

CReST Developers' Guide

John Cartlidge & Alex Sheppard

University of Bristol

Last Modified: September 2012

This document is being periodically updated and is not complete.

Documentation on the extending the code is required, including:

- Adding a module
- Adding a new Event
- Adding a new Subscription Type (Subscriptions module example)
- Factory classes and Singletons

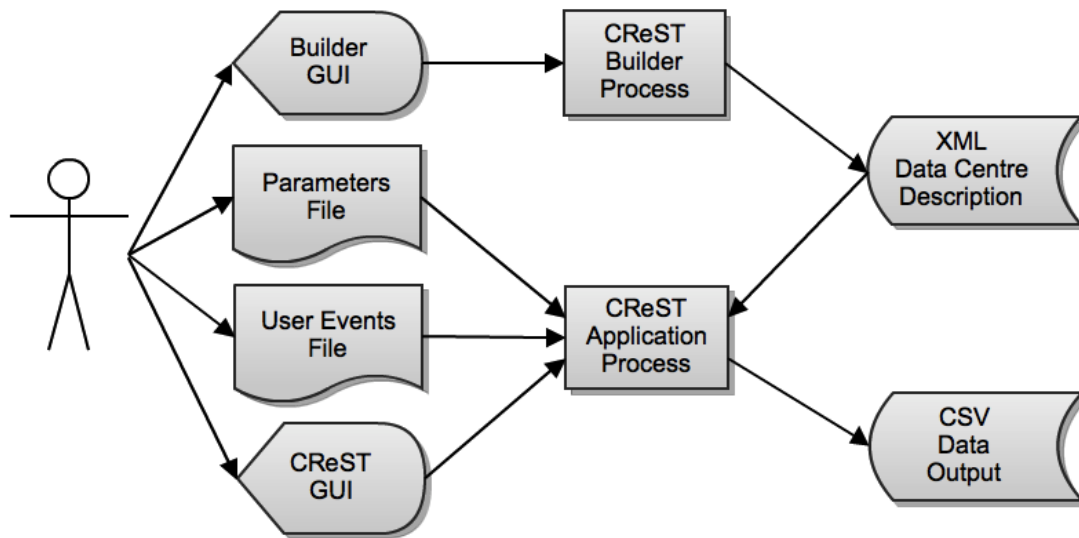


Fig 1. CReST architecture diagram. As input, CReST requires an XML configuration file describing each data centre. XML configuration files contain a full specification of all hardware and can be large and difficult to write manually. The CReST builder offers a graphical interface for users to generate and edit these files. In conjunction, CReST is able to read simulation parameters from a simple text parameters file. Parameters in this file overwrite those duplicated in the XML file and offer an easy way for users to edit a simulation configuration, or to run multiple simulations with varying configuration parameters. Finally, users can generate their own events via a User Events File, a simple text file defining event type and time of event. CReST can be run with or without a graphical interface. When using the GUI, users are presented with run-time visual feedback. All simulation data is logged to a CSV file.

