Software Engineer Practical Assessment

This is the program scenario:

- You work as a Junior Software Engineer.
- You are on the team responsible for developing the Sims series.
- The Sims 4 has already been released, but your team is still issuing updates which add content to the game and squash newly discovered bugs.
- **You are responsible for thinking of and building a new feature for the game.**

A **Feature Proposal** is a document which describes a new feature you'd like to add to an application. When you have a great idea, you need some way to communicate it to your team. Feature proposals provide a handy format for conveying these ideas and recording them for posterity.

And how are they structured? Click next to find out...

- **Title**: An impactful name for your feature.
- **Overview:** A brief description of your feature.
- **Value statement**: An explanation of why it would be a good idea to add your feature to the game.
- **Solution**: A description of how your feature would integrate with existing game mechanics. Oftentimes good features build on existing systems in some way, though they certainly don't have to.
- **Evaluation statement**: Provides a look at the pros and cons of adding your feature to the game. Includes a description of the possible benefits and risks, and addresses potential counterarguments to your proposal.

Example:

Diving for Underwater Objects

Overview:

Iterations of The Sims already include a variety of ways characters can interact with water, from fishing to swimming. Adding a diving component to the game would provide an entirely new way for players to move through the world. Characters could dive underwater to search for crafting materials or catch underwater sea creatures.

Explanation: Notice how the overview is concise and to the point. The goal here is to articulate your idea clearly without getting bogged down in the verbiage. This section represents the "what" of the idea.

Value Statement:

Adding such a feature would introduce a new source of crafting materials and provide players with a fresh way to catch fish. Characters could approach and dive in natural bodies of water, unlocking new areas to explore.

Explanation: here we're trying to explain "why" it would be a good idea to implement the new feature.

Solution:

The camera would follow player-controlled characters, allowing them to see points of interest (like fish and crafting materials) underwater. The player could steer their Sim toward these items and swim forward to collect them, or leave their Sim to act autonomously. Many components of the diving feature could be built on top of existing gameplay systems. The ability to dive for fish, for example, would share some implementation aspects with the existing fishing system, and the ability to collect materials would overlap with other gameplay mechanics like foraging and crafting. The overall diving mechanic would be implemented as a new way to interact with bodies of water. The diving mechanic might also share some qualities with the existing swimming mechanic, although the diving mechanic might diverge significantly. For example, movement, camera behavior, and object interaction would likely be different for diving and fishing.

Explanation: This section includes the "how" of the idea - in what ways could existing game systems be extended to accommodate the new idea?

Evaluation Statement:

Moving forward with this feature would require a deeper look into how the mechanic would interact with existing game features to ensure the cost is offset by potential player benefits. If implementation proves reasonable based on existing game systems, this feature could provide a new avenue for players to interact with the game world, contributing to The Sims' aim of creating an expansive virtual world.

Explanation: This section gives you an opportunity to rebut any arguments you foresee against your idea. If the feature comes with caveats, or if there are points of contention that need to be addressed, this is the place to do it.

CG Proposal One:

Title: "Paws & Professions: Pet Careers"

Overview:

In The Sims 4, players have enjoyed the companionship of pets, but their role has been largely passive. The " Paws & Professions: Pet Careers " feature revolutionizes this by introducing career options for pets. This feature allows players' pets to engage in various careers, such as therapy animals, show pets, rescue animals, or even social media stars. Each career path offers unique activities, challenges, and rewards, further integrating pets into the Sims' daily lives and careers.

Value Statement:

Introducing careers for pets in The Sims 4 would add a new dimension of gameplay, deepening the emotional bond between Sims and their pets. It offers players more ways to interact with their pets, new goals to achieve, and unique rewards to unlock. This feature would not only enhance the pets' role in the game but also provide players with novel experiences and storylines, increasing engagement and replayability. Additionally, it could educate players about the diverse roles animals play in our society.

