----------------------------------------------------------------------------------------

**TestElegansGermline.hpp**

**This is the main model file that defines a *C. elegans* germ line simulation.**

It has a single method, *TestLarvalDevelopment*, which is called when TestElegansGermlineRunner executes.

Briefly, the main steps carried out by *TestLarvalDevelopment* are:

0) Seed the random number generator.

1) Read in a parameter file specified in the command line arguments.

Make any small changes to the parameter set specified by additional

command line arguments.

2) Set up a collection of Nodes, positioned at the starting location of each cell

3) Turn the Nodes into a NodeBasedCellPopulation, where each cell has a cell cycle model based on your choice of statechart.

4) Specify the starting properties of a cell, and request ancestor tracking.

5) Make a Simulation object out of the cell population and set up its

properties (time step, end time etc.).

6) Specify the force between cells

7) Add a leader cell boundary condition

8) If using contact inhibition in the cell cycle model, add a

VolumeTrackingModifier to measure cell compression

9) Add cell removal by apoptosis and fertilisation

10) Add data output

11) Solve, and save the final state

12) Delete the node collection

To run a simulation, build from the main Chaste directory using the command:

scons b=GccOpt co=1 ./projects/ElegansGermline/test/TestElegansGermline.hpp

An executable TestElegansGermlineRunner is produced in the directory: ./projects/ElegansGermline/build/optimised. It should be run with at minimum two command line arguments; the name of a parameter file and an output directory e.g.:

./TestElegansGermlineRunner “Baseline.txt” “MyOutputDirectoryName”

To write a model of your own building on our code, you will need to produce a similar **TestMyModel.hpp** file in the same directory as this one, perhaps using TestElegansGermline.hpp as a template. More info on writing Chaste tests is available on the Chaste website.

----------------------------------------------------------------------------------------

**TestLoadOffLatticeFromArchive.hpp**

A test that tries to load a saved simulation file and continue the run from there. Useful for checking whether your changes to the code break saving/loading. Also a useful template for how to continue simulations in general.

Note: I find archiving tends to be a bit buggy, and don’t use it much in my *C. elegans* work. If you experience problems with Saving/Loading and need archiving for your work, please let me know and I will try to find a solution.

----------------------------------------------------------------------------------------

**Fertilisation.hpp(cpp)**

A cell killer that removes oocytes from a simulation once they reach the proximal end of the gonad, together with one sperm. A very *C. elegans* germ line specific class! As such, it expects cells to have some quite specific properties in their CellData:

- DistanceAwayFromDTC (so we can work out if proximal end reached)

- Differentiation\_Sperm (= 1.0 for a sperm, 0.0 otherwise)

- Differentiation\_Oocyte (= 1.0 for an oocyte, 0.0 otherwise)

If any of these are missing an error message will inform you.

The fertilisation class has one member variable, *mSpermathecaLength*, which determines how close to the end of the gonad an oocyte has to get before it can be ovulated. Ovulation can only occur from adulthood (17 hours) onward, and only one oocyte may be ovulated in any given time step. The key method is *CheckAndLabelCellsForApoptosisOrDeath,* which decides on the cells that should be removed.

----------------------------------------------------------------------------------------

**OocyteFatedCellApoptosis.hpp(cpp)**

Kills oocyte-fated cells that have yet to reach the proximal arm, with some hourly probability *p*. Again, a *C. elegans* germ line specific class, so it is expected that cells will have certain properties in their CellData, namely:

- DistanceAwayFromDTC (so we can work out whether a cell has reached

the safety of the proximal arm yet)

- OocyteFated (=1.0 for oocyte-fated cells, 0.0 otherwise)

If one of these properties is missing, an error message will inform you. Again, the important method is *CheckAndLabelCellsForApoptosisOrDeath*, which identifies oocyte-fated cells less than 250 microns from the DTC, and generates a random number to decide whether that cell begins apoptosis. The formula for the hourly death probability is based on the existing Chaste class **RandomCellKiller.hpp(cpp)**.

----------------------------------------------------------------------------------------

**RepulsionForceSizeCorrected.hpp(cpp)**

Defines a spring repulsion force between cell centres based on Hooke’s law, but with a slight modification to increase the stability of multicellular simulations. This class inherits from **GeneralizedLinearSpringForce.hpp(cpp)** in the main Chaste code. However, in our version the differences in drag experienced by cells of different sizes are accounted for, by scaling the applied force linearly with cell radius. This step happens at the end of the method *AddForceContribution*, and is the only part of the code that significantly differs from **GeneralizedLinearSpringForce**.

