

On December 4, as stated in the syllabus, you are to turn in your CD with code and documentation. While the documentation between ILM's may vary, in what they turn in, everyone should include a Requirements Specification, Project Plan, Detailed Design Documentation, and a testing Plan and Results. It is likely there are many different interpretations of what is required in such documents. In the syllabus, the documents are defined. This document represents some ideas that Dr. Allan pulled from the web. It is hoped that these will give you a better idea of what is required. You may have better ideas from your software engineering classes. This list is not intended to be binding, but merely to give you ideas of what you might want to include.

1. A **project plan**, according to the [Project Management Body of Knowledge](#), is "...a formal, approved document used to guide both *project execution* and *project control*. The primary uses of the project plan are to document planning assumptions and decisions, facilitate communication among *stakeholders*, and document approved scope, cost, and schedule *baselines*. A project plan may be summary or detailed

At a minimum, a project plan answers basic questions about the project:

1. **Why?** - What is the problem or value proposition addressed by the project? Why is it being sponsored?
2. **What?** - What is the work that will be performed on the project? What are the major products/deliverables?
3. **Who?** - Who will be involved and what will be their responsibilities within the project? How will they be organized?
4. **When?** - What is the project timeline and when will particularly meaningful points, referred to as milestones, be complete?

To be a complete project plan, the project plan must also describe the execution, management and control of the project. This information can be provided by referencing other documents that will be produced or it may be detailed in the project plan itself.

2. A **Software Requirements Specification (SRS)** is a complete description of the behavior of the system to be developed. It includes a set of use cases that describe all the interactions the users will have with the software.

A **use case** in software engineering and systems engineering is a description of a system's behavior as it responds to a request that originates from outside of that system. The use case technique is used to capture a system's behavioral requirements by detailing scenario-driven threads through the functional requirements. Each use case focuses on describing how to achieve a goal or task.

I think the idea of a storyboard is appropriate here – as we envision a sequence of steps the user performs in interacting with the software. An example of a storyboard is at the end of this document. The storyboard gives a mockup of the user interface. This seems like an important detail to convey and get approved before one starts to code.

3. Testing Plan:

Decide the specific criteria that each segment of the system/subsystem must meet. Such criteria are described by the user of the system/subsystem and typically are a mix of functional and performance

requirements, such as processing data within a certain time frame, producing a report, or responding to an online query within a certain amount of time. It may also include being able to run the code on the web or from a variety of platforms or browsers.

Identify the testing tools to be used during the preparation for and execution of the test.

Indicate whether the testing will use the normal input and database or whether some special test input is to be used.

Indicate the anticipated limitations imposed on the testing because of system or test conditions (timing, interfaces, equipment, personnel).

Discuss the range over which a data output value or a system performance parameter can vary and still be considered acceptable.

Provide a detailed list of the system and communications functions to be tested.

Indicate whether the test is to be controlled by manual, semiautomatic, or automatic means

Describe the manner in which input data are controlled in order to test the system with a minimum number of data types and values, exercise the system with a range of bona fide data types and values that test for overload, saturation, and other “worst case” effects, and exercise the system with bogus data and values that test for rejection of irregular input.

White box testing (a.k.a. clear box testing, glass box testing, transparent box testing, translucent box testing or structural testing) uses an internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all paths through the software.

Black box testing takes an external perspective of the test object to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid inputs and determines the correct output. There is no knowledge of the test object's internal structure. Often the set of tests are decided before the software is even written.

The primary goal of **unit testing** is to take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect. Each unit is tested separately before integrating them into modules to test the interfaces between modules. Unit testing has proven its value in that a large percentage of defects are identified during its use.

If you have two units and decide it would be more cost effective to glue them together and initially test them as an integrated unit, an error could occur in a variety of places:

- Is the error due to a defect in unit 1?
- Is the error due to a defect in unit 2?
- Is the error due to defects in both units?
- Is the error due to a defect in the interface between the units?
- Is the error due to a defect in the test?

Integration testing is a logical extension of unit testing. In its simplest form, two units that have already been tested are combined into a component and the interface between them is tested. A component, in