CReST Program Flow

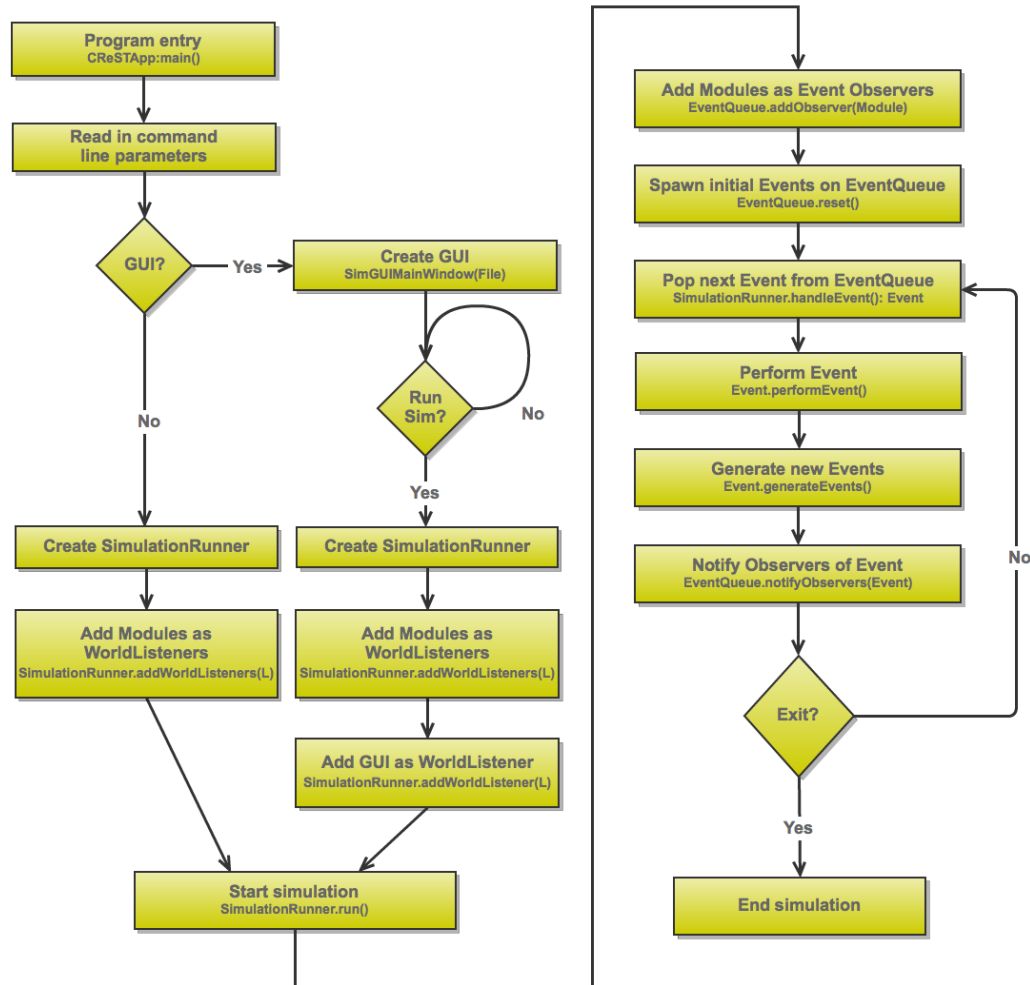


Fig 2. CReST is run from the main entry point `CReSTApp.main()`, which reads in configuration parameters including: the name of the XML file containing the data centre configuration, an optional parameters text file containing simulation parameters (these overwrite the XML file) and an option to specify whether to run the GUI or not (-ng). If a GUI is required, it is created using the `SimGUIMainWindow` class. The GUI enables users to edit the name of the XML configuration file before a simulation is started. The `SimulationRunner` object is created and all simulation `ModuleRunners` are added as listeners using the `SimulationRunner.addWorldListeners(L)` method. If a GUI object has been created, it also adds itself as a `WorldListener`. If there is no GUI, the simulation automatically begins with a call to `SimulationRunner.run()`, else the GUI calls this method when the user selects “run”. The `SimulationRunner` sets all `ModuleRunner` objects to observe Events in the `EventQueue` by calling `EventQueue.addObserver(ModuleRunner)`. Initial events are then added to the `EventQueue` by calling `EventQueue.reset()`, which generates a new `EventSpawner`. The `SimulationRunner` then enters the simulation “event loop” until an “Exit” signal is received from a `StopSimEvent` (either generated by the GUI or internally by the simulation). An event is popped from the `EventQueue` via a call to `SimulationRunner.handleEvent()`. Every event contains two methods that are called in sequence from the `Event.perform()` method. Firstly,

Event.performEvent() is called to perform the event logic, for example a Server failure. Secondly, Event.generateEvents() is called to add new Events to the EventQueue that are generated as a consequence of the Event logic, for example a new server FixEvent. EventQueue observers are then notified of the event via a call to EventQueue.notifyObservers(Event e). All ModuleRunners (which implement the Observer interface) then receive the Event e via an internal call to Observer.update(e) and act accordingly, potentially adding new Events to the EventQueue based on their logic. If the Event is not a StopSimEvent a new Event is popped from the EventQueue via SimulationRunner.handleEvent(), else the simulation exits.

