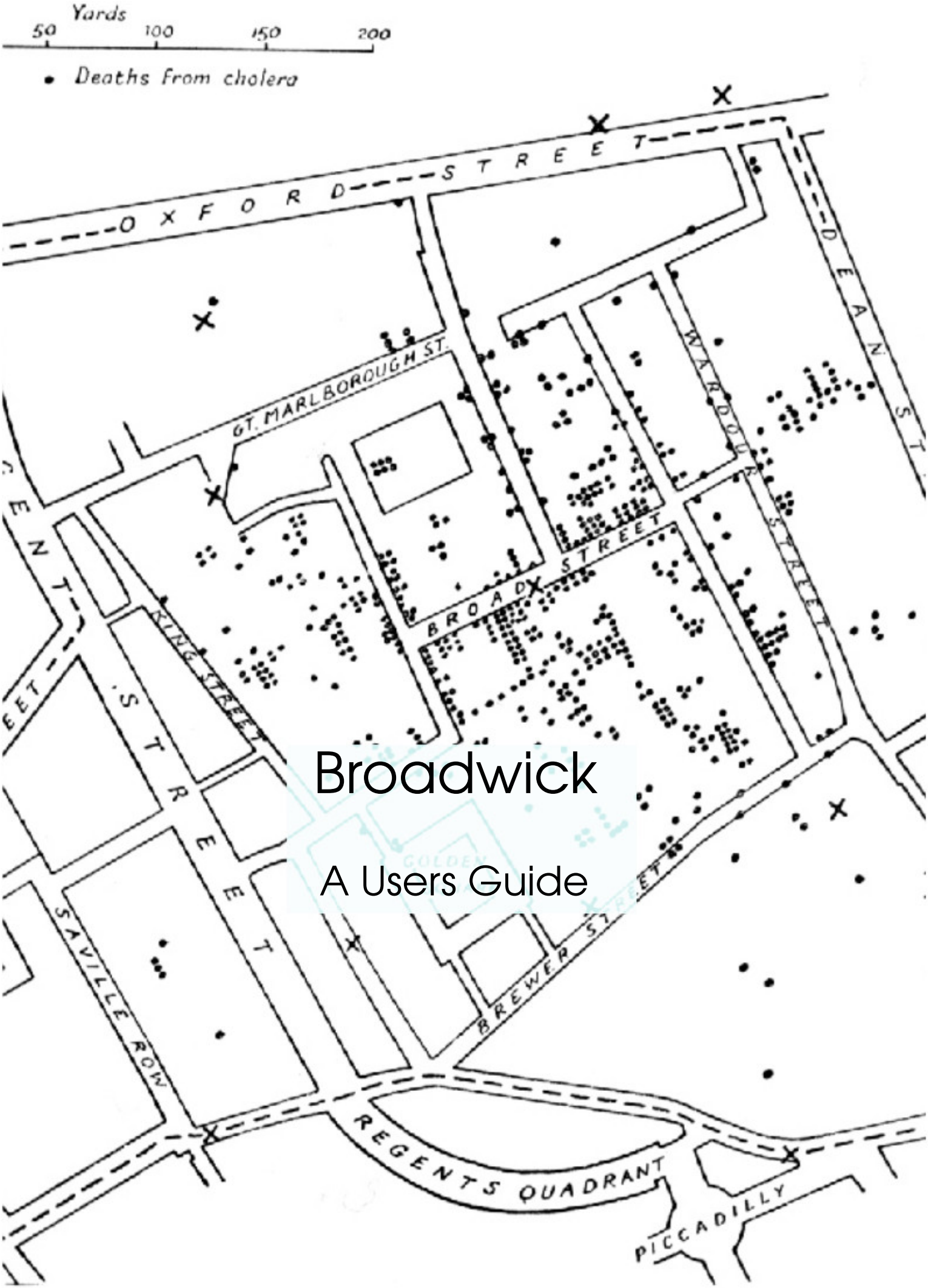
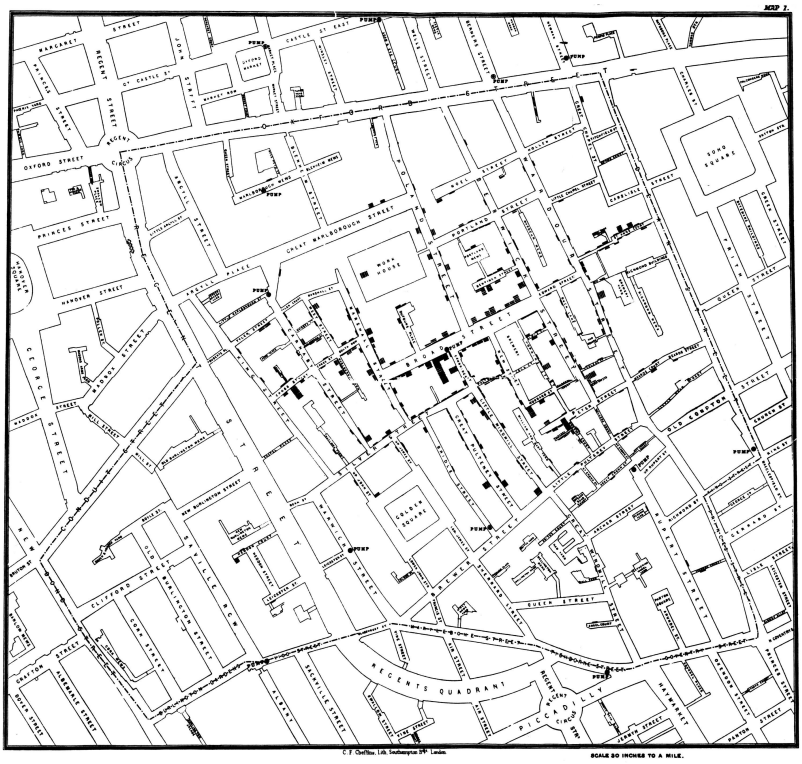


