Tutorial 4: Arcade Menu

Thomas Chapman

# Specification

This can vary in size, but it describes in plain English, perhaps with diagrams, what it you are trying to do. It does NOT discuss how. For a simple tutorial assignment this might just be a copy and pasted sentence from the tutorial. For a complete game project this could be pages of notes – a full game design document.

Develop a menu for an arcade game.

# Technical design

This is a technical language and technical diagram heavy explanation of HOW you are going to implement the specification. For a simple tutorial assignment, you want one or two appropriate diagrams or algorithms (flow chart, pseudocode, UML diagrams(level5), etc.). There are lots of little tutorials, so we don’t want something excessive for each. Over all your apps/projects we should be able to see a couple of examples of all the technical design approaches/diagrams we’ve covered. If this was a big game project however, then this would span multiple pages with descriptions of code, diagrams, pseudocode, etc.

This is the pseudocode for the arcade game menu, it can handle user inputs, balance, and game fees: Test

controlGameApplication

**local option, balance**

call payInitialFee**(balance)**

call enterOption**(option)**

while (not(option = 'Q') and (balance>0))

call processOption**(option, balance)**

call enterOption**(option)**

endwhile

output("Thanks for playing")

call showBalance(balance)

proc payInitialFee**(OUT: balance)**

set balance to 100

endproc

proc enterOption**(OUT: option)**

output("Enter option (P:play or B:balance or Q: quit> ")

input(option)

call putInUppercase(**option)**

endproc

proc putInUppercase(**IN/OUT: option)**

set option to uppercase(option)

endproc

proc processOption**(IN: option, IN/OUT: balance)**

if(option = 'P') then

call playGame**(balance)**

else

if(option = 'B') then

call showBalance**(balance)**

else

output(“ERROR: Invalid Command!”)

endif

endif

endproc

proc playGame**(IN/OUT: balance)**

output(“Playing...”)

set balance to balance – 20

//...

endproc

proc showBalance**(IN: balance)**

output(“The current balance is £”, balance)

endproc

# Test plan

How do you know when to stop, how do you know it works, is it working appropriately (fun), etc. Again, for a small tutorial assignment, this would probably the classic basic table of test input and what you expect it to output. For larger programs this could include instructions on different tasks the user is meant to perform, and how they are meant to work – imagine you were writing instructions for a test department with staff who don’t know you or the code. For a big game project it would include instructions on what the player is expected to feel, is this bit meant to be easy, are they supposed to be confused, is this bit meant to be hard, how long is it meant to take to play this bit, etc. Then you play test, observe and test. For big projects you’d use an issue tracker (bitbucket) to record information about all these things and figure out which are important and need fixing and which aren’t (it’s not complex, just a shared online repository of bugs with priorities). It’s part of the classic software development lifecycle, design->implement->test->repeat until you get something really good. Fail fast, fail early. Find the optimum solution. For a large project you’d include user testing, where you watch someone play and make notes about what they liked, didn’t like, where they got stuck, if something broke, balance issues – things too easy or hard, things too confusing, etc.

# GiT commit log

All work should be kept on GiT, bitbucket and github are free to use. Make sure the repository is marked private or people will google the code and find it. A screen shot of the git commit log will suffice, it needs to show who did what and when. At level 4 it will take a while to learn to use GiT, but we will eventually.

# Schedule

Apply some common sense, if it’s a simple tutorial item, if it’s small, then a basic bullet point list is fine. So, tutorial assessment, Joe wants to get it finished for the week after next session. Sets aside 4hrs: 0.25hr spec, 0.5hr design, 3hr implement, 0.25min test, 1hr slack. See how long it really takes, next time adjust accordingly.

|  |  |  |
| --- | --- | --- |
| **Task** | **Estimated Hrs** | **Actual Hrs** |
| Spec | 0.25 | 0.25 |
| Design | 0.5 | 0.25 |
| Implement | 3 | 2.5 |
| Debug and test | 0.25 | 3.5 |
| Slack | 1 | 0 |
| **total** | **5** | **6.5** |

If it’s a big project that’s going to be more like 10+ hours, then you need Trello. Schedules can be done in Trello and screen shots used, refer to the notes on Trello, you need to use the ‘Plus’ plugin for Chrome so you can estimate how long things will take, this is how you guesstimate when the project will be finished – this is absolutely critically important to the people paying your wages. Think of all the tasks and put them on ‘cards’, shuffle them around, keep breaking them down into smaller tasks on more cards, until you’ve written 1 or 2 hours on each card. Trello will tell you how long the whole thing will take. As you work, mark off the tasks as done and how much time they REALLY took. It takes effort, but is the only way to get better at task estimation. It’s important because YOU are the biggest cost to your employer.