# Project Structure

**/** (root)

+- **data**

| This folder contains the data files.

+- **geotools**

| This folder contains the GeoTools libraries.

+- **lib**

| +- SimulatorExtensions.jar

+- **src**

+- **org/geotools/swing/tool**

| +- InfoToolHelper.java

+- **simulator/extensions/location**

| +- **model**

| | +- Graph.java

| | +- LocationPool.java

| +- **ui**

| | +- LogWindow.java

| | +- MapWindow.java

| +- **utils**

| +- GraphUtils.java

| +- InitUtils.java

| +- LoadingUtils.java

| +- MappingUtils.java

| +- StyleUtils.java

+- GISSystem.java

+- LocationAwareExtension.java

+- RPCHandler.java

# File: “src/simulator/extensions/location/LocationAwareExtension.java”

This file is responsible for providing the CPN Tools simulation extension. This is done by extending the AbstractExtension class, which is provided by the CPN Tools package.

The first time the extension is loaded, we set up the extension by initializing the logging window, map window, GIS system, construct an RPC handler object and set up the routing graph. This is only done once globally.

CPN Tools will call a getRPCHandler() method to get back an object which will be used to provide the RPC method calls.

|  |  |
| --- | --- |
| **package** simulator.extensions.location;  **import** java.io.IOException;  **import** java.io.OutputStream;  **import** java.io.PrintStream;  **import** java.util.Timer;  **import** java.util.TimerTask;  **import** org.cpntools.simulator.extensions.AbstractExtension;  **import** org.cpntools.simulator.extensions.Channel;  **import** org.cpntools.simulator.extensions.Extension;  **import** org.geotools.map.MapContent;  **import** simulator.extensions.location.model.Graph;  **import** simulator.extensions.location.ui.LogWindow;  **import** simulator.extensions.location.ui.MapWindow;  **import** simulator.extensions.location.utils.GraphUtils;  **import** simulator.extensions.location.utils.InitUtils;  **import** simulator.extensions.location.utils.MappingUtils;  **public** **class** LocationAwareExtension **extends** AbstractExtension {    **private** **static** LogWindow *LOG\_WINDOW* ;  **private** **static** RPCHandler *RPC\_HANDLER*;  **private** **static** MapWindow *MAP\_WINDOW*;  **private** **static** Graph *GRAPH*;  **private** **static** GISSystem *GIS\_SYSTEM*;  **private** **static** **boolean** *isInitialized* = **false**;    **public** LocationAwareExtension() {  **super**();    **if** (!*isInitialized*) {  *LOG\_WINDOW* = **new** LogWindow();  }    System.*setErr*(**new** PrintStream(**new** OutputStream() {  @Override  **public** **void** write(**int** b) **throws** IOException {  *LOG\_WINDOW*.showByte((**char**) b);  }  }));    **if** (!*isInitialized*) {  *LOG\_WINDOW*.showInfo("Location Aware Extension initializing ...");  *LOG\_WINDOW*.showInfo(" - Log window");  *LOG\_WINDOW*.showInfo(" - RPC handler");  *RPC\_HANDLER* = **new** RPCHandler(*LOG\_WINDOW*);  *LOG\_WINDOW*.showInfo(" - Location pool");  **try** {  InitUtils.*initialize*(*RPC\_HANDLER*.getLocationPool(), *LOG\_WINDOW*);  } **catch** (IOException e) {  *LOG\_WINDOW*.showInfo("Fatal error: " + e.getMessage());  }  *LOG\_WINDOW*.showInfo(" - Graph");  *GRAPH* = GraphUtils.*createGraphFromLocationPool*(  *RPC\_HANDLER*.getLocationPool());  *LOG\_WINDOW*.showInfo(" - Map content");  MapContent content = MappingUtils.*createMapContentFromLocationPool*(  *RPC\_HANDLER*.getLocationPool());  *LOG\_WINDOW*.showInfo(" - Map window");  *MAP\_WINDOW* = **new** MapWindow(content);  *LOG\_WINDOW*.showInfo("- GIS system");  *GIS\_SYSTEM* = **new** GISSystem(*RPC\_HANDLER*, *LOG\_WINDOW*, *MAP\_WINDOW*, *GRAPH*);    Timer gisTimer = **new** Timer(**true**);  gisTimer.schedule(**new** TimerTask() {  **public** **void** run() {  *GIS\_SYSTEM*.startPoller();  }  }, 10 \* 1000);    *isInitialized* = **true**;  }  }  @Override  **public** Extension start(Channel c) {  *LOG\_WINDOW*.showInfo("Location Aware Extension starting");  **return** **super**.start(c);  }  @Override  **public** **int** getIdentifier() {  **return** Extension.*TESTING*;  }  @Override  **public** String getName() {  **return** "Location Aware Extension";  }    @Override  **public** Object getRPCHandler() {  *LOG\_WINDOW*.showInfo("getRPCHandler() invoked");  **return** *RPC\_HANDLER*;  }    **public** **static** **void** main(String[] args) {  LocationAwareExtension lae = **new** LocationAwareExtension();  **try** {  ((RPCHandler)lae.getRPCHandler()).assignFeature(  "RepairTeam1", "CRT", "Customer4");  } **catch** (Exception e) {  e.printStackTrace();  }  // System.exit(0);  }    } | *Constructor: set up all windows and systems when loaded the first time.*  *InitUtils, GraphUtils and MappingUtils is used to create various objects.*  *The GISSystem class is a background runner which is responsible for updating the features. In our example, the GIS system randomly moves repair teams, but more complex integrations with other running systems are possible.*  *We wait ten seconds before initializing the GIS system to allow the system some time to set up.*  *CPN Tools will call the getRPCHandler() method to retrieve an object it can execute method calls on.*  *main() method only used for testing purposes. Not called in actual production environment.* |

# File: “src/simulator/extensions/location/RPCHandler.java”

This object provides the actual RPC methods to CPN Tools. There is no need to explicitly indicate any possible method calls, as CPN Tools will automatically discover all methods which: (1) return a String, Boolean or Integer value (BIS, not the primitive boolean or int types), (2) only take BIS values as arguments and (3) can throw an Exception.

The RPC handler needs to implement the NamedRPCHandler interface, included in the CPN Tools simulator extension package.

Our implementation includes all basic geometric relation functions, provided by GeoTools.