Broadwick

A Users Guide



Published by C.F. Cheffins, Lith, Southhampton Buildings, London, England, 1854 in Snow, John. On the Mode of Communication of Cholera, 2nd Ed, John Churchill, New Burlington Street, London, England, 1855. This image was originally from *http://en.wikipedia.org/wiki/File:Snow-cholera-map-1.jpg*

Cover Picture

A variant of the original map drawn by Dr. John Snow (1813-1858), a British physician who is one of the founders of medical epidemiology, showing cases of cholera in the London epidemics of 1854, clustered around the locations of water pumps.

HTTP://EPICSCOTLAND.GITHUB.IO/BROADWICK/

First printing, March 2013

Contents

1. Introduction

In the event of an outbreak it is important to have modelling tools in place to estimate the likely origin, speed of spread, and size, and to be able to predict the impact of intervention measures. However, different diseases in different animal populations require somewhat different approaches due to the detection, transmission and recovery (or not) characteristics of hosts infected with the pathogen, and the contact patterns between susceptible hosts (whether of the same species or not). Consequently specific models developed for one disease system are unlikely to be entirely suitable for another.

Broadwick is a framework for developing sophisticated epidemiological based mathematical models, and consists of several Java libraries and bespoke packages. The components of Broadwick are written in such a way that a scientist may combine them in order to rapidly prototype a model for a new specific scenario.

- Supports single (e.g. within herd) or structured populations (e.g. multi-species or locations)
- Inclusion of movement over network data (e.g. Cattle movement Tracing System)
- Stochastic Individual Based simulations (including fast approximate options)
- Approximate Bayesian Computation inference for estimating model parameters from data via simulations
- Markov Chain Monte Carlo inference for estimating model parameters from data

1.1 License

Broadwick is released under the Apache 2 license.

2. Using Broadwick

2.1 Creating a New Project

Broadwick contains a set of packages that can be used as required. The framework is designed to be flexible and does not place any requirement on the user on how to use the framework. It is possible to use the classes and packages of Broadwick without using the powerful framework, creating your own main() method and taking responsibility for reading data files and configuration items though this is not the recommended way of using Broadwick.

