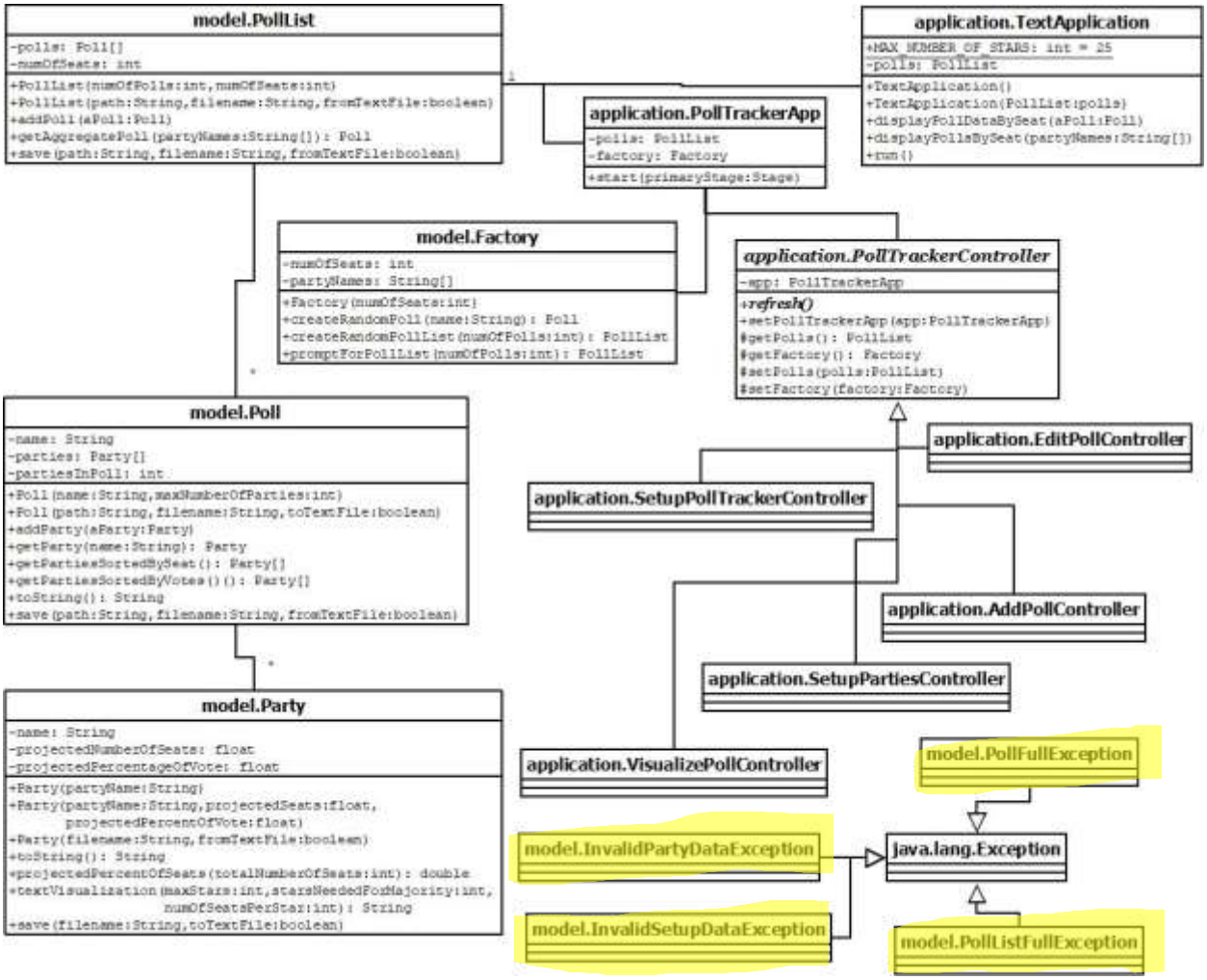
Instructor Set Project – Requirements for Iteration 3

You'll continue with the project you started for iterations 1 and 2. This means you need a correct version of all the classes from the class diagram provided for iteration 1. If there are errors in any of these classes, they will have to be fixed before starting this second iteration of the project.

For this third iteration, each team member will create a JUnit test for one of the classes and will add exception throwing and/or handling to one or more classes. The team will also create a test script.