----------------------------------------------------------------------------------------

**CellTrackingOutput.hpp(cpp)**

Outputs the positions of certain tracked cells over time. The constructor takes two arguments: the number of time steps to wait between measurements (an int), and the subset of cells to be tracked (also an int; a value of 5 results in every 5th cell being tracked).   
  
This class inherits from Chaste’s **AbstractCellBasedSimulationModifier**.

Key methods are *SetupSolve,* which opens an output file called *TrackingData.txt* in the simulation results directory, and *UpdateAtEndOfTimeStep,* which is called after each timestep and actually writes the data. The form of the output data is:

Time \t Cell\_ID \t X\_coord \t Y\_coord \t Z\_coord

----------------------------------------------------------------------------------------

**GonadArmDataOutput.hpp(cpp)**

Outputs a collection of data relevant to *C. elegans* germ line simulations. This class expects cells to have a LOT of *C. elegans* specific items of cellData stored; if one of these items is missing an error will warn you.

The constructor takes one parameter: the number of time steps to allow between measurements (an int). This class inherits from Chaste’s **AbstractCellBasedSimulationModifier**, with a *SetupSolve* method that opens an output file, and an *UpdateAtEndOfTimeStep* method that writes out the data.

The form of each row of output data is:

Time \t

GonadLength \t

Cell cycle length, accounting for arrests \t

Sperm count \t   
Proliferative cell count \t

Death rate parameter \t

Total cell count \t

Last proliferative cell distance from DTC in microns \t

First meiotic cell distance from DTC in microns \t

G1 cell count \t

S cell count \t

G2 cell count \t

M cell count \t

MeioticS cell count \t

First row containing 2 meiotic cells, measured in cell rows from the DTC \t

Last row containing 1 mitotic cell, measured in cell rows from the DTC

**WARNING**: The final two items require cell row counting, which is problematic for a computer. **Our current algorithm systematically underestimates the number of cell rows, which is why for publications we still count them manually.** If you’re going to use this output, open up a .vtu output file in Paraview and color cells by their row number to check that the result is sensible or correct for systematic bias. Otherwise, the RScript we provided supports reading from text files containing manual counts, if preferred.

----------------------------------------------------------------------------------------

**GlobalParameterStruct.hpp(cpp)**

A singleton class inheriting from Chaste’s **SerializableSingleton**. This makes available an object containing a vector of parameter values, which are set initially by reading from a text file. The parameter values are then available to access from any other file in the project. If you try to retrieve a parameter value before it has been set, an error will result. Key methods are:

*Instance* – used to get a pointer to the single active GlobalParameterStruct

*ConfigureFromFile* – takes in a filename and path and sets parameters from it

*GetParameter* – takes in an int and returns the parameter value stored at that

index

*GetDirectory* – returns the name of the simulation output directory

*ResetDirectoryName* – Takes in a std::string and resets the name of the

output directory to it.

*ResetParameter* – Takes in an int index and a double, and sets the value of

parameter number “index” to that double.

**Note**: classes that use GlobalParameterStruct should serialize it in their .hpp file’s serialize method, by including the following block:

*SerializableSingleton<GlobalParameterStruct>\* p\_params\_wrapper = GlobalParameterStruct::Instance()->GetSerializationWrapper();*

*archive & p\_params\_wrapper;*

This class should also be used sparingly, because success depends on you knowing a lot about how the user’s parameter files look. Here, I’ve tried to limit its use to files that are clearly *C. elegans* germ line specific.

See *Baseline.txt* in the data directory, and also the README for more information on how parameter files should be laid out.

----------------------------------------------------------------------------------------

**StatechartCellCycleModel.hpp**

A wrapper around a statechart, designed to make the chart “look” like a normal cell cycle model to the rest of Chaste. Inherits from **AbstractCellCycleModel** and our own **AbstractStatechartCellCycleModel** (described below).

This class’s main responsibilities are:

1) Defining the length of the G1, S, G2 and M phases, and how they update

over time. Initially these are set to values suitable for a colonic crypt

simulation by **AbstractCellCycleModel**. They are reset to more suitable

values by this class’s *C. elegans* specific child

**ElegansDevStatechartCellCycleModel**.

2) Creating a daughter cell cycle model when a cell divides. See the method

*CreateCellCycleModel*.