2.1.1 Using The Command Line

The Broadwick distribution contains a maven archetype for generating a skeleton project that contains all the configuration files, source code etc that is required to start a project based upon Broadwick. It uses apache maven as it's build tool. To generate a skeleton using this archetype on the command line (assuming that the broadwick-archetype jar is in your local repository)

```
mvn3 archetype:generate -DarchetypeGroupId=broadwick -DarchetypeArtifactId=broadwick-
archetype  -DarchetypeVersion=1.1 -DgroupId=broadwick.proj -DartifactId=StochasticSir
-Dversion=0.1  -Dpackage=broadwick.stochasticsir
```

The groupId (maven uses the group id to uniquely identify your project), artifactId (is the name of your generated jar file without a version), version (the version number for your generated project) and the package to which the generated source will be created can be changed by modifiying the -DgroupId, -DartifactId, -Dversion and -Dpackage arguments above.

2.1.2 Using Netbeans

It is possibly easier to create a project using Netbeans (a free IDE available form Oracle, the 'owners' of Java). Open the Netbeans IDE and select File->New Project and choose a Maven project and "Project from Archetype" from the list of projects (see figproj1).

Click Next and choose the latest version of the broadwick-archetype from the "Known Archetypes" (see fig 2.2). The version of the broadwick archetype corresponds to the version of Broadwick.

The projects details can be specified on the next screen (fig 2.3).

Clicking "Finish" will create the project (fig 2.4).