|  |  |
| --- | --- |
| **package** simulator.extensions.location;  **import** java.util.Set;  **import** org.cpntools.simulator.extensions.NamedRPCHandler;  **import** org.geotools.data.DataUtilities;  **import** org.geotools.feature.simple.SimpleFeatureBuilder;  **import** org.geotools.graph.path.Path;  **import** org.geotools.graph.structure.Graphable;  **import** org.opengis.feature.simple.SimpleFeature;  **import** org.opengis.feature.simple.SimpleFeatureType;  **import** simulator.extensions.location.model.Graph;  **import** simulator.extensions.location.model.LocationPool;  **import** simulator.extensions.location.ui.LogWindow;  **import** simulator.extensions.location.utils.GraphUtils;  **import** com.vividsolutions.jts.geom.Geometry;  **import** com.vividsolutions.jts.geom.Polygon;  **public** **class** RPCHandler **implements** NamedRPCHandler {  **private** **final** **static** String *SEPARATOR* = "#";    **private** **final** LogWindow logWindow;  **private** **final** LocationPool locationPool;    **public** RPCHandler() {  **this**(**new** LocationPool(), **null**);  }  **public** RPCHandler(LogWindow logWindow) {  **this**(**new** LocationPool(), logWindow);  }    **public** RPCHandler(LocationPool locationPool) {  **this**(locationPool, **null**);  }    **public** RPCHandler(LocationPool locationPool, LogWindow logWindow) {  **this**.logWindow = logWindow;  **this**.locationPool = locationPool;  }  **public** LocationPool getLocationPool() {  **return** locationPool;  }  @Override  **public** String structureName() {  **return** "LA";  }    **public** String[] untangle(String s) {  **return** s.split(*SEPARATOR*);  }    **public** String tangle(String[] s) {  **if** (s.length == 0) **return** **null**;  StringBuilder out = **new** StringBuilder();  out.append(s[0]);  **for** (**int** x = 1; x < s.length; ++x) {  out.append(*SEPARATOR*).append(s[x]);  }  **return** out.toString();  }  **public** **void** log(String data) {  **if** (logWindow != **null**)  logWindow.showInfo(data);  }    **public** String sep() **throws** Exception {  **return** *SEPARATOR*;  }    // Define the RPC endpoints here    **private** String lastCustomer = "";  **public** String af(String fname, String transition) **throws** Exception {  log("assignFeature"+" "+fname+" "+transition);  SimpleFeature feature = locationPool.getFeature(fname);  **if** (transition.equals("RCC")) {  lastCustomer = fname;  }  **if** (transition.equals("CRT")) {  feature.setAttribute("status", "onroute");  feature.setAttribute("destination", lastCustomer);  }  **if** (transition.equals("RRT")) {  feature.setAttribute("status", "available");  feature.setAttribute("destination", "");  }  **return** fname;  }    **public** String assignFeature(String fname, String transition,  String customer) **throws** Exception {  log("assignFeature"+" "+fname+" "+transition+" "+customer);  SimpleFeature feature = locationPool.getFeature(fname);  **if** (transition.equals("CRT")) {  feature.setAttribute("status", "onroute");  feature.setAttribute("destination", customer);  }  **if** (transition.equals("RRT")) {  feature.setAttribute("status", "available");  feature.setAttribute("destination", "");  }  **return** fname;  }    **public** Boolean typeEquals(String feature, String featureType)  **throws** Exception {  log("typeEquals"+" "+feature+" "+featureType);  **return** locationPool.getFeatureType(feature).equals(featureType);  }    **public** Boolean attributeEquals(String feature, String propertyName,  String propertyValue) **throws** Exception {  String av = getAttribute(feature, propertyName);  log("hasProperty"+" "+feature+" "+propertyName+" "+propertyValue+" "+av);  **return** av.equals(propertyValue);  }    **public** String getAttribute(String feature, String propertyName)  **throws** Exception {  log("getAttribute"+" "+feature+" "+propertyName);  SimpleFeature location = locationPool.getFeature(feature);  Object att = location.getAttribute(propertyName);  **return** att.toString();  }    **public** Boolean equals(String f1, String f2) **throws** Exception {  log("equals"+" "+f1+" "+f2);  SimpleFeature bigLocation = locationPool.getFeature(f1);  SimpleFeature smallLocation = locationPool.getFeature(f2);  Geometry bigGeometry = (Geometry) bigLocation.getDefaultGeometry();  Geometry smallGeometry = (Geometry) smallLocation.getDefaultGeometry();  **return** bigGeometry.equals(smallGeometry);  }    **public** Boolean disjoint(String f1, String f2) **throws** Exception {  log("disjoint"+" "+f1+" "+f2);  SimpleFeature bigLocation = locationPool.getFeature(f1);  SimpleFeature smallLocation = locationPool.getFeature(f2);  Geometry bigGeometry = (Geometry) bigLocation.getDefaultGeometry();  Geometry smallGeometry = (Geometry) smallLocation.getDefaultGeometry();  **return** bigGeometry.disjoint(smallGeometry);  }  **public** Boolean intersects(String f1, String f2) **throws** Exception {  log("intersects"+" "+f1+" "+f2);  SimpleFeature bigLocation = locationPool.getFeature(f1);  SimpleFeature smallLocation = locationPool.getFeature(f2);  Geometry bigGeometry = (Geometry) bigLocation.getDefaultGeometry();  Geometry smallGeometry = (Geometry) smallLocation.getDefaultGeometry();  **return** bigGeometry.intersects(smallGeometry);  }    **public** Boolean touches(String f1, String f2) **throws** Exception {  log("touches"+" "+f1+" "+f2);  SimpleFeature bigLocation = locationPool.getFeature(f1);  SimpleFeature smallLocation = locationPool.getFeature(f2);  Geometry bigGeometry = (Geometry) bigLocation.getDefaultGeometry();  Geometry smallGeometry = (Geometry) smallLocation.getDefaultGeometry();  **return** bigGeometry.touches(smallGeometry);  }    **public** Boolean crosses(String f1, String f2) **throws** Exception {  log("crosses"+" "+f1+" "+f2);  SimpleFeature bigLocation = locationPool.getFeature(f1);  SimpleFeature smallLocation = locationPool.getFeature(f2);  Geometry bigGeometry = (Geometry) bigLocation.getDefaultGeometry();  Geometry smallGeometry = (Geometry) smallLocation.getDefaultGeometry();  **return** bigGeometry.crosses(smallGeometry);  }    **public** Boolean within(String f1, String f2) **throws** Exception {  log("within"+" "+f1+" "+f2);  SimpleFeature bigLocation = locationPool.getFeature(f1);  SimpleFeature smallLocation = locationPool.getFeature(f2);  Geometry bigGeometry = (Geometry) bigLocation.getDefaultGeometry();  Geometry smallGeometry = (Geometry) smallLocation.getDefaultGeometry();  **return** bigGeometry.within(smallGeometry);  }  **public** Boolean contains(String f1, String f2) **throws** Exception {  log("contains"+" "+f1+" "+f2);  SimpleFeature bigLocation = locationPool.getFeature(f1);  SimpleFeature smallLocation = locationPool.getFeature(f2);  Geometry bigGeometry = (Geometry) bigLocation.getDefaultGeometry();  Geometry smallGeometry = (Geometry) smallLocation.getDefaultGeometry();  **return** bigGeometry.contains(smallGeometry);  }    **public** Boolean overlaps(String f1, String f2) **throws** Exception {  log("overlaps"+" "+f1+" "+f2);  SimpleFeature bigLocation = locationPool.getFeature(f1);  SimpleFeature smallLocation = locationPool.getFeature(f2);  Geometry bigGeometry = (Geometry) bigLocation.getDefaultGeometry();  Geometry smallGeometry = (Geometry) smallLocation.getDefaultGeometry();  **return** bigGeometry.overlaps(smallGeometry);  }    **public** String buffer(String feature, Integer distance) **throws** Exception {  log("buffer"+" "+feature+" "+distance);  SimpleFeature location = locationPool.getFeature(feature);  Geometry geometry = (Geometry) location.getDefaultGeometry();  Polygon b = (Polygon) geometry.buffer(distance);  SimpleFeatureType bufferType =   DataUtilities.*createType*("buffer", "geom:Polygon");  SimpleFeature buffer =   SimpleFeatureBuilder.*build*(bufferType, **new** Object[] { b }, **null**);  locationPool.addFeature(feature+"\_buffer", "\_\_temp", buffer);  **return** feature+"\_buffer";  }    **public** Boolean shortestRoute(String startFeature, String endFeature)   **throws** Exception {  log("shortestRoute"+" "+startFeature+" "+endFeature);  String endFeatureType = locationPool.getFeatureType(endFeature);  Graph graph = GraphUtils.*createGraphFromLocationPool*(locationPool);  Set<String> endFeatureNames = locationPool.getFeatures(endFeatureType);  **double** bl = -1d;  String bf = **null**;  **for** (String candidateEndFeature : endFeatureNames) {  Path path = GraphUtils.*getShortestRoute*(  graph.get(startFeature),  graph.get(candidateEndFeature),  graph.getGenerator().getGraph());  **double** l = 0d;  **for** (Object ed : path.getEdges()) {  SimpleFeature feature = (SimpleFeature) ((Graphable) ed).getObject();  Geometry geometry = (Geometry) feature.getDefaultGeometry();  l += geometry.getLength();  }  **if** (bf == **null** || l < bl) {  bl = l;  bf = candidateEndFeature;  }    }  **return** endFeature.equals(bf);  }    } | *Sadly, CPN Tools is unable to deal with array or Collection types. As such, we provide functionality for returning lists of features by first concatenating feature identifiers with a separator character. A splitting function in CPN Tools can then split the string up to a list of features.*  *The structureName() method provides an identifier to CPN Tools which will be used as the prefix in RPC calls, e.g. “LA.equals(…)”.*  *Assign feature methods. These methods are used to capture the binding of a location to a location-dependent transition in the running process model.*  *This is useful if the GIS system needs to react to the assignment of features.*  *We hard-code some custom logic here to keep track of the last customer location and to change the status of repair teams.*  *Feature type and attribute functions.*  *Geometric relationships.*  *Returns true if the endFeature lies on the shortest route from startFeature compared to other features of the same type as endFeature.* |

