# Analysis

## Clients

* Alexis
* Skyler?
* Ben

## Background to the Problem

a

## Interviews?

## Observations of an Existing System

## Research of Scientific Models

### Time Increment Model

### Kepler’s Model

### Derived Model

## Potential Limitations to the Project

## Objectives

## Entity Relationship Diagram

## Things to be computerised?

## Chosen Model

# Documented Design

## Class Structure

Abstract body -> Fixed body & Rails Body

All bodies can have ownership over other bodies; stored implicitly in tree struct.

All bodies traverse trees or stay owned by a body; hence, binary systems are not possible as per design.

Tree is struct because I want to store it on the stack as it needs to be accessed quickly and isn’t often resize

The tree also stores its current members (unordered) for a similar reason. (Up for change if implementing a breadth or depth first search.)

## Flowchart

## Algorithms

### Julian’s Method

### Orbit Transformation

### Save-Load System

Depth first save and load; store the tree as a string of information with characters to denote going up and down the tree. Why avoiding database?

### Tree Traversal for Creation of a Body

MAKE THIS INTO A FLOW CHART PLEASE

When creating a body, it is required that the body has a parent, and hence the parent is searched for. A scope is specified for this search.

SoI is determined as the larger of either standard SoI radius or the distance from the body’s CoM to any child’s apoapsis plus the respective child’s SoI radius. Note that this second procedure may need to be recursive, as it requires the childrens SoI’s to be known first.

When creating a new body, make sure to factor in the fact the new body may be the new parent; if the two bodies both contain each other in their SoIs then that which has a larger mass becomes the parent. Remember to recalculate the SoIs of the children of said now smaller body (and the smaller body itself), as far out children may end up orbiting the new parent instead of the old parent.   
Specify that this has a flaw in the fact that 2 similarly sized objects will be modelled to orbit a given parent body, not a barycentre. In the case of a tie, continue to use the old body as the parent to save on processing, and again warn the user for its inaccuracies.  
Allow collisions of SoI along a path; this is INCREDIBLY unlikely to happen, however it *could* be helped by making children calculate the force between it and all parents and their parents siblings at runtime and swap when the forces misalign; this is however poor on performance for little simulation risk being removed, especially given that for such a case to exist there would already be large inaccuracies in the simulation regardless. If a user tries to create a body in 2 intersecting SoIs of the same scope, default to one arbitrarily as the alternative is to continue to process along all possible routes to check for intersects with other child bodies, to still get an inaccurate model regardless. Simply warn the user of the edge case instead.

While doing this, a heritage is found, which is used to then create the new body. The tree is traversed according to this heritage, until the given parent body is located and a new node is created.

Hill’s spheres describe where an orbit can stably continue around a body; something can technically orbit a body outside of it’s hill sphere but the gravitational influences would throw it off course eventually.

Arbitrary decisions: If apoapsis + hill sphere radius > parent hill sphere radius, warn that any subsequently made bodies must lie in the parent’s hill sphere before being accepted into and child’s gravitational influence. In fact, basically all overlapping hill sphere edge cases will be ignored; if they were not then the code would be a lot less optimised to “fix” issues, when the orbits would already be wildly inaccurate if fixed. It’s either the code runs smoothly to have inaccurate edge cases, or the code runs clunkily for inaccurate edge cases with slightly mildly better user experience.

FOR NOW disallow creation of a body that interferes with pre-existing bodies orbits (as in, make new bodies not contain other bodies in there hill spheres). This currently will not include checks for whether bodies will never cross into other hill spheres, which may need to be communicated to the user.

Moving body initualisation has some methods; use a pointer to the tree to call a tree function (this risked memory failure problems due to the unsafe nature of pointers), splitting the initialiser into 2 parts (this was messy and required the coder to be explicitally aware that two functions were required just to create the body; it kept the tree function called by the tree but it chucked around variables between scopes heavily), or to pass in a copy of the tree and run the function on the copy (worse for memory usage at the point of initialisation, however is the least risky and most clean method of coding). I used the latter