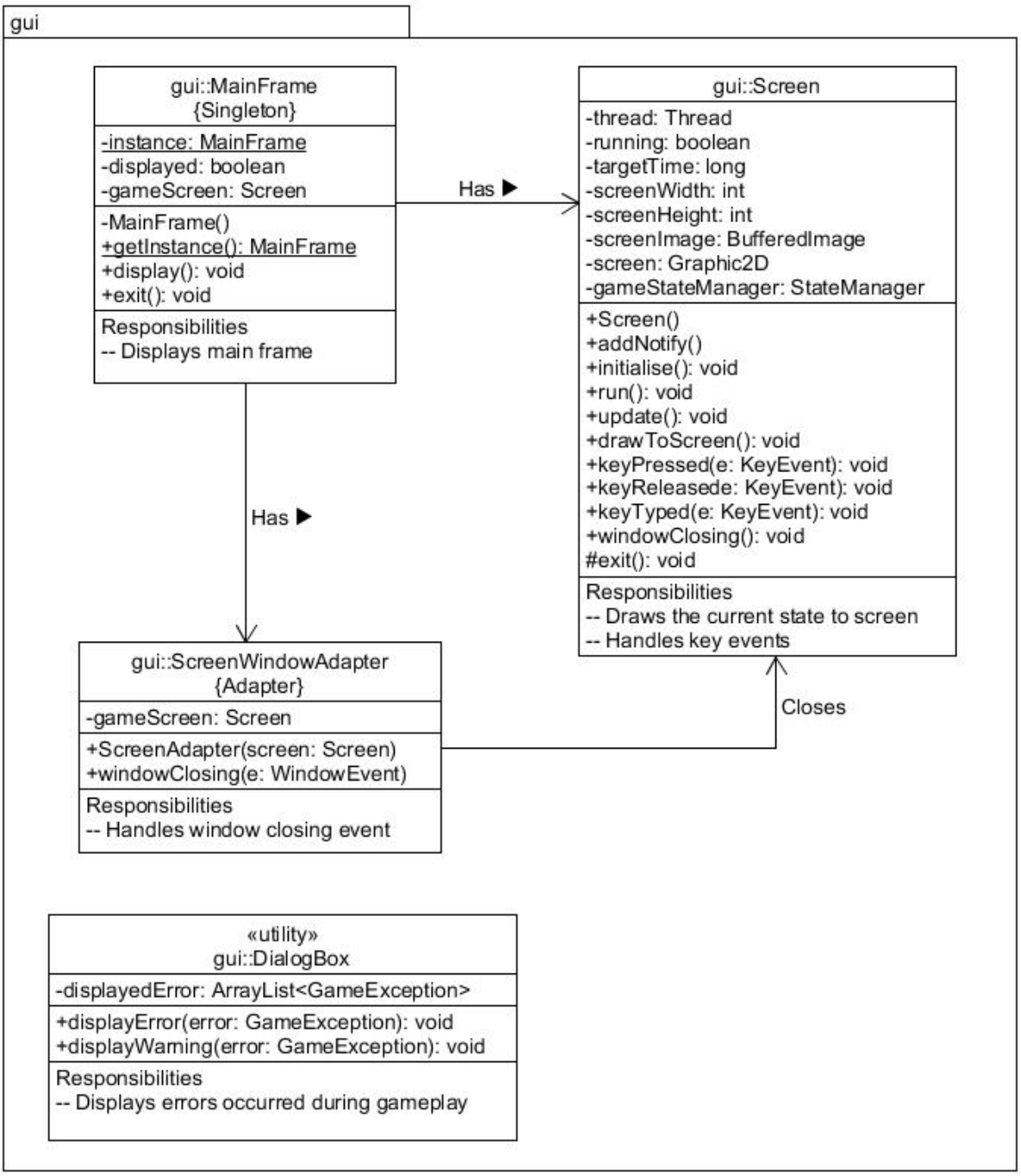
1. Description

We have two adapter classes which are used to improve the interface between classes. The key adapter is used to convert the raw user inputs in the form of key events into a more usable form. This is done by converting the key events when a key is pressed, released, or typed into a key object that makes the key pressed or released more identifiable and the value of the key typed more identifiable as well.