# File: “src/simulator/extensions/location/GISSystem.java”

This class sets up a background task which will randomly move repair teams around or move repair teams to their destination if they are on route. This is a proof of concept and can naturally be integrated with existing systems.

|  |  |
| --- | --- |
| **package** simulator.extensions.location;  **import** java.awt.Color;  **import** java.util.List;  **import** java.util.Random;  **import** java.util.Timer;  **import** java.util.TimerTask;  **import** org.geotools.feature.DefaultFeatureCollection;  **import** org.geotools.graph.path.Path;  **import** org.geotools.graph.structure.Edge;  **import** org.geotools.graph.structure.Graphable;  **import** org.geotools.graph.structure.Node;  **import** org.geotools.map.FeatureLayer;  **import** org.geotools.map.event.MapLayerEvent;  **import** org.geotools.map.event.MapLayerListEvent;  **import** org.geotools.styling.Style;  **import** org.opengis.feature.simple.SimpleFeature;  **import** simulator.extensions.location.model.Graph;  **import** simulator.extensions.location.ui.LogWindow;  **import** simulator.extensions.location.ui.MapWindow;  **import** simulator.extensions.location.utils.GraphUtils;  **import** simulator.extensions.location.utils.MappingUtils;  **import** simulator.extensions.location.utils.StyleUtils;  **import** simulator.extensions.location.utils.InitUtils.FeatureTypes;  **import** com.vividsolutions.jts.geom.Point;  **public** **class** GISSystem {  **private** RPCHandler rpcHandler;  **private** LogWindow logWindow;  **private** MapWindow mapWindow;  **private** Graph graph;    **private** **boolean** emptyRoutes = **true**;    **private** **static** Style *style* = StyleUtils.*createLineStyle*(Color.*RED*, 4);  **public** GISSystem(RPCHandler rpcHandler, LogWindow logWindow,   MapWindow mapWindow, Graph graph) {  **this**.rpcHandler = rpcHandler;  **this**.logWindow = logWindow;  **this**.mapWindow = mapWindow;  **this**.graph = graph;  }    **public** **synchronized** **void** notifyLayerChange(**int** index) {  MapLayerEvent mple = **new** MapLayerEvent(MappingUtils.*LAYERS*[index],  MapLayerEvent.*DATA\_CHANGED*);  MapLayerListEvent mplle = **new** MapLayerListEvent(mapWindow.getContent(),  MappingUtils.*LAYERS*[index],  mapWindow.getContent().layers().indexOf(  MappingUtils.*LAYERS*[index]), mple);  mapWindow.getMapFrame().getMapPane().layerChanged(mplle);  mapWindow.getMapFrame().getMapPane().repaint();  }    **public** **synchronized** **void** swapOutLayer(**int** index, FeatureLayer newLayer) {  mapWindow.getContent().removeLayer(MappingUtils.*LAYERS*[index]);  MappingUtils.*LAYERS*[index] = newLayer;  mapWindow.getContent().addLayer(MappingUtils.*LAYERS*[index]);  }    **public** **void** startPoller() {  Timer moveTeamsTimer = **new** Timer(**true**);  moveTeamsTimer.scheduleAtFixedRate(**new** TimerTask() {  **public** **void** run() {  DefaultFeatureCollection routes =   **new** DefaultFeatureCollection(**null**, **null**);  **for** (String fname : rpcHandler.getLocationPool().getFeatures(  FeatureTypes.*RepairTeam*.name())) {  SimpleFeature feature =   rpcHandler.getLocationPool().getFeature(fname);  **if** (feature.getAttribute("status").equals("available")) {  Node current = graph.get(fname);  @SuppressWarnings("unchecked")  List<Edge> outs = current.getEdges();  Random r = **new** Random();  Node newnode =   outs.get(r.nextInt(outs.size())).getOtherNode(current);  graph.put(fname, newnode);  Point p = ((Point) ((Graphable) newnode).getObject());  rpcHandler.getLocationPool().  getFeature(fname).setDefaultGeometry(p);  }  **if** (feature.getAttribute("status").equals("onroute")) {  Node current = graph.get(fname);  Node dest =  graph.get(feature.getAttribute("destination").toString());  **if** (current.equals(dest)) {  feature.setAttribute("status", "onsite");  logWindow.showInfo("Repair team "+fname+" is onsite");  } **else** {  Path shortest = GraphUtils.*getShortestRoute*(  current, dest, graph.getGenerator().getGraph());  List<?> edges = shortest.getEdges();  **for** (**int** i = 0; i < edges.size() - 1; i++) {  SimpleFeature edgeFeature =   (SimpleFeature) ((Graphable) edges.get(i)).getObject();  routes.add(edgeFeature);  }  Node newnode = (Node) shortest.get(shortest.size()-2);  graph.put(fname, newnode);  Point p = ((Point) ((Graphable) newnode).getObject());  rpcHandler.getLocationPool().  getFeature(fname).setDefaultGeometry(p);  logWindow.showInfo("Repair team "+fname+" is onroute to "+dest);  }  }  }  **if** ((routes.size() > 0)   || (routes.size() == 0 && emptyRoutes == **true**)) {  swapOutLayer(MappingUtils.*LAYER\_ROUTE*,   **new** FeatureLayer(routes, *style*));  notifyLayerChange(MappingUtils.*LAYER\_ROUTE*);  }  emptyRoutes = routes.size() > 0;  notifyLayerChange(MappingUtils.*LAYER\_REPAIRTEAMS*);  }  }, 0, 1 \* 1000);  }  } | *Used to force the repaint of a map layer when changed.*  *Used to swap out an existing map layer with an updated one.*  *For available repair teams, just move then to a random neighbour node.*  *For repair teams on route to a customer, bring then one node closer to the customer, following the shortest route.*  *Show the routes when necessary.*  *Execute this once every second.*  *Note that the current implementation does not deal with the differences in distance between neighbouring nodes. We assume that teams are able to travel to a neighbour node once every time unit, here compressed to one second for the sake of the experiment.* |

# File: “src/simulator/extensions/location/model/LocationPool.java”

This class represents all the features in the system, together with their type. The class also provides functionality for retrieving a GeoTools feature representation based on the String identifier. In essence, this provides a mapping between the String-based feature / feature type and GeoTools’ feature / feature type systems.