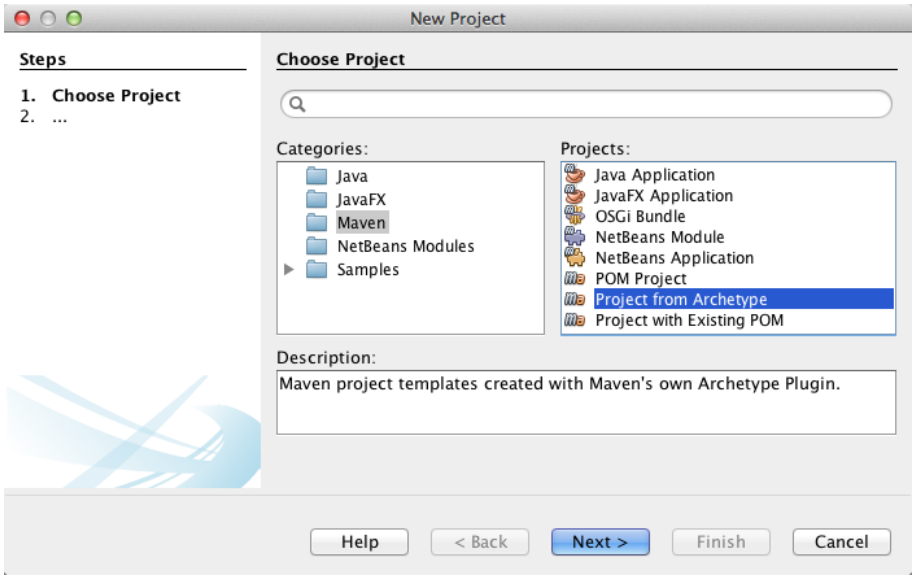


Figure 2.1: Creating a maven based project

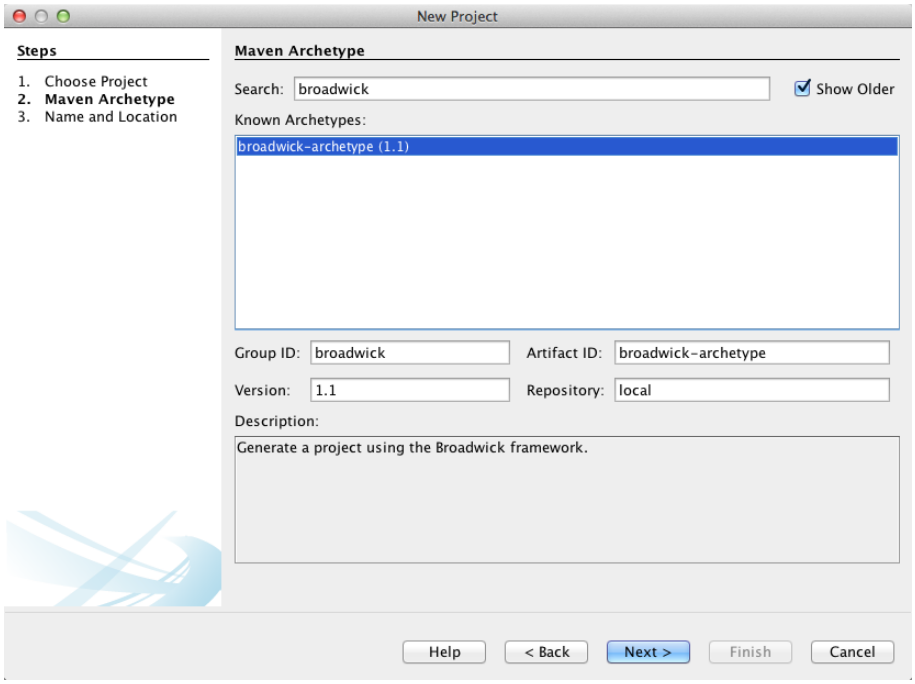


Figure 2.2: Using the Broadwick archetype to create a skeleton project.

A number of minor changes are needed in the generated project.

Select the name of the class (“App”) and right-click and select Refactor->Rename; Change the name of the class to StochasticSIR.

In the Broadwick.sh file change the ${artifactId} and ${version} to the artifactId and version specified when the project was created (StochasticSIR and 1.0 respectively). This is a shell script for running your project on Unix based systems, you will need to make it executable.

In Broadwick.xml (the configuration file for the generated project) change the name of the <classname> element to reflect the package and class (broadwick.stochasticsir.StocasticSIR)

We can build the generated project by selecting Run->Build Project from the menu bar or

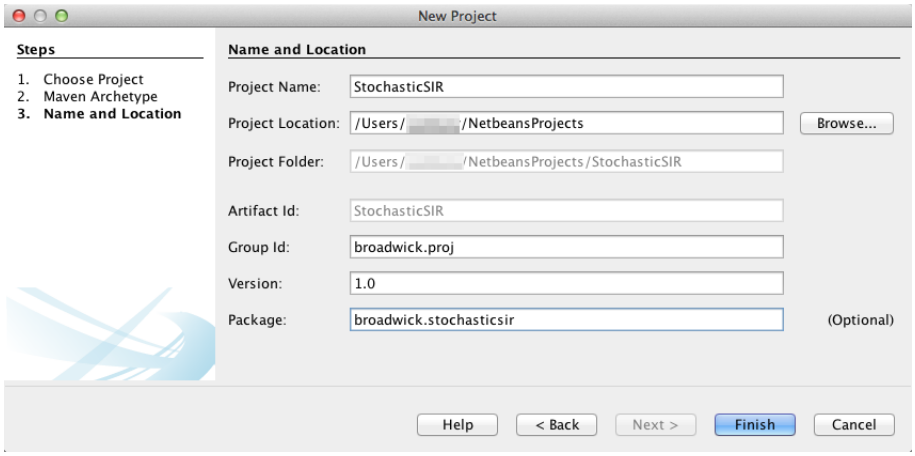


Figure 2.3: Setting project details for a Broadwick based project.