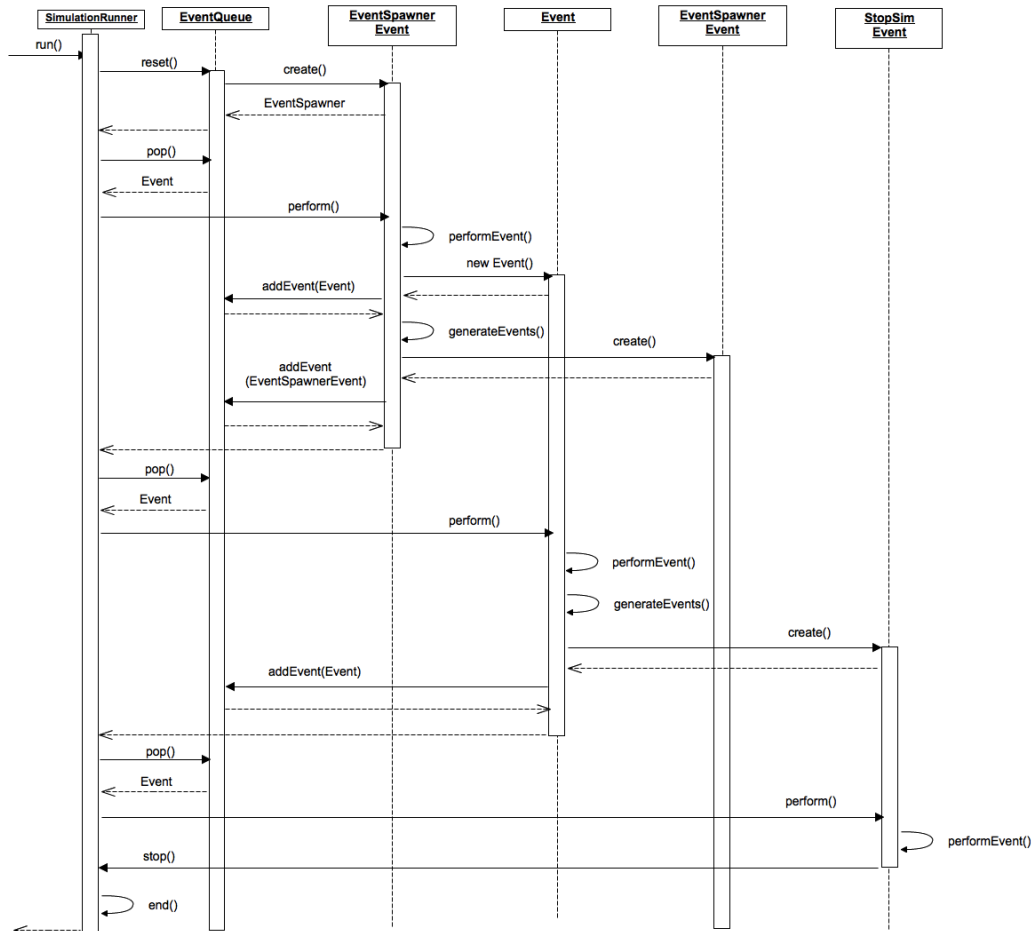


Fig 3. Process control and objects timeline for the SimulationRunner and EventQueue. When the SimulationRunner.run() method is called, the EventQueue is created via a call to EventQueue.reset(). An initial EventSpawnerEvent is then created via a call to EventSpawnerEvent.create(). A new EventSpawnerEvent is returned and added to the EventQueue and control is passed back to the SimulationRunner. A call to EventQueue.pop() pops the next Event off the EventQueue stack and calls Event.perform(). Event.perform() calls two protected sub-methods: Event.performEvent() and Event.generateEvents(). The first Event to be popped from the EventQueue is always an EventSpawnerEvent. The internal call to EventSpawnerEvent.performEvent() instantiates new Event objects and adds them to the EventQueue by passing them to EventQueue.add(Event e). The internal call to EventSpawnerEvent.generateEvents() creates a new EventSpawnerEvent and again adds this to the EventQueue using EventQueue.add. The first EventSpawnerEvent is now complete and is released from memory as control is returned to the SimulationRunner. While inside the event loop, the SimulationRunner continues to pop events from the EventQueue, each time calling Event.perform(). The Event then performs its own logic via an internal call to Event.performEvent() before generating new events via the method Event.generateEvents(). Finally, when a StopSimEvent is popped from the EventQueue, the method StopSimEvent.performEvent() returns a stop signal via a call to SimulationRunner.stop() and the simulation exits.