|  |  |
| --- | --- |
| **package** simulator.extensions.location.model;  **import** java.util.HashMap;  **import** java.util.HashSet;  **import** java.util.Map;  **import** java.util.Map.Entry;  **import** java.util.Set;  **import** org.opengis.feature.simple.SimpleFeature;  **public** **class** LocationPool {  **private** **final** Set<String> featureTypes = **new** HashSet<String>();  **private** **final** Map<String, SimpleFeature> features =   **new** HashMap<String, SimpleFeature>();  **private** **final** Map<String, String> featureTypeMap =   **new** HashMap<String, String>();    **public** **boolean** isFeatureType(String typeName) {  **return** featureTypes.contains(typeName);  }    **public** **boolean** isFeature(String name) {  **return** features.containsKey(name);  }    **public** **void** addFeature(String name, String typeName,   SimpleFeature feature) {  featureTypes.add(typeName);  features.put(name, feature);  featureTypeMap.put(name, typeName);  }    **public** **void** removeFeature(String name) {  features.remove(name);  featureTypeMap.remove(name);  }    **public** **void** addFeatureType(String typeName) {  featureTypes.add(typeName);  }    **public** **void** removeFeatureType(String typeName) {  featureTypes.remove(typeName);  Set<Entry<String, String>> entrySet = featureTypeMap.entrySet();  **for** (Entry<String, String> e : entrySet) {  **if** (e.getValue().equals(typeName)) {  features.remove(e.getKey());  featureTypeMap.remove(e.getKey());  }  }  }    **public** Set<String> getFeatureTypes() {  **return** **new** HashSet<String>(featureTypes);  }    **public** Set<String> getFeatures() {  **return** **new** HashSet<String>(features.keySet());  }    **public** Set<String> getFeatures(String typeName) {  Set<String> features = **new** HashSet<String>();  **for** (Entry<String, String> e : featureTypeMap.entrySet()) {  **if** (e.getValue().equals(typeName)) {  features.add(e.getKey());  }  }  **return** features;  }    **public** SimpleFeature getFeature(String name) {  **return** features.get(name);  }  **public** String getFeatureType(String name) {  **return** featureTypeMap.get(name);  }  } | *This method returns the GeoTools feature representation (SimpleFeature) for a feature identifier (a String).* |

# File: “src/simulator/extensions/location/model/Graph.java”

This class provides a simple mapping between a feature identifier (a String) and nodes in a GeoTools FeatureGraph. This allows to retrieve graph nodes for particular features.

|  |  |
| --- | --- |
| **package** simulator.extensions.location.model;  **import** java.util.HashMap;  **import** java.util.Map;  **import** org.geotools.graph.build.feature.FeatureGraphGenerator;  **import** org.geotools.graph.build.line.LineStringGraphGenerator;  **import** org.geotools.graph.structure.Node;  **public** **class** Graph {  **private** **final** FeatureGraphGenerator generator;  **private** **final** Map<String, Node> featureNodeMap = **new** HashMap<String, Node>();    **public** Graph() {  LineStringGraphGenerator lineStringGenerator =   **new** LineStringGraphGenerator();  generator = **new** FeatureGraphGenerator(lineStringGenerator);  }  **public** FeatureGraphGenerator getGenerator() {  **return** generator;  }    **public** **void** put(String name, Node node) {  featureNodeMap.put(name, node);  }    **public** Node get(String name) {  **return** featureNodeMap.get(name);  }    **public** **void** remove(String name) {  featureNodeMap.remove(name);  }  } |  |

# File: “src/simulator/extensions/location/utils/InitUtils.java”

This utility class provides static methods to load in all shape data and initialize the location pool.

