General comments

Requirements
Most teams gave a clear description of the requirements template that they were using, but several neglected to justify the format (or the changes that they'd made to it).

The requirements engineering process was not clearly described in most submissions: what was missing was (usually) a clear description of what "analysis" took place, how negotiation with the customer or within the team was supported, and what happened when any inconsistencies were identified. Several teams described excellent practices: prototyping, interviews, questionnaires.

The requirements specifications themselves were done very well in a few cases: precise requirements (that followed a template strictly) linked to excellent justification and the risk assessment. Typical problems included: design decisions expressed in the requirements (e.g., decisions about how the system would work); vague nonfunctional requirements; some confusion between constraints and nonfunctional requirements; and requirements that were too large (and hence very difficult to validate or test). Some excellent solutions presented "fit" criteria for their requirements, thus addressing concerns about validity.

Architecture
There were a few very good architecture reports, including some with a very abstract level of detail (which usually made the report easier to write and do well on). At conceptual level, you should be able to justify (almost) everything in your architecture by reference to the scenario and your requirements derived directly from the scenario.

The main issues were related to trying to put in too much detail - we are not looking for the best model or the longest lists, but at the most convincing and rounded solution. It was not sufficient to simply draw and describe a UML class diagram.

Some teams explicitly committed to OO (e.g. UML and/or Java); some explicitly committed to Unity, and some did not explicitly commit to any platform: all three are fine, but there does need to be an understanding that, if you commit to a paradigm for modelling, you need to understand how to map from that to the target implementation. Class diagrams do NOT map to Unity, but they do map to Java. It is good practice to consider and state whether you are committing to a particular platform when writing the conceptual model, as it helps the reader understand any unstated assumptions in your modelling (it's also good practice, but practically unachievable, to record all the assumptions you are making!).

Generally, a simple class diagram of key concepts for the game was sufficient - or a general model of the behaviour if not aiming at an OO implementation. Some teams used flowcharts or sequence charts effectively for behavioural/unity architectures. Unless there is a really clear

explanation of why they are relevant, it is not normally useful to include behavioural models alongside class models in the architecture.

Method selection and planning
Some very well researched method sections that thought through what the project and team required: it might even be that some teams had taken on board the comments on previous method sections, on the VLE.

Some people started with an essay on different models and methods. In a software engineering exam that would be appropriate, but here it is not ideal. Team feedback explains this. It is NOT appropriate to refer to the waterfall model as a method (check the first lecture if you are not sure why!).

For tools, there were generally clear statements of what was used, and a little on why the chosen tools were needed. A more thorough engineering-style report could have started by identifying what sort of things needed tool support, note the team needs, and then indicate the choice of tool that meets these criteria.

The structure of the question, and the precise wording of the question, were not reflected in many answers.

It is worth noting that, whereas there were some very convincing accounts of how Scrum (in particular) influenced the team working, the methods' use was not always obvious in the other deliverables.

Risks
Most teams provided a sufficiently detailed description of their risk registry, but only a few teams discussed the process through which risks were identified (e.g. brainstorming sessions, deduplication/merging or overlapping risks, filtering of extremely unlikely risks etc.). Many teams made good use of color coding and assigned each requirement a unique identifier, which is good practice for traceability/cross-referencing purposes. Some of the teams categorised risks according to the aspect of the project they are relevant to (e.g. management, product, client), and listed risks in descending order of severity which was useful.

There was some confusion between the concepts of "prevention" and "mitigation". In general, prevention reduces the probability of a risk materialising while mitigation reduces the damage done once a risk has already materialised. Most teams did not allocate risks to particular members of the team (e.g. project manager, client interface). This is not ideal as risks owned by everyone can effectively become risks owned by no-one.

Testing

Most teams presented clear descriptions of the technical side of testing, with a particularly high quality of information given on how their unit tests were conducted. By and large, teams also

showed how their testing mapped on to their requirements, with a large number of traceability matrices presented in their reports. Some teams even went further than this, and provided a second break-down of their requirements, and how they were fulfilled by testing, which was very clear. Furthermore, URLs to appropriate testing evidence were given by most teams.

However, outside the world of unit testing, a similar level of clarity was often not expressed in the assessment. Lots of teams mentioned that they had conducted ‘black box testing’, ‘manual testing’, or ‘play testing’, but did not discuss in sufficient depth what this had actually entailed, with ‘play testing’ being a particular culprit for this lack of detail. This is troublesome, as if detail is not given, there is no way for the marker to know if play testing consisted of one designer ‘running through’ the program and noting down if bugs occurred, or something much more rigorous.

Furthermore, few groups sufficiently justified the appropriateness of the testing strategy that they were following. Whilst many groups highlighted *in general* what different kinds of tests were , and made broad statements about their use, very few described why the specific tests that they used were appropriate to their specific circumstances. Even fewer supported their arguments with more than a cursory nod to the literature. Indeed, citations were roundly underutilised throughout the assessment. Whilst most groups included links to the resources that they were using (e.g. jUnit), or a general reference to a technique (e.g. smoke testing), it was rare to see any reference to a position piece advocating a best use policy. This, however, could go a long way to motivating your designs.

Implementation

Code was generally quite good. Most teams had a sensible decomposition into classes. A few teams had two or three very large classes taking responsibility for much of the functionality; these could have been further decomposed. Some teams with sensible decompositions had a few methods that were very large; these could be decomposed further too. Quite a few teams had hard-coded lists (e.g., of resources, of NPCs) that should have been handled in a more abstract way, making it easier to extend or change for assessment 4 (e.g., by using Factories).

Commenting was a mixed bag: two or three teams commented beautifully, focusing on the intention behind the code, and showing some strong editorial control over the commenting process. Other teams had some classes commented well, others not so well (or not at all). Several teams had extensive commenting but upon reading it it became clear that the comments were largely impenetrable to anyone outside the team, and thus some revisions would be necessary.

The implementation reports were generally done very well. A recurring issue was inconsistent tracing to requirements. Another issue in a few reports was presenting the list of features systematically, e.g., listing “features not implemented” followed by “features partially implemented”. In some cases it wasn’t clear where certain features fit.

GUI Report

GUI reports were done quite well overall. All teams described their menu structure and HUD effectively, with some good tracing to requirements. Better solutions also discussed the principles behind their GUI design (and defined their terminology). The strongest solutions talked about interaction design (i.e., how the players would interact with the GUI) as well.

Website

Websites were done uniformly well. The better solutions had given some thought to future extensions (i.e., how information across assessments should be separated out).