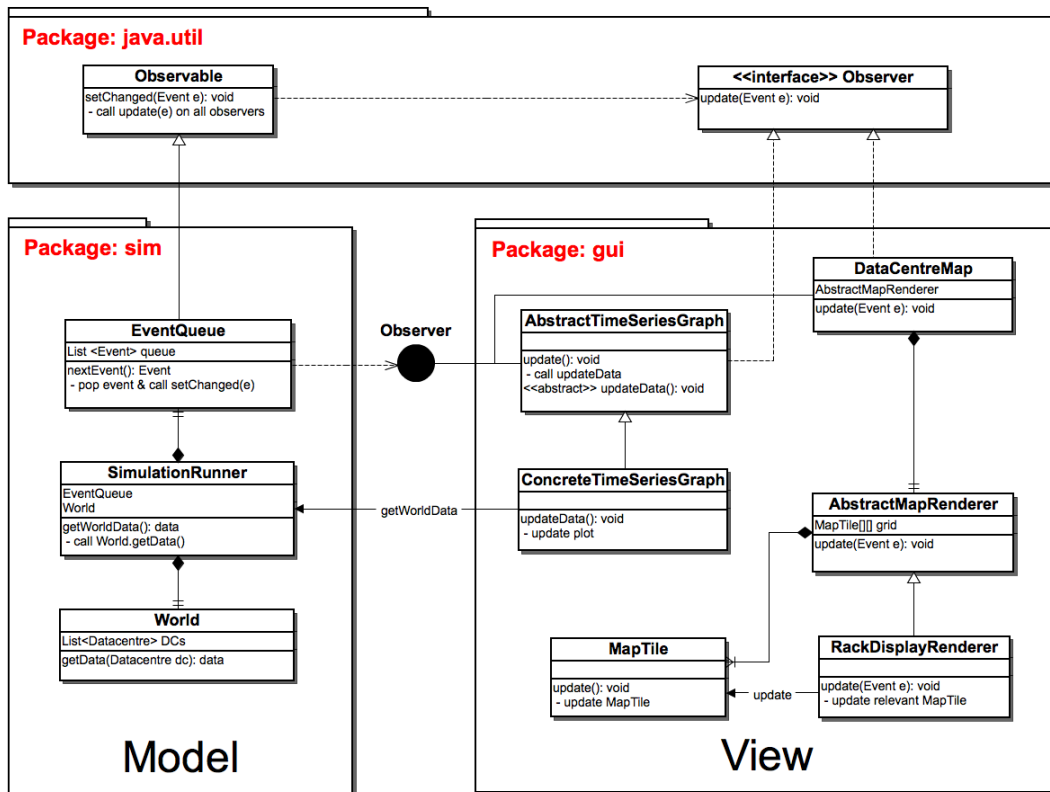


Fig 4. The Observable/Observer interface is used to keep a strict separation between Model (the SimulationRunner, EventQueue and World objects) and View (in this case, the Graphical User Interface). In the GUI view, every time series graph extends the AbstractTimeSeriesGraph that implements the Observer interface. Likewise, the DataCentreMap also implements the Observer interface. In the model, the EventQueue extends the Observable class. When the SimulationRunner pops a new Event from the EventQueue, the Observable.setChanged(Event) method is called. This calls the Observer.update() method in all objects implementing the Observer interface in the view. The DataCentreMap contains a MapRenderer that updates based on the new Event that they receive. Concrete implementations of the AbstractTimeSeriesGraph get the latest model data to plot via a call to SimulationRunner.getWorldData().