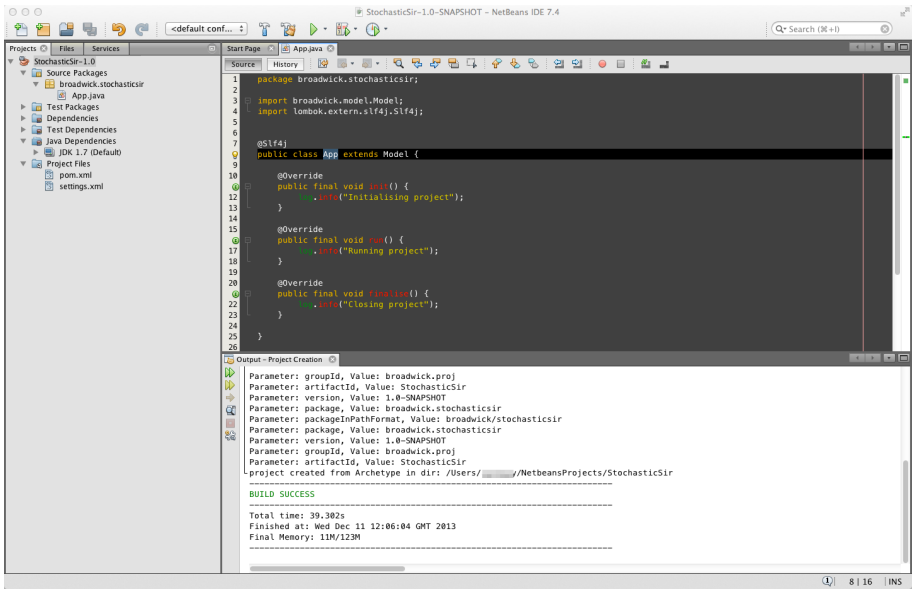


Figure 2.4: The generated project

clicking the icon in the toolbar (see fig 2.5 for some example output from this process). This will create a directory called target that contains, among other items, a jar file containing the compiled code and an executeable jar file ending in .one-jar.jar. The Broadwick.sh file is a shell script that will run the executable jar file on *NIX systems.

When Broadwick starts it looks for all the models specified in the <model> elements in the projects configuration file. It creates objects for each <model> found using the default (empty) constructor for the class given in the <classname> element of the model (this is why no constructor is generated for the project).

For each project object created Broadwick will call the init(), run() and finalise() methods in turn. In our skeleton project we simply logged the fact that these methods were called. A simplified outline of how Broadwick initialises itself is shown in fig 2.6.

A description of the configuration file is outlined in the next section.

Figure 2.5: Figure caption

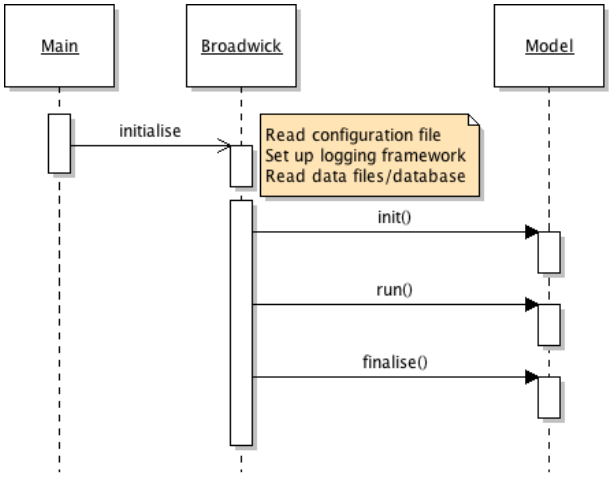


Figure 2.6: Schematic outline of the steps Broadwick performs on startup

2.2 Configuration Files

The configuration file MUST conform to the Broadwick.xsd specification that is supplied with the Broadwick source code. The configuration is contained within the <project> </project> tags and contains the following tags

<logs>	<console>	Contains <level> and <pattern> tags to define the level and structure of logging messages displayed on the console.
	<files>	Contains <level> and <pattern> tags like the <console> log but also a <name> for the name of the file to contain the log messages and a boolean <overwrite> tag that lets Broadwick know if any existing file with that name should be overwritten.
<data>	<databases>	Contains the <name> tag to specify the location of the database.
	<datafiles>	Contains the <DirectedMovementFile> <FullMovementFile> <BatchMovementFile> <PopulationFile> <LocationsFile> and <TestsFile> tags for the various file structures recognised by Broadwick. See the following tables in this section for a description of each.
<models>	<model>	Broadwick can run several models concurrently (though they will share the same logging and data configurations).

Each model element contains the following:

<classname>	Exactly one classname element that is the fully qualified name of the java class (that implements the broadwick.model.Model interface). This class MUST have a no argument constructor that Broadwick will use to create an instance through relection.
<priors>	The <priors> tag can contain as many elements as necessary, each prior contains an id attribute and optional <hint> and <initialVal> elements. The following priors are recognised by Broadwick <uniformPrior> additionally contains <min> and <max> elements for uniformly distributed priors. <gaussianprior> additionally contains <mean> and <deviation> tags for normally distributed priors. <parameter> The parameters for the model can be encoded in this tag using the id and value attributes to name the parameters and give the parameter value. These can be accessed by the broadwick.model.Model getParameterAs[TYPE](id) method that retrieves the parameter with the given id and converts it the the required type.