Solution:

The implementation of "Paws & Professions: Pet Careers" would utilize existing pet training and care mechanics. Each career path would have specific activities and milestones. For example, a therapy pet might require training in obedience and empathy, culminating in visits to hospitals or schools. A show pet might participate in competitions, requiring training in agility and tricks. Integrating these careers would leverage the existing relationship and skill-building systems, while adding new interactions and rewards specific to each career. The feature would also introduce new NPCs and locations, such as pet training centres or competition venues.

Evaluation Statement:

While this feature offers exciting new gameplay possibilities, it must be balanced to ensure it doesn't overshadow or overcomplicate the core human-centric gameplay of "The Sims 4." Careful attention should be given to the realism and ethics of pet careers. Benefits of this feature include increased player engagement and the potential to teach responsible pet ownership and empathy. Potential drawbacks include the complexity of implementing diverse career paths and ensuring they fit seamlessly into the existing game structure. Overall, "Paws & Professions: Pet Careers" promises to enrich the Sims universe by giving pets more active and meaningful roles in their Sims' lives.

CG Proposal Two:

Title: "SimReality: Virtual Gaming Experience"

Overview:

"SimReality" introduces a groundbreaking Virtual Reality (VR) gaming system within "The Sims 4," offering a unique meta-gaming layer. In this feature, Sims can don VR headsets and engage in various virtual games, ranging from adventurous quests to relaxing simulations. This VR system not only adds a new form of entertainment for Sims but also opens up a range of creative possibilities for players to explore virtual worlds within their Sims' world.

Value Statement:

Incorporating a VR gaming system enhances the depth and diversity of activities available to Sims, providing players with fresh, innovative content. It adds a layer of futuristic technology that reflects real-world advancements in gaming, thereby keeping the game relevant and engaging. This feature allows for the exploration of imaginative scenarios, potentially including cross-genre experiences, and enhances the game's appeal to a broader audience. Furthermore, it offers educational prospects, such as VR experiences that teach Sims new skills or simulate historical events.

Solution:

SimReality would integrate with the existing technology and skill systems in "The Sims 4." Sims can purchase a VR headset and access a virtual gaming console, which offers a range of games. Each virtual game would be a mini-game with unique challenges and rewards. For example, a VR adventure game could improve a Sim's problem-solving skills, while a VR meditation experience might boost their mental well-being. The feature can be designed to have minimal or elaborate interactions based on the player's preference, allowing for either a quick, casual gaming experience or a more immersive involvement.

Evaluation Statement:

While introducing VR gaming into "The Sims 4" offers exciting new gameplay possibilities, it's important to consider its impact on the game's overall feel and complexity. The design should ensure that VR gaming complements, rather than overshadows, the existing game mechanics. Benefits include increased player engagement through innovative, interactive content and the potential for educational and skill-building scenarios. However, challenges lie in ensuring the mini-games are diverse and enjoyable without being overly complex or resource-intensive. Overall, "SimReality" promises to enrich the Sims experience by blending the virtual and simulated worlds in a fun and imaginative way.

Task number 2: Task Overview What you'll learn What a class diagram is and how creating one saves your team time How to build a class diagram using a tool like Miro What you'll do Build a class diagram for your feature from the previous task

Success!

Your proposal was a hit! Everyone on your team is excited to include the new feature in the next Sims 4 update. Your project manager has split the rather large job of developing the entire feature into several more granular coding tasks (tickets) that can be accomplished by individual developers.

You've been given the lion's share of tasks, since you were the one who proposed the feature in the first place which is an awesome opportunity! Time to get started on one of these tickets...

Object Oriented Programming

Recall that C++ is an **object oriented programming** (OOP) language, which means a system is articulated as a collection of **objects** which interact with one another. Each object has a collection of **instance variables** which make up its state, as well as a series of **methods**, which describe its behavior.

The primary alternative to OOP is **Functional Programming**, which articulates a system as a collection of functions which call one another. The extra layer of abstraction provided by OOP in the form of objects makes it easier to reason about the system as a whole.

And So it Begins

You sit down to get started on one of your new tickets - it's one of the smaller ones, and a good place to begin. You'll be able to get into the groove on this one, and then move on to tackling some of the larger tickets afterward.