|  |  |
| --- | --- |
| **package** simulator.extensions.location.utils;  **import** java.io.File;  **import** java.io.IOException;  **import** java.util.ArrayList;  **import** java.util.HashMap;  **import** java.util.List;  **import** java.util.Map;  **import** java.util.Random;  **import** java.util.Set;  **import** org.geotools.data.simple.SimpleFeatureCollection;  **import** org.geotools.data.simple.SimpleFeatureSource;  **import** org.geotools.feature.FeatureIterator;  **import** org.geotools.feature.simple.SimpleFeatureBuilder;  **import** org.geotools.feature.simple.SimpleFeatureTypeBuilder;  **import** org.geotools.geometry.jts.JTSFactoryFinder;  **import** org.geotools.referencing.crs.DefaultGeographicCRS;  **import** org.opengis.feature.simple.SimpleFeature;  **import** org.opengis.feature.simple.SimpleFeatureType;  **import** simulator.extensions.location.model.LocationPool;  **import** simulator.extensions.location.ui.LogWindow;  **import** com.vividsolutions.jts.geom.Coordinate;  **import** com.vividsolutions.jts.geom.GeometryFactory;  **import** com.vividsolutions.jts.geom.MultiPolygon;  **import** com.vividsolutions.jts.geom.Point;  **public** **class** InitUtils {    **public** **enum** FeatureTypes   {*Road*, *Region*, *CallCenter*, *Customer*, *RepairStation*, *RepairTeam*};    **private** **static** File *lineFile* =  **new** File("data/callcenter1/line/line\_Shapefile.shp");  **private** **static** File *polyFile* =   **new** File("data/callcenter1/polygon/polygon\_Shapefile.shp");  **private** **static** File *pointFile* =   **new** File("data/callcenter1/point/point2\_Shapefile.shp");    **public** **static** **void** initialize(LocationPool pool, LogWindow logWindow)   **throws** IOException {  logWindow.showInfo(*lineFile*.toURI().toURL().toString());  logWindow.showInfo(*polyFile*.toURI().toURL().toString());  logWindow.showInfo(*pointFile*.toURI().toURL().toString());  **if** (!*lineFile*.exists() || !*polyFile*.exists() || !*pointFile*.exists())  logWindow.showInfo("\n!!! SOME FILES DO NOT EXIST !!!\n");  *initializeRoads*(pool, logWindow);  *initializeRegions*(pool, logWindow);  *initializeRepairStations*(pool, logWindow);  *initializeCustomers*(pool, logWindow);  *initializeCallCenters*(pool, logWindow);  *initializeRepairTeams*(pool, logWindow);  }  **private** **static** **void** initializeRoads(LocationPool pool, LogWindow logWindow)  **throws** IOException {  logWindow.showInfo(" (init roads)");  SimpleFeatureSource featureSource =   LoadingUtils.*getFeatureSourceFromShapefile*(*lineFile*);  SimpleFeatureCollection features = featureSource.getFeatures();  FeatureIterator<SimpleFeature> iterator = features.features();  **int** i = 0;  **while** (iterator.hasNext()) {  SimpleFeature feature = iterator.next();  pool.addFeature("Road" + ++i, FeatureTypes.*Road*.name(), feature);  }  logWindow.showInfo(pool.getFeatures(  FeatureTypes.*Road*.name()).size() + " roads parsed");  }    **private** **static** **void** initializeRegions(LocationPool pool,   LogWindow logWindow) **throws** IOException {  logWindow.showInfo(" (init regions)");  SimpleFeatureSource featureSource =   LoadingUtils.*getFeatureSourceFromShapefile*(*polyFile*);  SimpleFeatureCollection features = featureSource.getFeatures();  FeatureIterator<SimpleFeature> iterator = features.features();  **int** i = 0;  **while** (iterator.hasNext()) {  SimpleFeature feature = iterator.next();  String name = "Region" + ++i;  pool.addFeature(name, FeatureTypes.*Region*.name(), feature);  }  logWindow.showInfo(pool.getFeatures(  FeatureTypes.*Region*.name()).size() + " regions parsed");  }    **private** **static** **void** initializeRepairStations(LocationPool pool,   LogWindow logWindow) **throws** IOException {  logWindow.showInfo(" (init stations)");  Map<String, Class<?>> atts = **new** HashMap<String, Class<?>>(){  **private** **static** **final** **long** *serialVersionUID* = 1L;  {  **this**.put("region", String.**class**);  }};  SimpleFeatureSource featureSource =  LoadingUtils.*getFeatureSourceFromShapefile*(*pointFile*);  SimpleFeatureCollection features = featureSource.getFeatures();  FeatureIterator<SimpleFeature> iterator = features.features();  SimpleFeatureBuilder sfb = **new** SimpleFeatureBuilder(  LoadingUtils.*getExtendedFeatureTypeFromFeatureSource*(  "RepairStation", atts, featureSource));  **int** i = 0;  **while** (iterator.hasNext()) {  SimpleFeature feature = iterator.next();  **if** (!feature.getAttribute("Name").toString().startsWith("维修站"))  **continue**;  String name = "RepairStation" + ++i;  sfb.addAll(feature.getAttributes());  sfb.add(*getRegion*(feature, pool));  SimpleFeature extendedFeature = sfb.buildFeature(name);  extendedFeature.setAttribute("Name", name);  pool.addFeature(  name,  FeatureTypes.*RepairStation*.name(),  extendedFeature);  }  logWindow.showInfo(pool.getFeatures(  FeatureTypes.*RepairStation*.name()).size() + " repair stations parsed");  }    **private** **static** **void** initializeCustomers(LocationPool pool,   LogWindow logWindow) **throws** IOException {  logWindow.showInfo(" (init customers)");  Map<String, Class<?>> atts = **new** HashMap<String, Class<?>>(){  **private** **static** **final** **long** *serialVersionUID* = 1L;  {  **this**.put("region", String.**class**);  }};  SimpleFeatureSource featureSource =   LoadingUtils.*getFeatureSourceFromShapefile*(*pointFile*);  SimpleFeatureCollection features = featureSource.getFeatures();  FeatureIterator<SimpleFeature> iterator = features.features();  SimpleFeatureBuilder sfb = **new** SimpleFeatureBuilder(  LoadingUtils.*getExtendedFeatureTypeFromFeatureSource*(  "Customer", atts, featureSource));  **int** i = 0;  **while** (iterator.hasNext()) {  SimpleFeature feature = (SimpleFeature) iterator.next();  **if** (!feature.getAttribute("Name").toString().startsWith("客户"))  **continue**;  String name = "Customer" + ++i;  sfb.addAll(feature.getAttributes());  sfb.add(*getRegion*(feature, pool));  SimpleFeature extendedFeature = sfb.buildFeature(name);  extendedFeature.setAttribute("Name", name);  pool.addFeature(  name,  FeatureTypes.*Customer*.name(),  extendedFeature);  }  logWindow.showInfo(pool.getFeatures(  FeatureTypes.*Customer*.name()).size() + " customers parsed");  }    **private** **static** **void** initializeCallCenters(LocationPool pool,   LogWindow logWindow) **throws** IOException {  logWindow.showInfo(" (init centers)");  SimpleFeatureTypeBuilder sftb = **new** SimpleFeatureTypeBuilder();  sftb.setName("Call Center");  sftb.setCRS(DefaultGeographicCRS.*WGS84*);  sftb.add("Name", String.**class**);  sftb.add("the\_geom", Point.**class**);  sftb.add("region", String.**class**);  SimpleFeatureType callCenterType = sftb.buildFeatureType();  GeometryFactory geometryFactory =   JTSFactoryFinder.*getGeometryFactory*(**null**);  **int** csx[] = **new** **int**[]{-98, -129, 0, 0, 88, 74};  **int** csy[] = **new** **int**[]{125, 25, 100, -40, 132, 53};  **for** (**int** x = 0; x < csx.length; x++) {  String name = "CallCenter" + (x + 1);  Point point = geometryFactory.createPoint(  **new** Coordinate(csx[x], csy[x]));  SimpleFeatureBuilder builder = **new** SimpleFeatureBuilder(callCenterType);  builder.add(name);  builder.add(point);  builder.add("");  SimpleFeature feature = builder.buildFeature(name);  feature.setAttribute("region", *getRegion*(feature, pool));  pool.addFeature(  name,  FeatureTypes.*CallCenter*.name(),  feature);  }  logWindow.showInfo(pool.getFeatures(  FeatureTypes.*CallCenter*.name()).size() + " call centers parsed");  }    **private** **static** **void** initializeRepairTeams(LocationPool pool,   LogWindow logWindow) **throws** IOException {  logWindow.showInfo(" (init teams)");  SimpleFeatureTypeBuilder sftb = **new** SimpleFeatureTypeBuilder();  sftb.setName("Repair Team");  sftb.setCRS(DefaultGeographicCRS.*WGS84*);  sftb.add("Name", String.**class**);  sftb.add("the\_geom", Point.**class**);  sftb.add("status", String.**class**);  sftb.add("destination", String.**class**);  SimpleFeatureType repairTeamType = sftb.buildFeatureType();  GeometryFactory geometryFactory =   JTSFactoryFinder.*getGeometryFactory*(**null**);  List<String> repairStations =   **new** ArrayList<String>(pool.getFeatures(  FeatureTypes.*RepairStation*.name()));  Random random = **new** Random();  **for** (**int** x = 0; x < 5; x++) {  String name = "RepairTeam" + (x + 1);  Point station = (Point) pool.getFeature(  repairStations.get(random.nextInt(  repairStations.size()))).getDefaultGeometry();  Point point = geometryFactory.createPoint(  **new** Coordinate(station.getX(), station.getY()));  SimpleFeatureBuilder builder = **new** SimpleFeatureBuilder(repairTeamType);  builder.add(name);  builder.add(point);  builder.add("available");  builder.add("");  SimpleFeature feature = builder.buildFeature(name);  pool.addFeature(  name,  FeatureTypes.*RepairTeam*.name(),  feature);  }  logWindow.showInfo(pool.getFeatures(  FeatureTypes.*RepairTeam*.name()).size() + " repair teams parsed");  }    **private** **static** String getRegion(SimpleFeature feature, LocationPool pool) {  Set<String> regions = pool.getFeatures(FeatureTypes.*Region*.name());  Point pointGeom = (Point) feature.getDefaultGeometry();    **for** (String region : regions) {  SimpleFeature regionFeature = pool.getFeature(region);  MultiPolygon regionGeom = (MultiPolygon)  regionFeature.getDefaultGeometry();  **if** (regionGeom.contains(pointGeom)) **return** region;  }    **return** **null**;  }  } | *Roads are read from a shape file.*  *Regions are read from a shape file.*  *Repair stations are read from a shape file.*  *Only read features starting with维修站 (a repair station).*  *Add a “region” attribute.*  *Customers also read from shape file, a region attribute is also added.*  *Call centers are created in hard-coded manner, as they are not included in shape file. This method shows how to manually add features.*  *Five repair teams are initialized randomly.*  *They get status and destination attributes.*  *Get the region for a feature based on GeoTools contains() method. This method is used to assign region attributes to features, which point to other feature identifiers.* |

# File: “src/simulator/extensions/location/utils/LoadingUtils.java”

This utility class contains static functions to extract data from shape files and to get a GeoTools feature type from a feature source.