Each datafile contains specific information on it's layout, i.e. the columns in the file where the required data can be found. The structure of each recognised data file is outlined below.

DirectedMovementFile:

<name>	The name (including path from the configuration file) where the file can be found.
<alias>	An alias for the file.
<separator>	The character separating the columns in the datafile, e.g. ‘,’ ‘<tab>’.
<idColumn>	
<movementDateColumn>	
<movementDirectionColumn>	
<locationColumn>	
<speciesColumn>	
<dateFormat>	
<customTags>	This optional field allows for optional information to be stored in the database.

FullMovementFile:

<name>	The name (including path from the configuration file) where the file can be found.
<alias>	An alias for the file.
<separator>	The character separating the columns in the datafile, e.g. ‘,’ ‘<tab>’.
<idColumn>	
<departureDateColumn>	
<departureLocationIdColumn>	
<destinationDateColumn>	
<destinationLocationIdColumn>	
<marketIdColumn>	
<marketDateColumn>	
<speciesColumn>	
<dateFormat>	
<customTags>	This optional field allows for optional information to be stored in the database.

BatchMovementFile:

<name>	The name (including path from the configuration file) where the file can be found.
<alias>	An alias for the file.
<separator>	The character separating the columns in the datafile, e.g. ‘,’ ‘<tab>’.
<batchSizeColumn>	
<departureDateColumn>	
<departureLocationIdColumn>	
<destinationDateColumn>	
<destinationLocationIdColumn>	
<marketIdColumn>	
<marketDateColumn>	
<speciesColumn>	
<dateFormat>	
<customTags>	This optional field allows for optional information to be stored in the database.

PopulationFile:

<name>	The name (including path from the configuration file) where the file can be found.
<alias>	An alias for the file.
<separator>	The character separating the columns in the datafile, e.g. ‘,’ ‘<tab>’.
<lifehistory>	
<population>	
<speciesColumn>	
<dateFormat>	
<customTags>	This optional field allows for optional information to be stored in the database.

Lifehistory:

<name>	The name (including path from the configuration file) where the file can be found.
<alias>	An alias for the file.
<separator>	The character separating the columns in the datafile, e.g. ‘,’ ‘<tab>’.

Population:

<name>	The name (including path from the configuration file) where the file can be found.
<alias>	An alias for the file.
<separator>	The character separating the columns in the datafile, e.g. ‘,’ ‘<tab>’.

LocationFile:

<name>	The name (including path from the configuration file) where the file can be found.
<alias>	An alias for the file.
<separator>	The character separating the columns in the datafile, e.g. ‘,’ ‘<tab>’.
<locationIdColumn>	The column in the file containing the id of the location.
<eastingColumn>	The column in the file containing the easting coordinate. Coordinates aren’t strictly adhered to in Broadwick so a simple y-coordinate is sufficient.
<northingColumn>	The column in the file containing the northing coordinate. Coordinates aren’t strictly adhered to in Broadwick so a simple x-coordinate is sufficient.
<dateFormat>	
<customTags>	This optional field allows for optional information to be stored in the database.

TestsFile:

<name>	The name (including path from the configuration file) where the file can be found.
<alias>	An alias for the file.
<separator>	The character separating the columns in the datafile, e.g. ',' '<tab>'.
<idColumn>	One of these is required, specifying whether the test is performed on an individual, group (e.g. herd) or location (must match the id in the Location file).
<groupIdColumn>	
<locationIdColumn>	
<testDateColumn>	
<postiveResultColumn>	
<negativeResultColumn>	
<dateFormat>	
<customTags>	This optional field allows for optional information to be stored in the database.