Overview of Requirements

By the end of this iteration, all the below classes should be created. They should all compile and you should be able to run both the TextApplication class and the PollTrackerApp class. The classes new for this iteration are: InvalidPartyDataException, PollFullException, PollListFullException, InvalidSetupDataException. You will also create a JUnit test for the class Party, Poll, PollList and TextApp. These test classes are not shown in the class diagram.



Each team member should handle some of the errors that can occur when the application runs and create one test class. Test classes should be named as <Class Tested>Test. For example, PartyTest for the JUnit test class that is testing the Party class. All test classes should be placed in a package called test.

The choices for team members are:

1. Managing errors related to parties and creating a JUnit test for the Party class.
2. Managing errors related to polls and creating a JUnit test for the Poll class.
3. Managing errors related to poll lists and creating a JUnit test for the PollList class.
4. Managing errors related to setting up a poll tracker and creating a JUnit test for the TextApplication class.
5. Creating a test script and creating a JUnit test for the Factory class.

Detailed requirements for each of these are at the end of this document. In this iteration, you will all be updating the same classes. For example, error handling in the Party class also required changes to the PollList class to ensure PollList will compile with the updated Party class. Make sure that you communicate frequently about which classes you are making updates to. Make sure that you push your updated code (that compiles) to the shared repository frequently. And exercise caution when merging changes from multiple team members together to ensure none of the work is lost.

Code Quality

All code submitted will be graded on documentation (individual), legibility (individual), quality (individual), and consistency of coding styles between classes (team). See requirements for iteration 1 for additional details.

Grading

Individual grading is for your error handling code and the JUnit class/test script. In this case, others will require that you changes to classes compiles. If you add code to the repository that does not compile at any point for this iteration, you will lose marks. This reflects the extra work that you caused your team members. For the final version: **Code that does not compile or can't be run is not worth any marks and will not be graded.** (This includes if the code does not compile/run due to incorrect/missing code from iteration 1 or 2.) The code that you created will be marked as follows:

- Error Handling Functionality:
  - (1 mark) Created required Exception class with all required constructors.
  - (1 mark) When errors are detected, the exception is created and thrown as required.
  - (1 mark) Updated GUI to add required error message label and used this label to display descriptive error message when exception is caught due to errors.
  - (1 mark) All required code in entire project is updated to handle exceptions as appropriate to ensure entire project compiles with added thrown exceptions.
  - (-2 marks) Pushed code the repository that did not compile or caused compile errors in other classes which caused delays for other team members.
- Unit Testing:
  - (1 mark) Created JUnit test class that has at least one test case for each public method.
  - (1 mark) Each method is thoroughly tested.
  - (1 mark) Tests that ensure exceptions are thrown when required are also included.

- Code quality (1 mark): Quality of added code is high.
- Documentation (1 mark): Good use of javadoc and in-line documentation. Newly added code is well documented.

Team grading for the project as a whole will require that the team worked well together when multiple team members needed to update the same class. All should make significant effort to ensure the project code base compiles at all times and the code updates to the same class is merged carefully. If one team members did not manage to complete their work, we'll use similar criteria as previous iterations to grade the team.

Team grading breakdown is as follows:

- (2 marks) All classes work together and the project has all the required error handling and test classes.
- (1 mark) The team communicated well while working on the same classes, compile errors in the shared code base was (mostly) avoided throughout, code was pushed frequently and merged carefully and correctly.
- (1 mark) There is no code duplication between classes and all classes use the same coding conventions and have similar
    - Naming conventions
    - Spacing conventions
    - Placement of braces
    - Documentation conventions

Collaboration and Academic Misconduct

You are expected to **collaborate** with your team members. If you are struggling when trying to complete your class, first contact your team members for help. If you need more help than they can reasonably be asked to provide, contact your TA or instructor. If your team struggles to combine the code created by all team members, also make sure to contact your TA or instructor. Do NOT discuss the project with anyone else. Doing so will be considered **cheating** and will be brought to the attention of the Dean of Science.

You may copy code that you find elsewhere in the public domain and use it for your project. If you do so cite your source! Not citing your source is considered **plagiarism** and will be brought to the attention of the Dean of Science. So, use any code that is publicly available that you wish and indicate where you found the code. The best place to cite the source of code is by using method or in-line documentation. If your code is mostly put together by copying code from elsewhere, your mark may be reduced to reflect this.