The first order of business is to figure out how to break the behavior described in the ticket into a collection of objects. Once you have an idea of what sort of objects you're going to be working with and how they relate, you can move on to their implementation.

In this case, you figure it would be helpful to use a **Class Diagram** to visualize the subsystem. This will help you reason about how the objects are related to one another, and help you catch caveats and edge cases you might otherwise miss.

So, what's a class diagram? Click next to find out.

Class Diagrams

A class diagram is a document which visualizes the relationships between objects in an application. It's an excellent way to outline a complex system, giving you a feel for its structure before you begin implementing it.

By drawing out your system, you are forced to come up with clear relationships between your classes. You can easily catch edge cases and messy architectures before you begin implementing them. It may take a little longer to begin with, but saves you time backtracking in the future.

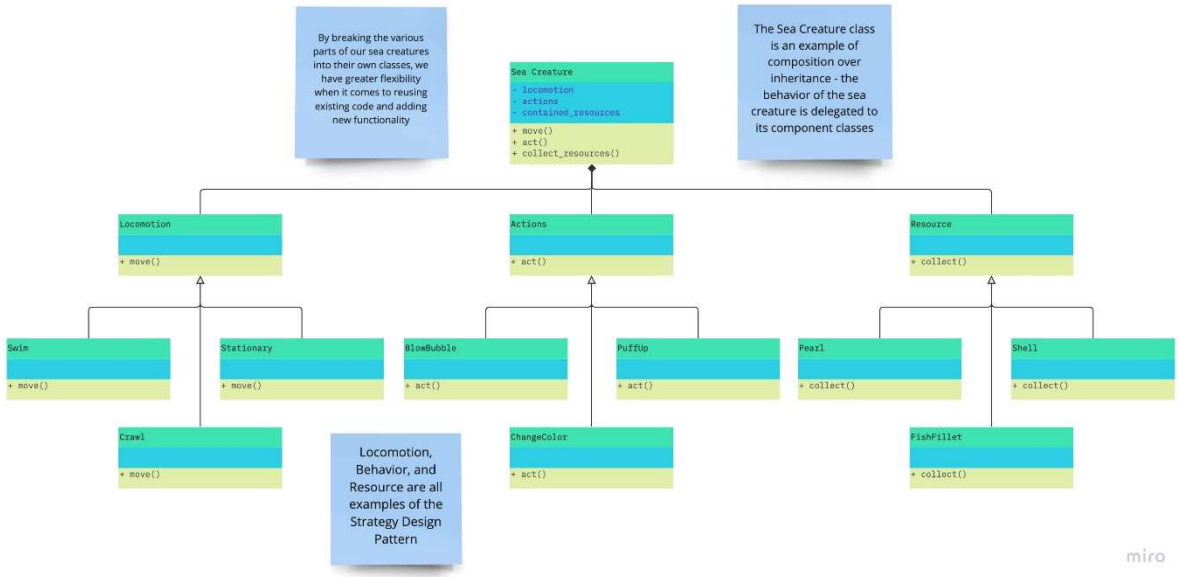
When Drafting Class Diagrams

Remember that the point of a class diagram is to get a feel for the high level structure of your system.

Don't get bogged down in the details - you should spend your time thinking about the high level relationships between classes - how they interface with one another, how they inherit from one another, how they are composed into more abstract objects.

At this point, we are thinking about the architecture of the system - how it should be broadly organized. You should lean on `design patterns` and software engineering best practices (like the `SOLID` principles) for assistance in organizing your system.

Before you build your class diagram for your feature, we'll show you an example first...



Before you start your own, here's an example on how to build a class diagram. **Have a look and then go next to get started on your own diagram for your new feature.**

Time to write your class diagram!

Now let's get to work....

Come up with a series of 5-15 objects that the feature from your proposal would use. These objects by no means have to capture the entire system, feel free to focus on one part of the feature. Then visualize these objects using a class diagram.