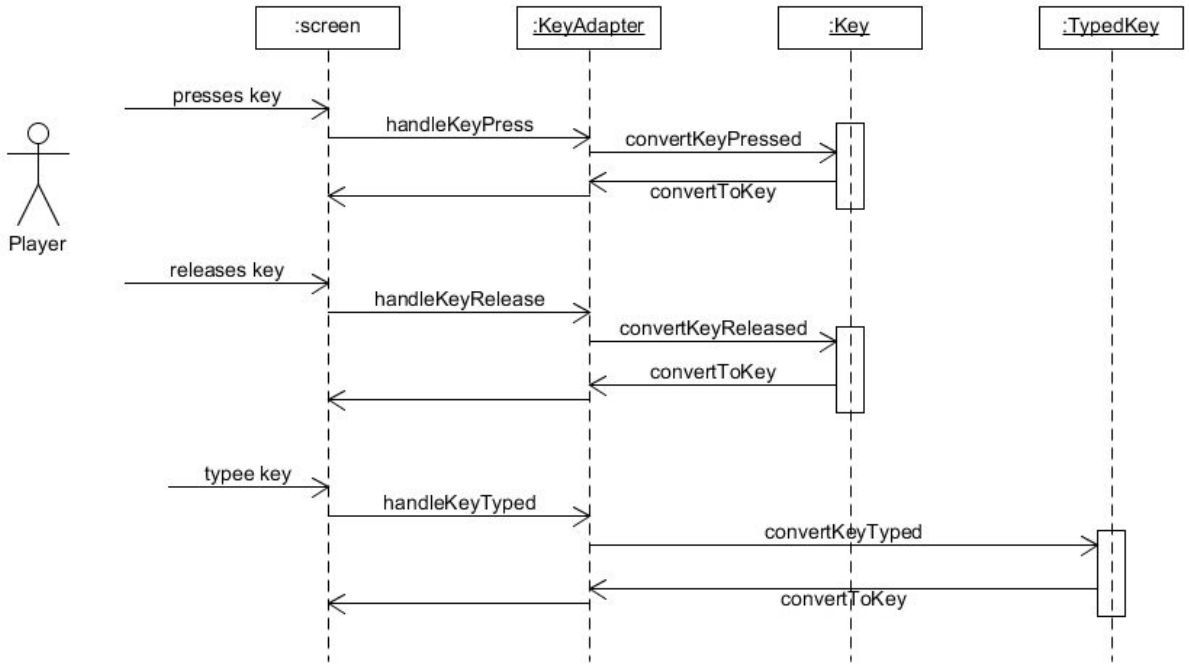
The window adapter is used to convert the raw user inputs in the form of window events into a more usable form. More specifically when the user presses the 'close' button of the window which contains the game screen, the adapter then used to notify the appropriate classes to exit the program.