You can find further information on D2L.

Detailed Requirements for each team member

1. Handling errors in Party data and testing the Party class

All code in the application should be updated to handle invalid data related to parties for TextApplication and the PollTrackerApp. To do so:

1. Create a class called InvalidPartyDataException that is a child class of java.lang.Exception. Make sure to provide a matching constructor in this child class for all constructors in the parent class. Each should invoke the parent constructor but does not do anything else. No other methods or instance variables need to be added.
2. Update all methods in the class Party as needed such that the method throws an InvalidPartyDataException if
    a. The projectedNumberOfSeats would be set to an invalid number. A valid number of seats must be non-negative.
    b. The projectedPercentOfVotes would be set to an invalid number. A valid number is between 0 and 1 (both inclusive).
3. Update the Factory class to catch the InvalidPartyDataException where required to ensure the Factory class compiles. In each catch statement, print a stack trace. This is for debugging purposes: the Factory class should not be generating any invalid data!
4. Update EditPollView to add a label that can contain an error message. The colour of the text should be red (or similar colour to indicate errors) and initially should contain the empty string.
5. Update EditPollController to catch all InvalidPartyDataExceptions that may be thrown when the user tries to update Party. In other words, add try-catch statements are required to ensure the code compiles. In each of the catch statements, set the text in the label created in the previous step to display an appropriate error message. (This error message may be the message contained in InvalidPartyDataException: make sure that it contains a descriptive error message!)
6. Update EditPollController to clear all error message when they no longer apply.
7. Update the PollList class to handle the InvalidPartyDataException where needed to ensure the code compiles. In this case determine if the method should be declared to throw the InvalidPartyDataException or if the exception should be caught and handled in the method itself. Make sure you document which option you choose and why.

Once the error handling code has been updated to use exceptions, create a JUnit test class that tests all public methods in the Party class.

2. Handling errors in Poll data and testing the Poll class

All code in the application should be updated to handle invalid data related to a poll in TextApplication and PollTrackerApp. To do so:

1. Create a class called PollFullException that is a child class of java.lang.Exception. Make sure to provide a matching constructor in this child class for all constructors in the parent class. Each should invoke the parent constructor but does not do anything else. No other methods or instance variables need to be added.
2. Update all methods in the class Poll as needed such that the method throws a PollFullException if there is an attempt to add a party to the poll when there is no space for additional parties in the poll.
3. Update the Factory class to catch the PollFullException where required to ensure the Factory class compiles. In each catch statement, print a stack trace. This is for debugging purposes: the Factory class should not be generating any invalid data!
4. Update controller classes as needed to ensure they compile by catching the PollFullException where required. This will depend on your team's implementation and may include EditPollController and AddPollController.
    a. If EditPollController needs to be updated to handle PollFullException:
        i. Catch the exception where required and place an error message in the label created by your team mate that is working on the Party class for this iteration.
        ii. Update EditPollController to clear all error message when they no longer apply. (This may partially be completed by your team mate.)
    b. If AddPollController needs to be updated:
        i. Update AddPollController to catch all PollFullExceptions where required. In each of the catch statements, set the text in the label created by your teammate that is working on the PollList class to display an appropriate error message.
        ii. Update EditPollController to clear all error message when they no longer apply.
5. Update the PollList class to handle the PollFullException where needed to ensure the code compiles. In this case determine if the method should be declared to throw the PollFullException or if the exception should be caught and handled in the method itself. Make sure you document which option you choose and why.

Once the error handling code has been updated to use exceptions, create a JUnit test class that tests all public methods in the Poll class.

3. Handling errors when the PollList is full and testing the PollList class

All code in the application should be updated to handle errors related to creating too many polls. To do so:

1. Create a class called PollListFullException that is a child class of java.lang.Exception. Make sure to provide a matching constructor in this child class for all constructors in the parent class. Each should invoke the parent constructor but does not do anything else. No other methods or instance variables need to be added.
2. Update all methods in the class PollList as needed such that the method throws a PollListFullException if there is an attempt to add a poll to the list when there is no space for additional polls.
3. Update the Factory class to catch the PollListFullException where required to ensure the Factory class compiles. In each catch statement, print a stack trace. This is for debugging purposes: the Factory class should not be generating any invalid data!
4. Update AddPollView to add a label that can contain an error message. The colour of the text should be red (or similar colour to indicate errors) and initially should contain the empty string.
5. Update AddPollController to catch all PollListFullExceptions that may be thrown when the user tries to add a Poll. In other words, add try-catch statements are required to ensure the code compiles. In each of the catch statements, set the text in the label created in the previous step to display an appropriate error message.
6. Update AddPollController to clear all error message when they no longer apply.
7. Update any other class in the project to handle the PollFullException where needed to ensure the code compiles. Determine if the method should be declared to throw the PollListFullException or if the exception should be caught and handled in the method itself. Make sure you document which option you choose and why.

Once the error handling code has been updated to use exceptions, create a JUnit test class that tests all public methods in the PollList class.

4. Handling remaining possible errors and testing the TextApplication class

All code in the application should be updated to handle errors related to giving invalid data when setting up the poll tracker. This includes the number of polls to create, the number of parties in the election and the number of seats available in the election. When the user is prompted for this information, make sure they enter an integer which is greater than 0. To do so:

1. Create a class called InvalidSetupDataException that is a child class of java.lang.Exception. Make sure to provide a matching constructor in this child class for all constructors in the parent class. Each should invoke the parent constructor but does not do anything else. No other methods or instance variables need to be added.
2. Update the constructor in PollList such that it throws the InvalidSetupDataException if either the number of seats or the number of polls is less than or equal to zero. Make sure to set an appropriate message that clearly indicates what the error is.
3. In the Factory class update the createRandomPollList and promptForPollList method such that they are declared to throw the InvalidSetupDataException. This will ensure the Factory class compiles. The methods should throw the exception since they should not be invoked with invalid arguments.
4. Update both TextApplication and SetupPollTrackerController to handle user errors related to the number of seats, number of parties and number of polls entered by the user. Do this by catching exceptions:
    a. Make sure that the user enters integers by catching any exceptions that occur when the user does not enter an integer.
    b. Make sure the user enters a valid number by catching the InvalidSetupDataException.
5. Provide useful error messages when invalid data is detected.
    a. In TextApplication, print a reasonable message to the console and re-prompt the user for the information.
    b. For PollTrackerApp
        i. Update SetupPollTrackerView to add a label that can contain an error message. The colour of the text should be red (or similar colour to indicate errors) and initially should contain the empty string.
        ii. Update SetupPollTrackerController to place an error message in the label created in the previous step when an exception is caught.
        iii. Update SetupPollTrackerController to clear all error message when they no longer apply.
6. Update any other class in the project to handle the InvalidSetupDataException where needed to ensure the code compiles. Determine if the method should be declared to throw the PollListFullException or if the exception should be caught and handled in the method itself. Make sure you document which option you choose and why.

Once the error handling code has been updated to use exceptions, create a JUnit test class that tests all method textVisualizationBySeat in the TextApplication class.

5. Test script and testing the Factory class

These requirements are optional for teams that have less than 5 team members.

Create a test script that can be used by a tested that has limited knowledge of the application. It should allow a tester to know how to thoroughly test the entire application, know what the expected behaviour is and what possible errors the application should be able to expect. But the script should leave some details, such as actual values to enter or order of operations to the tester. (This ensures that the tester has free reign to find unexpected bugs in the system.)

Also create a JUnit test class that tests all public methods and constructors in the Factory class. Most methods in this class return random data. The tests will have to verify that the random data returned is valid. Some validity parameters that should be considered during testing:

- The correct number of parties and party names are used.
- The correct number of polls are created.
- All seats are assigned to some party in a poll.
- The percentage of votes and number of seats 'make sense' with respect to each other.
- The total number of seats is not more than the total number of available seats.
- Etc.

The grading for this component will be slightly different than for the other four components. Instead of marks for the Error Handling Functionality, there will be marks for the Test Script as follows:

- Test script quality:
    - (1 mark) There is at least one script that walks through an expected and correct use of each tab in the application.
    - (1 mark) There is at least one script for each possible error related to use input and what the expected behaviour is on those errors.
    - (1 mark) The tester is encouraged through the script to find other bugs in the application.
    - (1 mark) The script is easy to read. There are no grammatical or spelling errors and the document is well organized. There should be screen shots that illustrate the text to help understand was is expected.