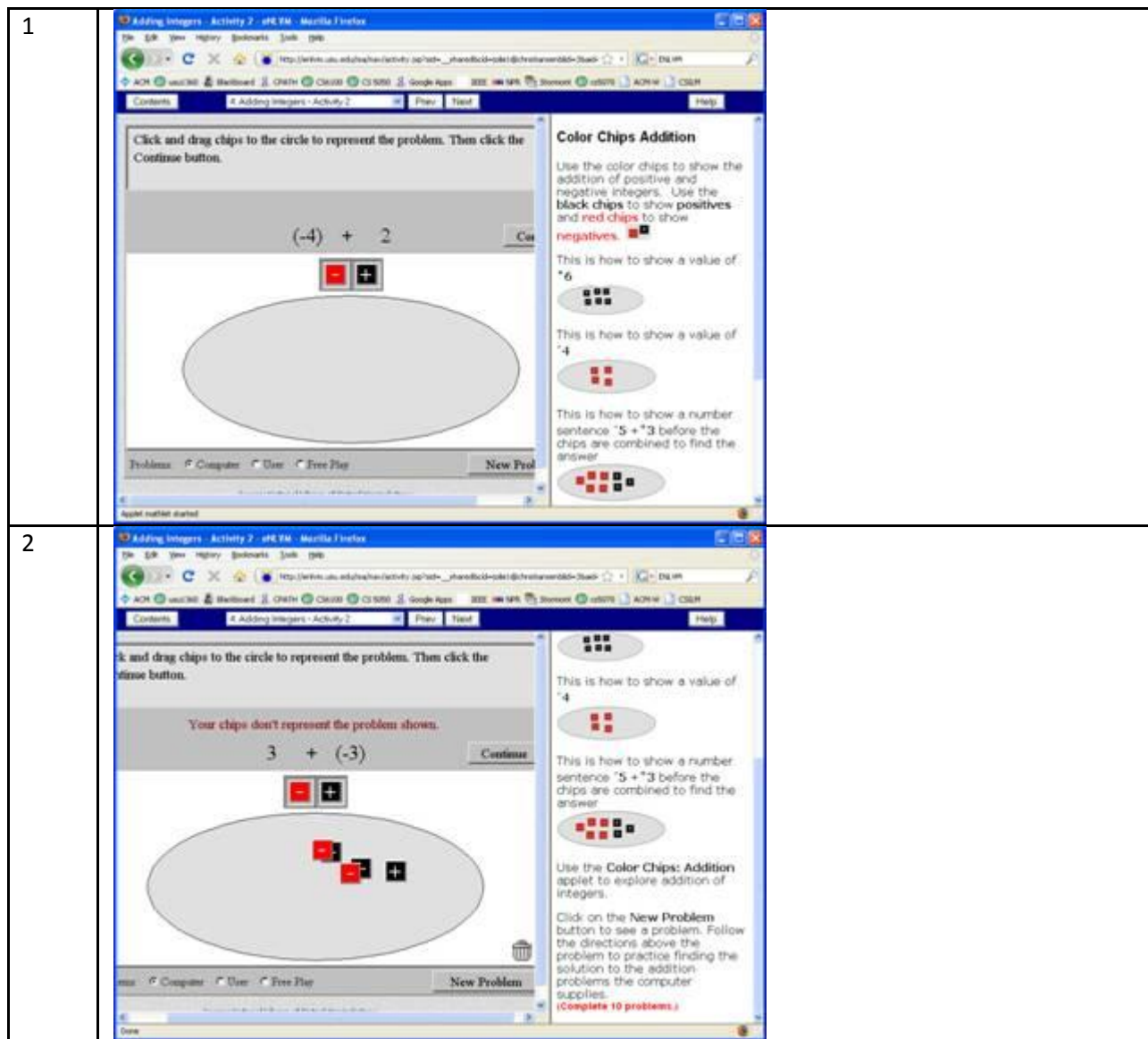
this sense, refers to an integrated aggregate of more than one unit. In a realistic scenario, many units are combined into components, which are in turn aggregated into even larger parts of the program.

Any time you modify an implementation within a program, you should also do **regression testing**. You can do so by rerunning existing tests against the modified code to determine whether the changes break anything that worked prior to the change and by writing new tests where necessary.

Storyboard

A storyboard will show:

1. What the user interface looks like
2. What steps the user will go through. A math ILM example of a storyboard follows. An explanation of what is happening in each case would be a good addition.



3

Enter a problem to solve. Click the button between the number fields to change the operation.

Entering your own problems

To enter your own problems, you can do the following:

1. Click on the **User** button.

- Type in the values then click the **Continue** button.
- Drag chips to represent the problem.
- Drag black chips into the circle to represent the positive numbers
- Drag red chips into the circle to represent the

4

Simplify by dragging minus chips onto plus chips.

5 + (-6)

2. This will give you 2 blank boxes to fill in. You will enter 2 and 2.

- Type in the values then click the **Continue** button.
- Drag chips to represent the problem.
- Drag black chips into the circle to represent the positive numbers
- Drag red chips into the circle to represent the negative numbers then click **Continue**.

3. Drag the positive chips on the negative chips to

5

Click and drag chips onto the workspace to represent and solve problems, and drag minus chips onto plus chips to cancel them out.

Sum = 0

Solving Addition Problems

Use the Color Chips applet to help solve problems related to addition. Be sure to click on the **Free Play** button before you begin the problem.

You will be given problems such as the following:

1. The workspace starts empty. Color Chips are dragged into the circle. After the problem has been simplified by removing zero pairs, 4 black chips are left.

There are many possibilities for the number of chips that could have been in the circle originally. For example, 5 black chips and 1 red chip could have been dragged into the circle to start the problem. Then 1 red chip and 1 black chip could be

Problem Computer Disc Free Play New Problem