Be sure to include the relationship between each object, and do some thinking about the state each one would need to track, as well as the behaviors each one would need to expose. You may use any program you'd like to draft up your class diagram, but if you're at a loss, consider using a tool like Miro.

Once you're finished with your class diagram, convert your file to a pdf and upload it below. Then it will be time for Task 3!

See GitHub for Solutions and Diagrams by Conor Griffin

Task 3

With Class Diagram In Hand...

It's time to write some actual code! All of the objects you outlined in the previous step need to be implemented in C++, a high performance coding language.

Game developers often use systems programming languages like C++ instead of higher level languages like Python and JavaScript, since their code needs to be highly optimized.

Games are usually quite performance-intensive, pushing users' hardware to the limit to render 3D graphics at a reasonable frame-rate. Some languages are inherently more performant than others for a variety of reasons (for example, compiled languages tend to beat interpreted languages in terms of speed), so game developers gravitate towards those.

Having considered all of this, you spin up your IDE of choice and get to work.

Header Files

The first step to creating an object is, of course, the definition of a **class**.

In C++, this is often carried out using a **header file** - the header defines what the class does, without describing how it works. Each header has an associated **source file**, which contains the actual guts of the code.

Header files include **instance variables** and **method declarations** (method stubs with a name, list of parameters, and return type), while source files include **method implementations** (the actual method bodies).

What are Headers used for?

Header files are included by your other C++ files in order to use their associated code. As such, they provide a handy description of **what a class does**, without bogging you down with **how it works**. They help other engineers understand how they can interact with your objects in a concise format. In a professional environment, you are never the only person reading your code. As such, the legibility of the code you write is very important - other engineers should be able to quickly read and comprehend what you've written.

How are they Structured?

Header files don't need to be limited to a single class - they can contain as much or as little code as you'd like.

Like many other parts of C++, there are quite a few competing conventions regarding what a single header should include. For example:

- Some developers prefer to define a **single class per header** file
- Some developers prefer to incorporate **many classes into a single file**
- Some developers go so far as to include the **entirety of their codebase in a single fil**e - these are called **header-only-libraries**, and they are designed to be easy to integrate with other projects.

Each approach has its own pros and cons as far as managing and building the project goes, and there is no "best" option.

In this task, you will be taking the **middle approach** - defining **several classes in a single header file**, without including their implementations.

See GitHub for Solutions and Diagrams by Conor Griffin

A Few Days Later

You've wrapped up work on the last ticket - those classes were quick to implement once you had their definitions mapped out. Another one down, it feels good to knock these out. You've submitted a pull request with your changes to the relevant git repo.

Soon, another engineer will review your changes and give you feedback, after which they'll be merged in. It's always better to have more eyes on a given piece of code.

You rummage through the sticky notes which briefly outline each ticket. A yellow one catches your eye - looks like a bugfix for the inventory system. Apparently items aren't actually going anywhere when they're removed. Can't have that. Time to make a patch!

The Yellow Ticket

You take a look at the yellow ticket - this is what it says:

The current version of the inventory system does not fully implement item removal. An object removed from the inventory is left in the item list with a quantity of 0. Implement a way to fully remove an item from the inventory when its quantity reaches 0. While reviewing the code, consider whether there are data structures more appropriate for item storage, or if there is code that can be streamlined and condensed.

Doesn't Sound Too Difficult

This seems like a relatively straightforward ticket, you probably won't have any trouble finding the bug.

The inventory system is a part of the Sims you haven't worked on before, so this will be an excellent opportunity to figure out how it works.

The ticket sounds like it might be a little messy, so you could also try to clean things up while you're hunting the bug. Leave the codebase a little nicer for the next person and whatnot.

You spin up your IDE, navigate to the inventory file, and stretch your fingers. Here we go!

Now Let's Make Some Changes

Currently, when an item in the inventory is removed, it simply decrements its quantity. There is no mechanism in place to completely get rid of items. Take a look at the remove_item() function, and figure out how you can modify it to completely eliminate items from the array when their quantity reaches zero.

See GitHub for Solutions and Diagrams by Conor Griffin