3) Initialising a cell cycle model. See the methods *SetCell* and *Initialise* (only

called at the start of the simulation)

4) Archiving itself and its statechart when a simulation is saved/loaded. See

the *Load* and *Save* methods at the end of the file.

A StatechartCellCycleModel holds a pointer to a statechart *(pStatechart)*, which takes over responsibility for updating the cell cycle phase and determining when a cell should undergo division. See the methods *ReadyToDivide, ResetForDivision, UpdateCellCyclePhase, ResetForDivision, SetReadyToDivide* and *SetCellCyclePhase*.

This class expects a statechart to respond to certain events and implement certain functions. The full list of required properties for a statechart model is given in the description of **FateUpcoupledFromCycle.** In the event that something is missing, a template substitution error will result.

**Users wishing to write their own statechart models should not need to edit this class, except to change the max number of states allowed (line 38).** Usually, you can just write your ownstatechart to plug into this class, taking **FateUpcoupledFromCycle.hpp(cpp)** as an example.

----------------------------------------------------------------------------------------

**AbstractStatechartCellCycleModel.hpp**

Defines two methods that every StatechartCellCycleModel should expose: *SetCellCyclePhase,* which allows a statechart to control the cell cycle, and *SetReadyToDivide*, which allows a statechart to call for division. These methods are defined in a separate class so that **StatechartInterface** can know that they exist, without having to mess around referencing a template class.

----------------------------------------------------------------------------------------

**StatechartInterface.hpp**

Defines a number of utility functions that will be required by many statechart models. These include things like setting the cell radius and calling for division.

----------------------------------------------------------------------------------------

**ElegansDevStatechartCellCycleModel.hpp**

As far as possible, we have tried to keep **StatechartCellCycleModel** free of *C. elegans* specific code, so it can be reused in other contexts. This child class is where functions relevant to the worm are carried out.

Specifically, here we override the getter methods for the G1, S, G2 and M phase durations provided in **AbstractCellCycleModel**. When a phase duration is requested by the statechart, this class will look at the simulation time and parameters, and calculate an appropriate phase length for the current stage in development. Phase lengths thereby constantly change as the worm develops.

We went the route of overriding these getter functions, because calculating the appropriate phase length takes a little bit of work, and it seemed wasteful to ensure at every single timestep that every cell cycle phase length is up to date. **However, this does create the potential for bugs, so take care to note that mG1Duration and similar members of AbstractCellCycleModel are NOT kept updated throughout the run.** **If a cell cycle phase length needs to be accessed, it should always be done by calling a getter function.**

Otherwise, here we override the *Initialise* method to allow for *C. elegans* specific setup of phase lengths, and we override *CreateCellCycleModel* to ensure that daughter cells get a copy of this class, not its parent.

----------------------------------------------------------------------------------------

**FateUncoupledFromCycle.hpp(cpp)**

An example statechart model of cell behaviour, which can be used to replace the template parameter in **StatechartCellCycleModel**. Gets its name from the fact that the sperm/oocyte fate decision is not coupled to the cell cycle (i.e. that decision need not occur in a particular phase).

This class makes use of the Boost Statecharts library by Anderas Huber: <http://www.boost.org/doc/libs/1_57_0/libs/statechart/doc/index.html>.

The general idea is that the header file defines the structure of a chart, using templates. Each state is a templated struct, with parameters:

<my name, parent name, initially active child (if applicable)>

The header file also declares a struct for each event that the chart will respond to, and it declares which events each state listens for.

Meanwhile the .cpp file implements two sets of functions. The first are associated with the statechart object itself, and are used in communicating with the rest of Chaste, via **StatechartCellCycleModel**. The second are associated with specific states, and give the response that will be triggered by a particular event. The responses (which may include transitions or changes to the cell’s properties) are what define the behaviour of a statechart model.

**This is the file that users need to adapt or replace in order to use their own statechart models in a simulation.** Because it’s much easier to see what’s going on in the context of an example, we’ve commented this portion of the code thoroughly, and we suggest it might be easiest to look through it step by step as you produce your own model.

For completeness though, here is a list of all the methods and member variables that a Statechart model must define to be compatible with the rest of Chaste (are these all marked REQUIRED in the code).