A simplified configuration file is generated in the skeleton project. It contains configuration items for logging to console and to file for different logging levels (info, warning, error, debug, trace) and we can specify the pattern to apply to the log message.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<project>
    <logs>
        <console>
            <level>info</level>
            <pattern>[%thread] %-5level %msg %n</pattern>
        </console>
        <file>
            <name>broadwick.stochasticsir.log</name>
            <level>info</level>
            <pattern>[%thread] %-5level %msg %n</pattern>
            <overwrite>true</overwrite>
       </file>
    </logs>

    <models>
          <model id="Broadwick Project">
                 <classname>broadwick.stochsir.StochasticSIR</classname>
          </model>
    </models>
</project>
```

Common logging patterns are:

%C	Outputs the fully-qualified class name of the caller issuing the logging request.
%M %method	Outputs the method name where the logging request was issued.
%L %line	Outputs the line number from where the logging request was issued.
%F %file	Outputs the file name of the Java source file where the logging request was issued. This is not very fast and should be avoided.
%d	Used to output the date of the logging event e.g. %dHH:mm:ss,SSS
%m (%msg)	Outputs the application-supplied message associated with the logging event.
%t (%thread)	Outputs the name of the thread that generated the logging event.
%n	Outputs the platform dependent line separator character or characters
%r	Outputs the number of milliseconds elapsed since the start of the application until the creation of the logging event.
%p %level	Outputs the level of the logging event.

More details on logging patterns can be found at `http://logback.qos.ch/manual/layouts.html`.

The model section requires a <classname> giving the fully qualified class name and optional <priors> and <parameter> sections.

2.3 Extending the Model

Our stochastic SIR model that we have created is a valid Broadwick model but does not perform any useful calculations. We will add some parameters to the configuration file and read (and log them) in the init() method.

Firstly, let us define beta and rho parameters for the susceptible->infectious rate and for the infectious->recovered rates respectively and parameters for the maximum time for which we will run the simulation and the name of a file in which we will save the time series data. To do this modify the configured model section by:

```
<model id="Broadwick Project">
    <classname>broadwick.stochasticsir.StochasticSIR</classname>

    <parameter id="beta" value="0.2" />
    <parameter id="rho" value="0.3" />
    <parameter id="tMax" value="100" />
    <parameter id="outputFile" value="broadwick.stochasticSIR.dat" />
</model>
```

Now edit the init() method of the StochasticSir class as shown in fig 2.7.

The ‘Model’ class contains getParameterValue(String), getParameterValueAsDouble(String), getParameterValueAsInteger(String), getParameterValueAsBoolean(String) methods to extract parameters from the configuration file as strings (default), doubles, integers and booleans (if the parameter is written as “true" or “false").

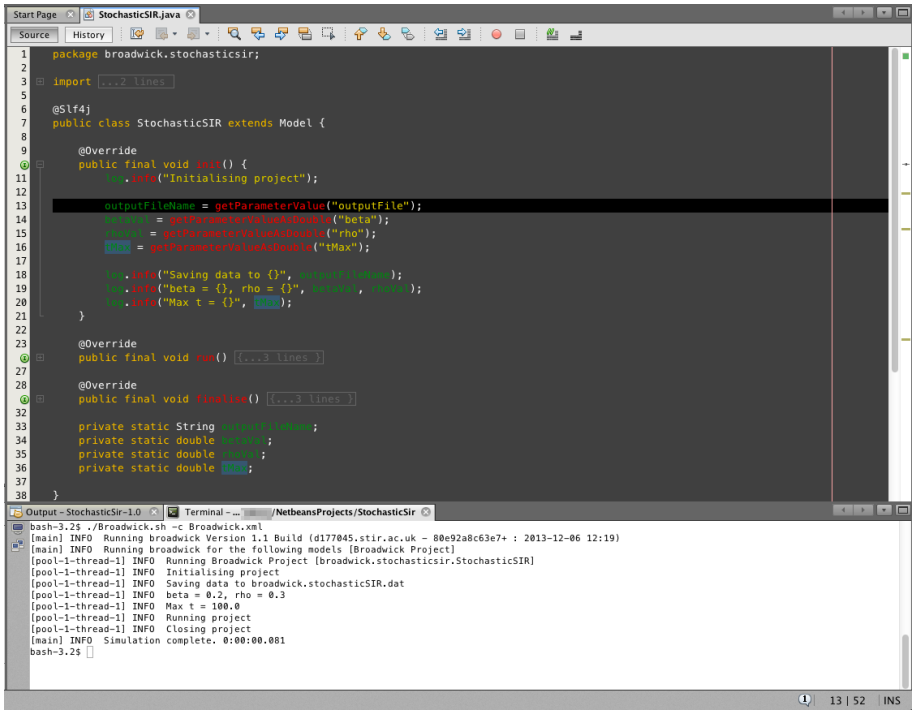


Figure 2.7: Reading some parameters to our model

3. Calculation Packages

Broadwick supports several simulation and fitting models. In this chapter we will give an outline of these methods and how they can be simulated (and combing) using the Broadwick framework. We will, for the most part, dispense with the theory and concentrate on how the methods are implemented in Broadwick. The code snippets in this chapter are taken from the examples that are distributed with the Broadwick source code.

3.1 Markov Chains

A Markov chain is a random sequence of states where the current state depends solely on the previous state. In this sense, it is a "memoryless" process as the transition from one state to the next does not depend on the sequence of states that preceeded it.

A Markov Chain can be implemented in Broadwick using the MonteCarloStep and MarkovChain classes. The MonteCarloStep encapsulates the functionality of a state by maintaining a collection of the coordinates defining the state as a java.util.Map<String,Double> (i.e. the name and value of the state). A MarkovChain object is constructed using a MonteCarloStep object as an initial point and, optionally, a MarkovProposalFunction for generating the next step. The generateNextStep method uses the proposal function to generate the next step in the chain as the following code snippet demonstrates,