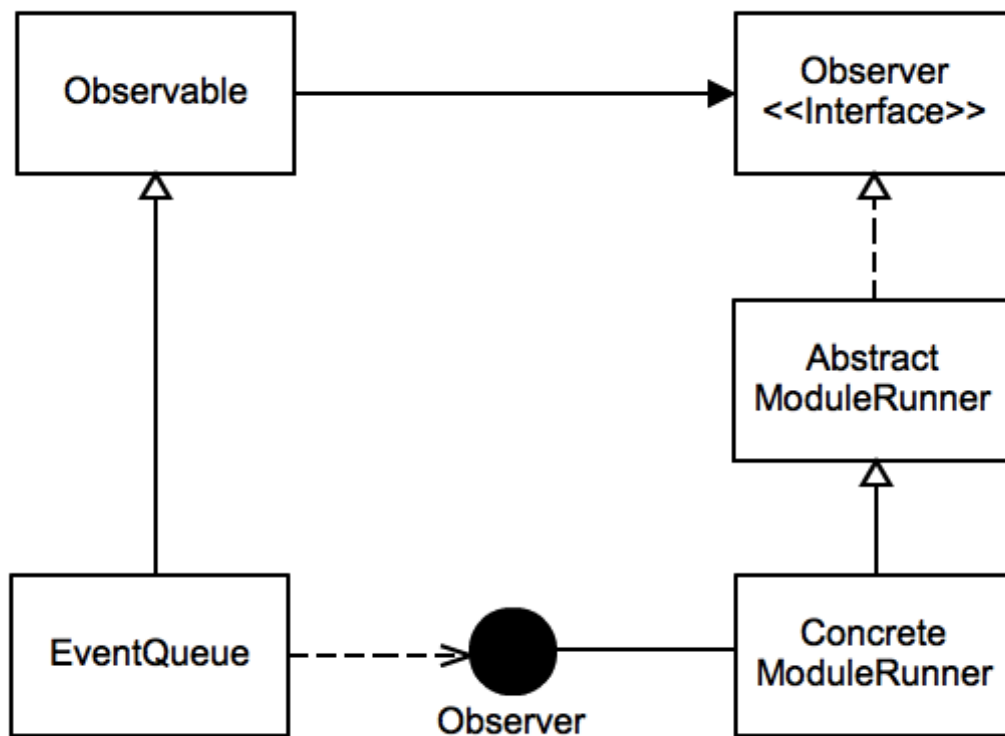


Fig 5. Each module has a ConcreteModuleRunner object that extends the AbstractModuleRunner class. The AbstractModuleRunner implements the Observer interface which is used to pass new Events popped from the EventQueue via the Observer.update() method. Using a similar architecture to the GUI shown in Fig 4., ModuleRunners act as a view on the EventQueue model, using the Observer/Observable interface to observe new simulation Events. Each ModuleRunner acts on the Events that are observed, potentially adding new Events to the EventQueue as a consequence.