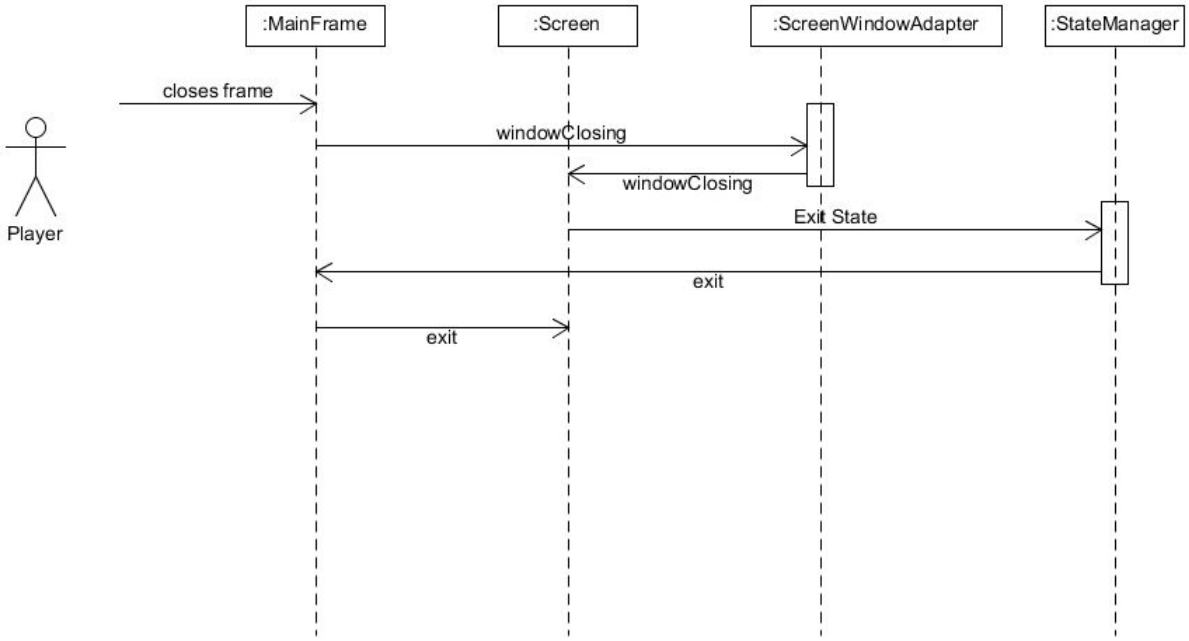
2. Class Diagram





3. Sequence Diagram



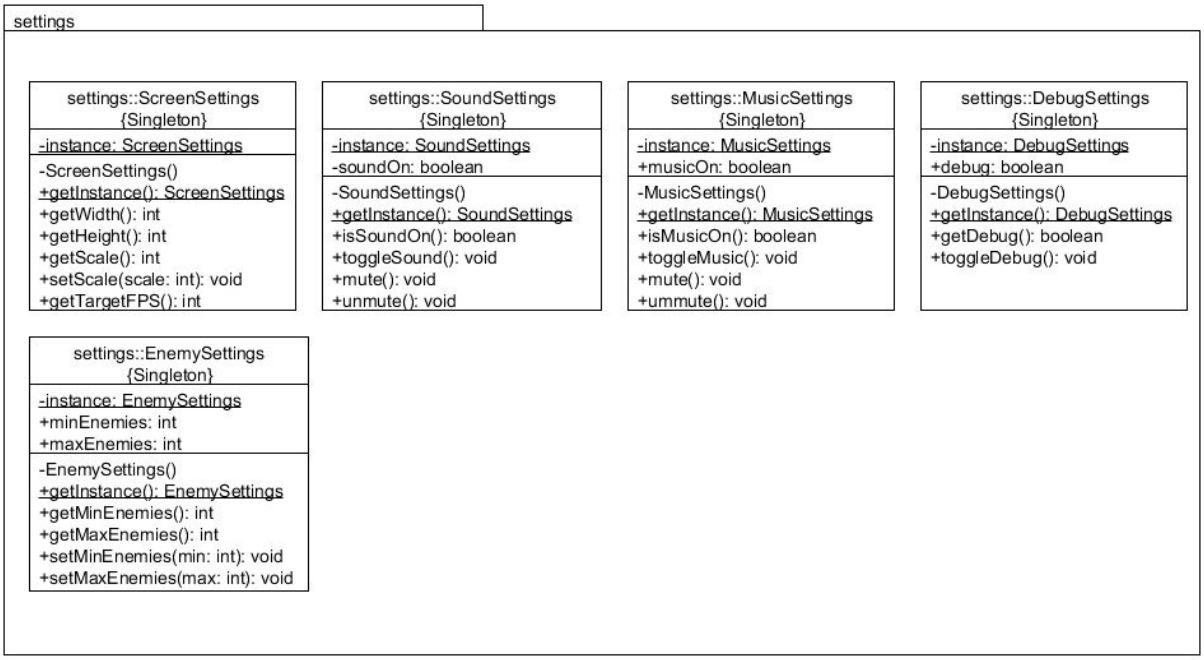


Design Pattern II - Singleton

1. Description

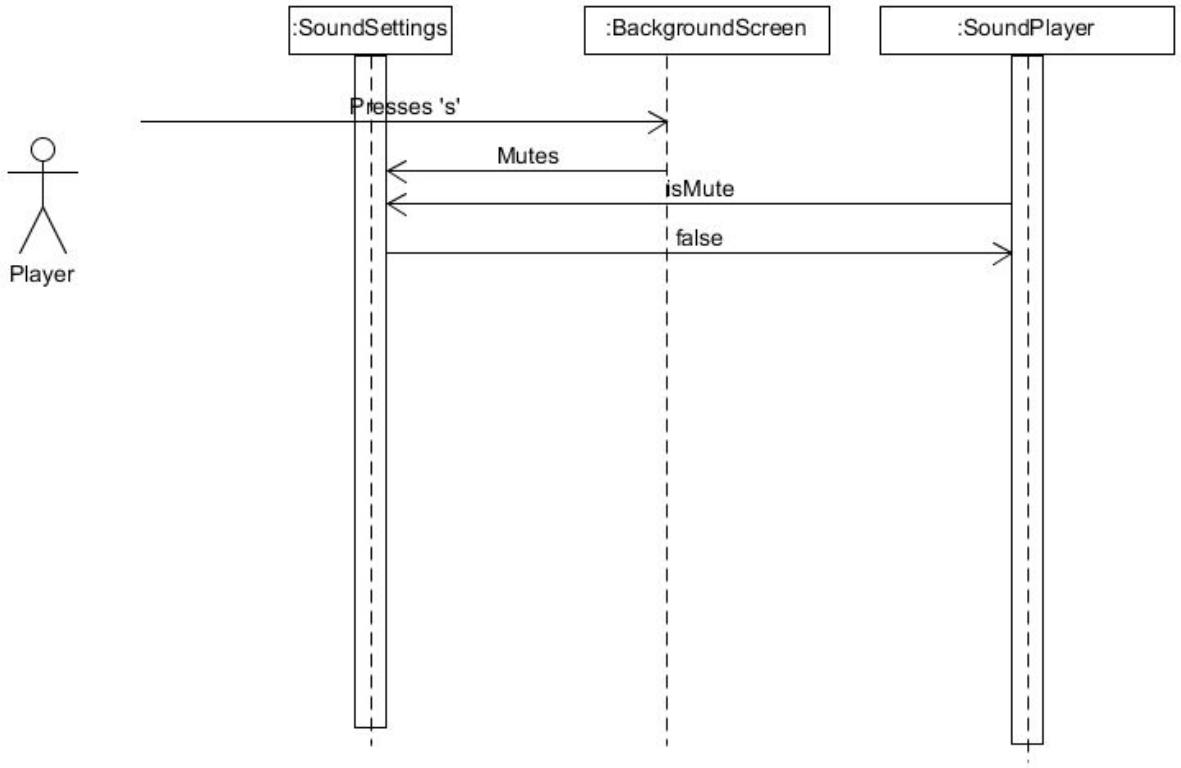
We have multiple implementations of the singleton design pattern. These implementations were mainly used when there was a single instance of an object that has to be used globally and using the singleton design pattern is better than passing the instance as an argument. The bulk of the singleton implementations are in the “settings” package which is used for the global settings of the game, of which there is always only one instance.

2. Class Diagram



3. Sequence Diagram

Once instance of a singleton in use:



This is but one instance of the a singleton is action. We also have singleton implementations for loading resources, which means that resources can be loaded once, and used globally. We also have a singleton implementation for the main frame, as there is only one instance that is used globally; this allows the main frame to be exited when necessary.

Exercise 3 - Software Engineering Economics

1) Explain how good and bad practices are recognized

The projects were analyzed on the basis of project cost and time-to-market, weighted according to project size (in function points). The benchmark for this analysis was the average weighted cost and average weighted time-to-market. Projects that performed better than average, on both cost and duration were classified as projects exhibiting "good practice", whereas projects which performed worse than average on both metrics were classified as exhibiting "bad practice".