* *CellPtr pCell*  
  A pointer to the cell being controlled by this statechart.
* *double TimeInPhase*  
  Used to count the time a cell has spent in its current cell cycle phase.
* *boost::shared\_ptr<CLASSNAME>Copy(boost::shared\_ptr<CLASSNAME> myNewStatechart)*Used to copy the state of the parent chart into a daughter chart on division
* *std::bitset<MAX\_STATES> GetState()*

Encodes the current state as a bitset for archiving

* *std::vector<double> GetVariables()*

Packages all variables associated with the chart into a vector for archiving.

* *void SetState(std::bitset< MAX\_STATES > state)*

Sets the state of a statechart from a bitset, for loading from an archive.

* *void SetVariables(std::vector<double> variableValues)*Sets the values of chart-associated variables from a stored vector, again for loading from an archive.

A statechart should also respond to the following events:

* *EvCheckCellData*Fired by **StatechartCellCycleModel** each time step to request that the chart updates
* *EvGoToCellCycle\_Mitosis\_M*
* *EvGoToCellCycle\_Mitosis\_S*
* *EvGoToCellCycle\_Mitosis\_G2*
* *EvGoToCellCycle\_Mitosis\_G1*Events fired by **StatechartCellCycleModel** at the start of a new simulation to force cells into unsynchronised initial phases.

Failure to implement the required methods, variables, or responses in a statechart model will result in a template substitution error.

Admittedly, Boost Statecharts code can be a bit long and complicated to write. I’ve previously worked on a code generator for this project, which generates the necessary C++ files automatically from a simpler representation of a chart (e.g. a diagram in a JavaScript GUI). If there’s any interest in that, I’ll do some more work on it and release it here.

----------------------------------------------------------------------------------------

**FateDecisionCoupledToCycle.hpp(cpp)**

A second example of a statechart model of cell behaviour. The primary difference is that in this model, the sperm/oocyte fate decision occurs simultaneously with entry into meiosis, and is therefore coupled to the cell cycle in the sense that it can only happen during G1.

Hence the name **FateDecisionCoupledToCycle**.

From a code point of view, this class is very similar to **FateUncoupledFromCycle.hpp(cpp)**, and you should look at **FateUncoupledFromCycle** for reference as it is fully commented. This class is also *C. elegans* specific, as you might expect.

----------------------------------------------------------------------------------------

**DTCMovementModel.hpp(cpp)**

This file specifies the movement of the Distal Tip Cell, and maintains a collection of equally spaced points on the DTC’s path (*PathPointCollection*).

Implemented as a simulation modifier. The constructor takes 5 arguments:

*Unc5* (bool) = whether the gonad turn has begun

*Vab3* (bool) = whether the DTC has stopped as adulthood has been reached

*StartingPointLocations* (std::vector of c\_vector<double,3>) = initial collection

of points on DTC path

*StartingPointTypes* (std::vector<int>) = flags carrying info about each midline

point (e.g. is it part of straight or turn)

*currentLocation* (std::vector<double,3>) = initial DTC position

*spacing* (double) = gap between midline points.

The key method is *UpdateAtEndOfTimeStep,* which is called at each timestep and refreshes the DTC position.

**Note:** This is one of two boundary condition classes that will be refactored in the near future. The aim of the refactor will be to create more generally applicable Abstract classes, and to remove *C. elegans* specific code into children of these classes, as we have done for statecharts. The leader cell boundary condition could then be more easily applied to other systems.

----------------------------------------------------------------------------------------

**LeaderCellBoundaryCondition.hpp(cpp)**

This file enforces a tube shaped boundary condition, based on the collection of leader cell path points it receives from **DTCMovementModel**. The boundary consists of all points a certain distance from the leader cell path.

Inherits from **AbstractCellPopulationBoundaryCondition.** The constructor has 3 arguments:

*pLeaderCell* (boost::shared\_ptr< DTCMovementModel<3> >) = a pointer to the object controlling leader cell movement.

*TubeRadius* (double)= initial radius of the boundary, measured from midline.

*MaxMovementDistance* (double) = max distance moved by a cell per timestep.

The key method is *ImposeBoundaryCondition,* which repositions cells lying outside the boundary*,* and places them inside again. **Note** that Boundary Condition verification has been disabled for speed in this class. It should be implemented if this class is to be used in combination with other, potentially conflicting, boundary conditions.

**Note:** This is one of two boundary condition classes that will be refactored in the near future. The aim of the refactor will be to create more generally applicable Abstract classes, and to remove *C. elegans* specific code into children of these classes, as we have done for statecharts. The leader cell boundary condition could then be more easily applied to other systems.