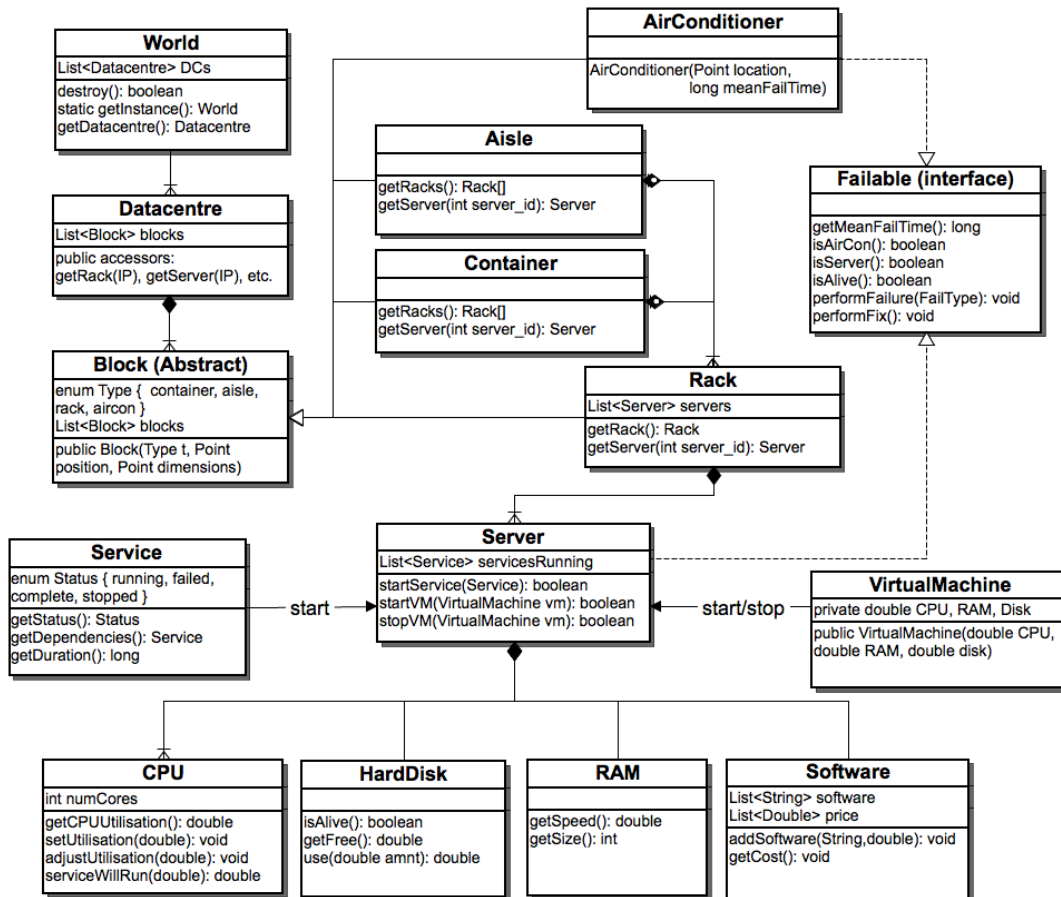


Fig 6. Class hierarchy for objects representing physical hardware in the model. Each simulation contains a World object that contains one or more Datacentre objects. Each datacentre contains a List of (at least one) abstract Block objects. Blocks are extended by 4 concrete types: Aisle, Container, AirConditioner and Rack. Aisles and Container objects contain a List of (at least one) Rack objects. In turn, Rack objects contain a List of (at least one) Server object. Both Server and AirConditioner objects implement the Failable interface, since they represent hardware that can fail. Server objects contain HardDisk, RAM and Software objects and a List of (at least one) CPU objects. Servers can run Service and VirtualMachine objects, which are started and stopped via methods stop() and start().

ViewEditDatacentre

This view is a simpler version of the PhysicalView. It could be seen as a 'basic' setup, whereas the PhysicalView is 'advanced'. It allows the user to create a filled datacentre by just entering either the datacentre dimensions, or the number of servers and room dimension ratio. Therefore you can very quickly build a datacentre to run a simulation on, without having to worry about individual rack or server configurations.

The view is closely linked to the PhysicalView so that the 2 views remain consistent when the user is configuring a datacentre. The PhysicalView contains the Working World object. There are 4 methods New, Update, Delete and a list selection, which execute the same code in both views.

Note:

As a forewarning, if you set an item in a JList, it will fire the corresponding ListSelectionListener for that JList. This proved a bit of an issue when trying to keep the 2 JLists in each view consistent.

For example, if you change the selected item in the ViewEditDatacentre view, you will need to update the selected item in the list on the PhysicalView. In setting the item, this list event listener will fire as if you have selected the item, and now this will update the selected item in the ViewEditDatacentre view. This loop will continue.

The solution is to have 2 types of selection for the listener. Whether it was the user that selected the list item (normal selection) or whether it has happened as a result of being set by the other view (background selection).

In short, the background selection does not set the selected list item in the other view and therefore stops the loop.

Interactions between the 2 views:

ViewEdit (VE)	SimplePhysical (SP)
Init()- Disable all fields	-
New()- SP new ConfigWorld SP set selected index	
List selection (background) Populate fields Draw map	-
List selection (normal) SP set selected index	List selection (normal) populateDatacentre VE set list items & selected index (background)
Update()- Save datacentre SP set selected index	Generate()- Same as SP New()
Delete()- Remove working datacentre populateWorld	

Some forum threads on this issue:

<http://stackoverflow.com/questions/3092834/can-i-set-the-selected-item-for-a-jlist-without-having-an-event-thrown-to-the-li>

<http://stackoverflow.com/questions/3318824/jlist-clearselection-issue>

Although not problematic, but worth being aware of, is that the listener is fired twice, on mouse click and mouse release. The *e.getValueIsAdjusting()* solution stops the second listener event and thus reduces code execution time.