2) Explain why Visual Basic being in the good practice group is a not so interesting finding of the study

The fact that Visual Basic was found to be in the good practice group was most likely an effect of companies hiring developers based on availability (and thereby cost), rather than choosing a programming language on the actual merits of the language. That is to say, if developers who are skilled in Visual Basic are widely available, these developers consequently become cheaper to hire, resulting in lower overall project costs. Thus, it might be more appropriate to conclude that practice of hiring widely available developers is a factor, rather than the choice of language itself.

3) Enumerate 3 other factors that could have been studied in the paper and why you think they would belong to good/bad practices

1. Location

   The location of the team might play an important role. If parts of the development are outsourced to a cheaper development team in a different country, this might introduce communication issues, which may affect the time-to-market negatively, possibly offsetting the potential cost savings of hiring cheap labor. One might expect that a team that works closely together (e.g. in the same building) would be better able to communicate than a team that is distributed over several continents in different time zones. It may be the case that the expected cost savings to be had by outsourcing to, say, India is offset by the costs resulting from a longer time-to-market. One would expect having a development team work in the same location would constitute a good practice.

2. Test Coverage

   The test coverage per project would have been an interesting factor to study. High test coverage would likely be correlated with lower defects, which one would

intuitively consider to be a good practice. However, the time-to-market might be longer (resulting in a higher project cost) as a consequence of a greater amount of code having to be written (i.e. the tests). This would present a challenge to the authors' definition of success versus failure based on cost and duration. That is to say, a high degree of test coverage may be a case where a good practice does not fall into the "good practice" quadrant as a result of the authors' narrow (i.e. cost/duration) definition of what constitutes "good".

3. Maintainability

Rather than focusing only on the period between a project's inception and its launch (i.e. its technical Go Live), it would have been interesting to also look at the various projects' maintainability. That is to say, how much time/effort/cost is involved in making changes and improvements to a given project over time? If a given project goes through several versions, as most software projects do, one would expect a well-designed piece of software to be easier to change and maintain, resulting in long-term savings. However, A system which is well-designed and well-tested might take longer to develop from initial start to Go Live. Additionally, it might also be more expensive to develop, if highly skilled (i.e. more expensive) developers are more likely to build well-designed, well-tested systems. This would mean that a project where the first version takes longer and costs more to develop than average might perform better than average during the development of the second version. So, the practice of designing maintainable/extendable systems which are well-tested may be regarded as a "bad practice" in the beginning, but may prove to be a good practice with each subsequent version. How exactly the maintainability of a piece of software would be measured, is not entirely clear. It might be possible to identify design patterns which are used in each project and to look at which of these were strongly related to cost/duration over several versions of a project.

4) Describe in detail 3 bad practice factors and why they belong to the bad practice group

1. Once-only project

A project which is meant to produce an ‘isolated’ product, where the factors used to produce the product may also be ‘isolated’. Which means that a group of people produce a product, with no intentions of maintaining it, in a particular field. This group of people might disperse once the project is completed, or might produce another project in a completely different field. Thus making the development of that project a one time thing. A once-only project may perhaps be considered the opposite of a release-based project. As release-based projects fall into the "good practices" quadrant, it may not be surprising, then, that once-only projects have a high likelihood of failure. Without the benefit of feedback from previous releases and the experience that the development

team acquires through the process of developing multiple releases, once-only projects are more likely to fail.

2. Many team changes, inexperienced team

   Multiple team changes may mean that an efficient work environment has to be established each time. Where teams need to reestablish a high level of communication that is necessary to complete the project. An inexperienced team will take longer to develop the skills, procedures, and flow of communication that an experienced team may already have. Whereas an experienced team may already be implementing a process such as Scrum and may already be using project management,development, and communication tools which help facilitate the project's progress, an inexperienced team would be faced with a learning curve, which can be costly, both financially and temporally. A team which changes frequently is faced with the same problem, as new members who are unfamiliar with each other's style of work or with the particular tools involved, would have to develop the necessary skills/knowledge before being able to fully contribute to the project.

3. Rules & regulations - driven

   A project which is rules- and regulations-driven may be required to satisfy externally imposed stipulations, which are not directly related to the functionality of the product. These additional requirements make these projects more costly and time-consuming than projects which are not faced with these additional requirements.