|  |  |
| --- | --- |
| **package** simulator.extensions.location.utils;  **import** java.io.File;  **import** java.io.IOException;  **import** java.io.Serializable;  **import** java.net.MalformedURLException;  **import** java.nio.charset.Charset;  **import** java.util.HashMap;  **import** java.util.Map;  **import** java.util.Map.Entry;  **import** org.geotools.data.shapefile.ShapefileDataStore;  **import** org.geotools.data.shapefile.ShapefileDataStoreFactory;  **import** org.geotools.data.simple.SimpleFeatureSource;  **import** org.geotools.feature.DefaultFeatureCollection;  **import** org.geotools.feature.simple.SimpleFeatureTypeBuilder;  **import** org.opengis.feature.simple.SimpleFeatureType;  **import** simulator.extensions.location.model.LocationPool;  **public** **class** LoadingUtils {  **public** **static** SimpleFeatureType getExtendedFeatureTypeFromFeatureSource(  String name, Map<String, Class<?>> attributes,  SimpleFeatureSource featureSource) {  SimpleFeatureType sft = featureSource.getSchema();  SimpleFeatureTypeBuilder stb = **new** SimpleFeatureTypeBuilder();  stb.init(sft);  stb.setName(name);  **for** (Entry<String, Class<?>> e : attributes.entrySet()) {  stb.add(e.getKey(), e.getValue());  }  SimpleFeatureType newFeatureType = stb.buildFeatureType();  **return** newFeatureType;  }  **public** **static** SimpleFeatureSource getFeatureSourceFromShapefile(File file) {  **try** {  Map<String, Serializable> params = **new** HashMap<String, Serializable>();  params.put(ShapefileDataStoreFactory.*DBFCHARSET*.key, "GB18030");  params.put(ShapefileDataStoreFactory.*URLP*.key, file.toURI().toURL());  params.put(ShapefileDataStoreFactory.*CREATE\_SPATIAL\_INDEX*.key, **true**);  params.put(ShapefileDataStoreFactory.*ENABLE\_SPATIAL\_INDEX*.key, **true**);  params.put(ShapefileDataStoreFactory.*MEMORY\_MAPPED*.key, **false**);  //DataStore store = DataStoreFinder.getDataStore(params);  ShapefileDataStore store = **new** ShapefileDataStore(file.toURI().toURL());  store.setCharset(Charset.*forName*("GB18030"));  SimpleFeatureSource featureSource =  store.getFeatureSource(store.getTypeNames()[0]);  **return** featureSource;  } **catch** (MalformedURLException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  }  **return** **null**;  }    **public** **static** DefaultFeatureCollection  featureCollectionFromLocationPool(String typeName,  LocationPool pool) {  DefaultFeatureCollection features =  **new** DefaultFeatureCollection(**null**, **null**);  **for** (String name : pool.getFeatures(typeName)) {  features.add(pool.getFeature(name));  }  **return** features;  }  } | *Important: set the correct charset to read in Chinese.* |

# File: “src/simulator/extensions/location/utils/MappingUtils.java”

This utility class provides static methods to construct the map content for the map window, with the different layers.

|  |  |
| --- | --- |
| **package** simulator.extensions.location.utils;  **import** java.awt.Color;  **import** java.util.List;  **import** org.geotools.feature.DefaultFeatureCollection;  **import** org.geotools.map.FeatureLayer;  **import** org.geotools.map.Layer;  **import** org.geotools.map.MapContent;  **import** org.geotools.styling.Style;  **import** simulator.extensions.location.model.LocationPool;  **import** simulator.extensions.location.utils.InitUtils.FeatureTypes;  **public** **class** MappingUtils {    **public** **static** **final** **int** *LAYER\_REGIONS* = 0;  **public** **static** **final** **int** *LAYER\_CALLCENTERS* = 1;  **public** **static** **final** **int** *LAYER\_CUSTOMERS* = 2;  **public** **static** **final** **int** *LAYER\_REPAIRSTATIONS* = 3;  **public** **static** **final** **int** *LAYER\_REPAIRTEAMS* = 4;  **public** **static** **final** **int** *LAYER\_ROADS* = 5;  **public** **static** **final** **int** *LAYER\_ROUTE* = 6;  **public** **static** **final** FeatureLayer[] *LAYERS* = **new** FeatureLayer[7];    **public** **static** **void** clearMapContent(MapContent content) {  List<Layer> layers = content.layers();  **for** (Layer layer : layers) content.removeLayer(layer);  }    **public** **static** MapContent createMapContentFromLocationPool(  LocationPool pool) {  MapContent content = **new** MapContent();  *clearMapContent*(content);    Style style;  style = StyleUtils.*createPolygonStyle*(  Color.*RED*, 3, 1, **new** **float**[]{10, 10}, Color.*RED*, 0.1);  *LAYERS*[*LAYER\_REGIONS*] =  **new** FeatureLayer(LoadingUtils.*featureCollectionFromLocationPool*(  FeatureTypes.*Region*.name(), pool), style);  content.addLayer(*LAYERS*[*LAYER\_REGIONS*]);    style = StyleUtils.*createLineStyle*(Color.*ORANGE*, 8);  *LAYERS*[*LAYER\_ROADS*] =   **new** FeatureLayer(LoadingUtils.*featureCollectionFromLocationPool*(  FeatureTypes.*Road*.name(), pool), style);  content.addLayer(*LAYERS*[*LAYER\_ROADS*]);    style = StyleUtils.*createPointStyle*(Color.*GREEN*, Color.*GREEN*, 5, "Name");  *LAYERS*[*LAYER\_CALLCENTERS*] = **new** FeatureLayer(  LoadingUtils.*featureCollectionFromLocationPool*(  FeatureTypes.*CallCenter*.name(), pool), style);  content.addLayer(*LAYERS*[*LAYER\_CALLCENTERS*]);    style = StyleUtils.*createPointStyle*(Color.*BLUE*, Color.*BLUE*, 5, "Name");  *LAYERS*[*LAYER\_CUSTOMERS*] = **new** FeatureLayer(  LoadingUtils.*featureCollectionFromLocationPool*(  FeatureTypes.*Customer*.name(), pool), style);  content.addLayer(*LAYERS*[*LAYER\_CUSTOMERS*]);    style = StyleUtils.*createPointStyle*(  Color.*MAGENTA*, Color.*MAGENTA*, 5, "Name");  *LAYERS*[*LAYER\_REPAIRSTATIONS*] =   **new** FeatureLayer(LoadingUtils.*featureCollectionFromLocationPool*(  FeatureTypes.*RepairStation*.name(), pool), style);  content.addLayer(*LAYERS*[*LAYER\_REPAIRSTATIONS*]);    style = StyleUtils.*createPointStyle*(Color.*BLACK*, Color.*BLACK*, 4, **null**);  *LAYERS*[*LAYER\_REPAIRTEAMS*] =   **new** FeatureLayer(LoadingUtils.*featureCollectionFromLocationPool*(  FeatureTypes.*RepairTeam*.name(), pool), style);  content.addLayer(*LAYERS*[*LAYER\_REPAIRTEAMS*]);    style = StyleUtils.*createLineStyle*(Color.*RED*, 4);  *LAYERS*[*LAYER\_ROUTE*] = **new** FeatureLayer(  **new** DefaultFeatureCollection(**null**, **null**), style);  content.addLayer(*LAYERS*[*LAYER\_ROUTE*]);    **return** content;  }  } | *The LoadingUtils class is used to build a feature collection (used in feature layers) from a location pool (our internal representation).* |

# File: “src/simulator/extensions/location/utils/GraphUtils.java”

This utility class provides static methods for graph operations. These are mainly used for the calculation of shortest routes and connecting manually creating nodes to the graph in an optimal manner.