<http://forums.devshed.com/java-help-9/listselectionlistener-registers-two-events-on-one-mouse-click-285809.html>

Datacentre Generation

The ViewEditDatacentre view can generate a datacentre directly by entering the dimensions, or indirectly by the number of servers required and a DC size ratio.

Using the dimensions fits in with the existing datacentre generator class. Using the number of servers and ratio requires some calculation to transform this into the dimensions, before calling the generator.

The Maths

There are code comments in package config.DatacentreGenerator based on the following logic:

- Calculate the number of servers that fit in a rack
- How many racks are needed for the required number of servers?
- Use quadratic formula to find the datacentre length (num aisles)
- Find the positive root and round up
- If the shorter dimension is an even number, there will be 2 empty rows at the end. Remove extra row.
- Make sure there is space for at least 1 aisle with gap either side
- Make sure that there are not more aisles than racks required!
- Number of aisles that fit in the shorter dimension
- With this many aisles, how long does each aisle need to be to fit the number of racks
- Set datacentre dimensions now we know what they should be.

Once the dimensions have been determined the generator class makes use of the new Layout Pattern factory methods in package builder.datacentre.layout. There are 3 new patterns, Long Aisles, Split Aisles and Random. The existing method (Old Method) is used as default and each has a method generateLayout().

The generateLayout method sets the positioning of the aisles, racks, servers and aircon within the datacentre and returns a ConfigDatacentre object as before. Each server object has as direction value (North, South, East, West) which is the direction the front of the server faces. In the PhysicalView there is an option to set the direction of all the servers within the current rack.

Replacements

The replacements module runner observes failure events of type fix and acts upon four possible situations that can occur on both servers and aircon units.

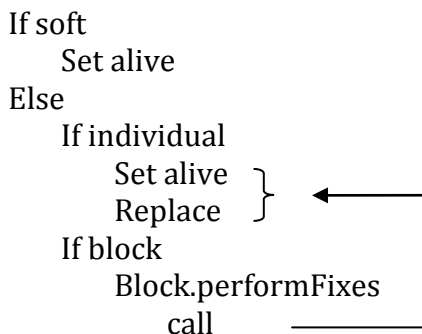
In the builder, there is an option to specify how you want replacements to occur, either in block replacements or individually. Block replacements won't occur until the block threshold (the proportion of dead servers/aircon in the block) has been reached, again set in the builder.

	Block	Individual
Soft	Set alive	Set alive
Hard	Block.performFixes - Set alive - Replace	Set alive Replace

The module also differentiates between soft and hard failure events. A failure of type soft only needs to be set alive / working again, however this code should never be reached as it is dealt within the server/aircon class and is only left in as a safeguard. This is due to the fact that soft fixes should occur regardless of whether the replacements module is active or not.

In the case of a hard failure, the server/aircon needs to be physically replaced. Replacement servers are available in the XML config file or by a function which determines the specifications based on time. Again, the method to use is set in the builder.

In the case of a block replacement, there is an additional step. The block that the current server/aircon is contained within needs to be checked to see if the threshold has been reached. If it hasn't, then the server/aircon isn't replaced. If it has, then each server/aircon is looped through in turn and replaced, using the same method as an individual replacement.



Age

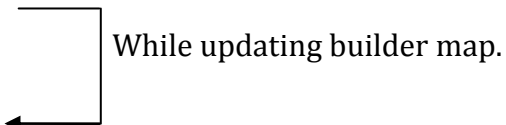
There is a basic ageing function within the builder to allow you to build a datacentre with different hardware representing failures that have happened over time.

When the replace button is pressed, the ReplacementServers update() method is called and follows the following logic:

- for each rack, calculate mean fail time by looping through each server
- move to point in time
- find racks with fail time less than current
- replace rack and fill with replacement servers.

The time that it jumps forward to is specified at the top of the class, by default 200 days.

Ideally this should be replaced by using a simulation with only the failures and replacements modules on. The challenge here is getting the Datacentre object from the simulation back into the builder, which uses a ConfigDatacentre object, in real time so that the datacentre map can be updated.

1. ConfigDatacentre object in Builder
Save XML
Run simulator
 2. Datacentre object in Simulator
(via World.getInstance.getDC())
 3. ConfigDatacentre object in Builder
- 
- While updating builder map.