|  |  |
| --- | --- |
| **package** simulator.extensions.location.utils;  **import** java.awt.geom.Point2D;  **import** java.util.Collection;  **import** java.util.Set;  **import** org.geotools.data.DataUtilities;  **import** org.geotools.feature.SchemaException;  **import** org.geotools.feature.simple.SimpleFeatureBuilder;  **import** org.geotools.graph.build.feature.FeatureGraphGenerator;  **import** org.geotools.graph.path.DijkstraShortestPathFinder;  **import** org.geotools.graph.path.Path;  **import** org.geotools.graph.structure.Edge;  **import** org.geotools.graph.structure.Graphable;  **import** org.geotools.graph.structure.Node;  **import** org.geotools.graph.traverse.standard.DijkstraIterator;  **import** org.geotools.graph.traverse.standard.DijkstraIterator.EdgeWeighter;  **import** org.geotools.resources.geometry.ShapeUtilities;  **import** org.opengis.feature.simple.SimpleFeature;  **import** org.opengis.feature.simple.SimpleFeatureType;  **import** com.vividsolutions.jts.geom.Coordinate;  **import** com.vividsolutions.jts.geom.Geometry;  **import** com.vividsolutions.jts.geom.GeometryFactory;  **import** com.vividsolutions.jts.geom.LineSegment;  **import** com.vividsolutions.jts.geom.LineString;  **import** com.vividsolutions.jts.geom.MultiLineString;  **import** com.vividsolutions.jts.geom.Point;  **import** simulator.extensions.location.model.Graph;  **import** simulator.extensions.location.model.LocationPool;  **import** simulator.extensions.location.utils.InitUtils.FeatureTypes;  **public** **class** GraphUtils {  **public** **static** Graph createGraphFromLocationPool(LocationPool pool) {  Graph graph = **new** Graph();  *addLines*(pool.getFeatures(FeatureTypes.*Road*.name()), graph, pool);  @SuppressWarnings("unchecked")  Collection<Edge> edges = graph.getGenerator().getGraph().getEdges();  *addPoints*(pool.getFeatures(  FeatureTypes.*CallCenter*.name()), edges, graph, pool);  *addPoints*(pool.getFeatures(  FeatureTypes.*RepairStation*.name()), edges, graph, pool);  *addPoints*(pool.getFeatures(  FeatureTypes.*Customer*.name()), edges, graph, pool);    @SuppressWarnings("unchecked")  Collection<Node> nodes = graph.getGenerator().getGraph().getNodes();  **for** (String fname : pool.getFeatures(FeatureTypes.*RepairTeam*.name())) {  Node nearest = *getNearestGraphNode*(  nodes, (Point) pool.getFeature(fname).getDefaultGeometry());  graph.put(fname, nearest);  }    **return** graph;  }    **private** **static** **void** addLines(Set<String> multilines,   Graph graph, LocationPool pool) {  FeatureGraphGenerator graphGenerator = graph.getGenerator();  GeometryFactory gf = **new** GeometryFactory();  SimpleFeatureType lineType;  **try** {  lineType = DataUtilities.*createType*("lseg", "geom:LineString");  } **catch** (SchemaException e) {  **return**;  }    **for** (String fname : multilines) {  SimpleFeature feature = pool.getFeature(fname);  MultiLineString mls = (MultiLineString) feature.getDefaultGeometry();  LineString ls = (LineString) mls.getGeometryN(0);  **for** (**int** i = 0; i < ls.getNumPoints() - 1; i++) {  LineSegment lseg = **new** LineSegment(  ls.getCoordinateN(i), ls.getCoordinateN(i + 1));  SimpleFeature lineTypeFeature = SimpleFeatureBuilder.*build*(  lineType, **new** Object[] { lseg.toGeometry(gf) }, **null**);  graphGenerator.add(lineTypeFeature);  }  }  }    **private** **static** **void** addPoints(Set<String> points,   Collection<Edge> edges, Graph graph, LocationPool pool) {  FeatureGraphGenerator graphGenerator = graph.getGenerator();  GeometryFactory gf = **new** GeometryFactory();  SimpleFeatureType lineType;  **try** {  lineType = DataUtilities.*createType*("lseg", "geom:LineString");  } **catch** (SchemaException e) {  **return**;  }    **for** (String fname : points) {  SimpleFeature feature = pool.getFeature(fname);  Point point = (Point) feature.getDefaultGeometry();  Edge nearestEdge = *getNearestGraphEdge*(edges, point);  Point2D pointOnEdge = *getColinearPoint*(nearestEdge, point);  Coordinate pointOnEdgeCoords =   **new** Coordinate(pointOnEdge.getX(), pointOnEdge.getY());  graphGenerator.remove(nearestEdge.getObject());  LineSegment lsegX = **new** LineSegment(  pointOnEdgeCoords, point.getCoordinate());  LineSegment lsegA = **new** LineSegment(  ((Point) nearestEdge.getNodeA().getObject()).getCoordinate(),  pointOnEdgeCoords);  LineSegment lsegB = **new** LineSegment(  pointOnEdgeCoords,   ((Point) nearestEdge.getNodeB().getObject()).getCoordinate());    SimpleFeature lineTypeFeatureX = SimpleFeatureBuilder.*build*(  lineType, **new** Object[] { lsegX.toGeometry(gf) }, **null**);  SimpleFeature lineTypeFeatureA = SimpleFeatureBuilder.*build*(  lineType, **new** Object[] { lsegA.toGeometry(gf) }, **null**);  SimpleFeature lineTypeFeatureB = SimpleFeatureBuilder.*build*(  lineType, **new** Object[] { lsegB.toGeometry(gf) }, **null**);  graphGenerator.add(lineTypeFeatureX);  graphGenerator.add(lineTypeFeatureA);  graphGenerator.add(lineTypeFeatureB);  Node createdNode =   ((Edge) graphGenerator.get(lineTypeFeatureX)).getNodeB();  graph.put(fname, createdNode);  }  }    **public** **static** Path getShortestRoute(Node start, Node dest,  org.geotools.graph.structure.Graph graph) {  EdgeWeighter weighter = **new** DijkstraIterator.EdgeWeighter() {  **public** **double** getWeight(Edge e) {  SimpleFeature feature = (SimpleFeature) e.getObject();  Geometry geometry = (Geometry) feature.getDefaultGeometry();  **return** geometry.getLength();  }  };  DijkstraShortestPathFinder pf =   **new** DijkstraShortestPathFinder(graph, start, weighter);  pf.calculate();  Path path = pf.getPath(dest);  **return** path;  }    **public** **static** Point2D getColinearPoint(Edge e, Point p) {  SimpleFeature f = (SimpleFeature) (((Graphable) e).getObject());  LineString ls = (LineString) f.getDefaultGeometry();  Point2D p2d = ShapeUtilities.*nearestColinearPoint*(  ls.getCoordinateN(0).x, ls.getCoordinateN(0).y,  ls.getCoordinateN(1).x, ls.getCoordinateN(1).y,  p.getCoordinate().x, p.getCoordinate().y);  **return** p2d;  }    **public** **static** Edge getNearestGraphEdge(Collection<Edge> edges,   Point pointy) {  Edge nearestEdge = **null**;  **double** dist = -1;  **for** (Edge e : edges) {  Point2D p = *getColinearPoint*(e, pointy);  **double** newdist = *calculateEuclideanDistance*(  pointy.getX(), pointy.getY(),  p.getX(), p.getY());  **if** (dist < 0 || newdist < dist) {  dist = newdist;  nearestEdge = e;  }  }  **return** nearestEdge;  }    **public** **static** Node getNearestGraphNode(Collection<Node> nodes,   Point pointy) {  Node nearestNode = **null**;  **double** dist = -1;  **for** (Node n : nodes) {  Point p = ((Point) ((Graphable) n).getObject());  **double** newdist = *calculateEuclideanDistance*(  pointy.getX(), pointy.getY(),  p.getCoordinate().x, p.getCoordinate().y);  **if** (dist < 0 || newdist < dist) {  dist = newdist;  nearestNode = n;  }  }  **return** nearestNode;  }  **public** **static** **double** calculateEuclideanDistance(**double** xOrig, **double** yOrig,  **double** xDest, **double** yDest) {  **double** distance = Math.*sqrt*((xDest - xOrig) \* (xDest - xOrig) +  (yDest - yOrig) \* (yDest - yOrig));  **return** distance;  }  } | *Utility method to construct a graph from our example case location pool.*  *Add lines to the graph. These will become edges. New nodes will be created if necessary.*  *Add points to the graph. These will become nodes.*  *The point needs to be connected to the graph. To do so, we find the nearest existing edge to the point. This edge is cut in to across the colinear point. A third edge connects the new point to the colinear point.*  *Get shortest route.*  *Get nearest colinear point.*  *Get nearest graph edge based on Euclidean distance.*  *Get nearest graph node based on Euclidean distance.* |

# File: “src/simulator/extensions/location/utils/StyleUtils.java”

This utility class contains a number of utility functions to create layer styles.

|  |  |
| --- | --- |
| **package** simulator.extensions.location.utils;  **import** java.awt.Color;  **import** org.geotools.data.FeatureSource;  **import** org.geotools.factory.CommonFactoryFinder;  **import** org.geotools.styling.FeatureTypeStyle;  **import** org.geotools.styling.Fill;  **import** org.geotools.styling.Font;  **import** org.geotools.styling.Graphic;  **import** org.geotools.styling.LineSymbolizer;  **import** org.geotools.styling.Mark;  **import** org.geotools.styling.PointSymbolizer;  **import** org.geotools.styling.PolygonSymbolizer;  **import** org.geotools.styling.Rule;  **import** org.geotools.styling.Stroke;  **import** org.geotools.styling.Style;  **import** org.geotools.styling.StyleBuilder;  **import** org.geotools.styling.StyleFactory;  **import** org.geotools.styling.TextSymbolizer;  **import** org.opengis.feature.simple.SimpleFeatureType;  **import** org.opengis.filter.FilterFactory;  **import** com.vividsolutions.jts.geom.LineString;  **import** com.vividsolutions.jts.geom.MultiLineString;  **import** com.vividsolutions.jts.geom.MultiPolygon;  **import** com.vividsolutions.jts.geom.Polygon;  **public** **class** StyleUtils {  **private** **static** StyleFactory *styleFactory* =  CommonFactoryFinder.*getStyleFactory*(**null**);  **private** **static** FilterFactory *filterFactory* =  CommonFactoryFinder.*getFilterFactory*(**null**);    **public** **static** Style createStyle(FeatureSource<?, ?> featureSource) {  SimpleFeatureType schema = (SimpleFeatureType) featureSource.getSchema();  Class<?> geomType = schema.getGeometryDescriptor().getType().getBinding();  **if** (Polygon.**class**.isAssignableFrom(geomType)  || MultiPolygon.**class**.isAssignableFrom(geomType)) {  **return** *createPolygonStyle*(  Color.*RED*, 3, 1, **new** **float**[]{10, 10}, Color.*RED*, 0.1);  } **else** **if** (LineString.**class**.isAssignableFrom(geomType)   || MultiLineString.**class**.isAssignableFrom(geomType)) {  **return** *createLineStyle*(Color.*BLACK*, 3);  } **else** {  **return** *createPointStyle*(Color.*GREEN*, Color.*GREEN*, 10, **null**);  }  }  **public** **static** Style createPolygonStyle(Color strokeColor, **int** size1,   **double** salpha, **float**[] dashArray,  Color fillColor, **double** falpha) {  Stroke stroke = *styleFactory*.createStroke(  *filterFactory*.literal(strokeColor),  *filterFactory*.literal(size1),  *filterFactory*.literal(salpha));  stroke.setDashArray(dashArray);  Fill fill = *styleFactory*.createFill(  *filterFactory*.literal(fillColor),  *filterFactory*.literal(falpha));  PolygonSymbolizer sym = *styleFactory*.createPolygonSymbolizer(stroke,  fill, **null**);  Rule rule = *styleFactory*.createRule();  rule.symbolizers().add(sym);  FeatureTypeStyle fts = *styleFactory*.createFeatureTypeStyle(  **new** Rule[] { rule });  Style style = *styleFactory*.createStyle();  style.featureTypeStyles().add(fts);  **return** style;  }  **public** **static** Style createLineStyle(Color color, **int** size) {  Stroke stroke = *styleFactory*.createStroke(  *filterFactory*.literal(color), *filterFactory*.literal(size));  LineSymbolizer sym = *styleFactory*.createLineSymbolizer(stroke, **null**);  Rule rule = *styleFactory*.createRule();  rule.symbolizers().add(sym);  FeatureTypeStyle fts = *styleFactory* .createFeatureTypeStyle(  **new** Rule[] { rule });  Style style = *styleFactory*.createStyle();  style.featureTypeStyles().add(fts);  **return** style;  }  **public** **static** Style createPointStyle(Color stroke, Color fill,  **int** size, String attributeLabel) {  Graphic gr = *styleFactory*.createDefaultGraphic();  Mark mark = *styleFactory*.getCircleMark();  mark.setStroke(*styleFactory*.createStroke(  *filterFactory*.literal(stroke), *filterFactory*.literal(size)));  mark.setFill(*styleFactory*.createFill(*filterFactory*.literal(fill)));  gr.graphicalSymbols().clear();  gr.graphicalSymbols().add(mark);  gr.setSize(*filterFactory*.literal(10));    Rule rule = *styleFactory*.createRule();    PointSymbolizer sym = *styleFactory*.createPointSymbolizer(gr, **null**);  rule.symbolizers().add(sym);    **if** (attributeLabel != **null**) {  Font[] f = **new** Font[] { *styleFactory*.createFont(  *filterFactory*.literal("DejaVu Sans"),  *filterFactory*.literal(Font.Style.*NORMAL*),  *filterFactory*.literal(Font.Weight.*NORMAL*),  *filterFactory*.literal(24)) };  StyleBuilder sb = **new** StyleBuilder();  TextSymbolizer tsym = *styleFactory*.createTextSymbolizer(  *styleFactory*.createFill(*filterFactory*.literal(Color.*BLACK*)),  f,  sb.createHalo(Color.*WHITE*, 1d),  sb.attributeExpression(attributeLabel),  **null**,  **null**);  rule.symbolizers().add(tsym);  }    FeatureTypeStyle fts = *styleFactory*.createFeatureTypeStyle(  **new** Rule[] { rule });  Style style = *styleFactory*.createStyle();  style.featureTypeStyles().add(fts);  **return** style;  }  } | *Create an automatic style for various geometry types.* |

# File: “src/simulator/extensions/location/ui/LogWindow.java”

This class provides a logging window to show feedback messages while the extension is running. This replaces standard POSIX error output as that output channel is captured by CPN Tools and not shown.

|  |  |
| --- | --- |
| **package** simulator.extensions.location.ui;  **import** javax.swing.JFrame;  **import** javax.swing.JScrollPane;  **import** javax.swing.JTextArea;  **public** **class** LogWindow **extends** JFrame {  **private** **static** **final** **long** *serialVersionUID* = -5610807955258781126L;  **private** JTextArea textArea = **new** JTextArea();  **public** LogWindow() {  **super**("");  setSize(300, 300);  add(**new** JScrollPane(textArea));  setVisible(**true**);  }  **public** **void** showInfo(String data) {  textArea.append(data+"\r\n");  textArea.setCaretPosition(textArea.getText().length()-1);  **this**.validate();  }  **public** **void** showByte(**char** b) {  textArea.append(b+"");  textArea.setCaretPosition(textArea.getText().length()-1);  **this**.validate();  }  } |  |

# File: “src/simulator/extensions/location/ui/MapWindow.java”

This class provides a map window for showing a GeoTools map with various layers while the extension is running.

|  |  |
| --- | --- |
| **package** simulator.extensions.location.ui;  **import** org.geotools.map.MapContent;  **import** org.geotools.swing.JMapFrame;  **public** **class** MapWindow {    **private** **final** JMapFrame mapFrame;  **private** **final** MapContent content;  **public** MapWindow() {  **this**(**new** MapContent());  content.setTitle("Map Overview");  }    **public** MapWindow(MapContent content) {  **this**.content = content;  mapFrame = **new** JMapFrame();  mapFrame.setMapContent(content);  mapFrame.enableToolBar(**true**);  mapFrame.pack();  mapFrame.setSize(800, 600);  mapFrame.setVisible(**true**);  }    **public** JMapFrame getMapFrame() {  **return** mapFrame;  }    **public** MapContent getContent() {  **return** content;  }    } |  |