```
final Map<String, Double> coordinates = new LinkedHashMap<>();
        {
            coordinates.put("x", 0.0);
            coordinates.put("y", 0.0);
        }
final MonteCarloStep initialStep = new MonteCarloStep(coordinates);

final MarkovChain mc = new MarkovChain(initialStep);
for (int i = 0; i < chainLength; i++) {
    final MonteCarloStep nextStep = mc.generateNextStep(mc.getCurrentStep());
    mc.setCurrentStep(nextStep);

    log.trace("{}", nextStep.toString());
}
```

By default, a MarkovNormalProposal object is used to generate the next step by sampling from a Normal distribution centered on the current coordinate and with a standard deviation of 1. New proposal classes implement the MarkovProposalFunction interface and using this object when creating the Markov Chain.

```
final MarkovProposalFunction myProposer = new MarkovProposalFunction();
final MarkovChain mc = new MarkovChain(initialStep, myProposer);
```

3.2 Monte Carlo Simulation

Monte Carlo simulation is a broad class of methods that reply on repeated simulation of [random] processes to derive numerical results. Broadwick uses the MonteCarloScenario abstract class to encapsulate the process to be simulated, which is used by the MonteCarlo class to implement the Monte Carlo process.

Internally, the MonteCarlo class uses a producer-comsumer pattern by creating a ThreadFactory to spawn several simulation processes which are in turn consumed by a MonteCarloResults object. This MonteCarloResults object takes the results of each simulation (the producer threads) and calculates the required statistics.

These classes (MonteCarloScenario, MonteCarloResults) are extended for each implementation. By way of example, we will calculate π by throwing darts randomly at a square dartboard and calaulating the fraction that fall within the unit circle encompasses by the square.

First, we implement the run method of our class that extends the MonteCarloScenario class (this class has an internal random number generator, rng, to generate random number).

```
@Override
public MonteCarloResults run() {
    final MyResultsConsumer results = new MyResultsConsumer();

    final double x = rng.getDouble(-1, 1);
    final double y = rng.getDouble(-1, 1);

    final double r = Math.sqrt(x * x + y * y);
    if (r < 1) {
        results.addHit();
    } else {
        results.addMiss();
    }
    return results;
}
```

The MonteCarloResults class is responsible for both storing the results of each simulation and for acting as a consumer object that maintains a collection of the results returned by the producers.

```
class MyResultsConsumer implements MonteCarloResults {

@Override
public double getExpectedValue() {
    return hits.getSum() / (hits.getSum() + misses.getSum());
}
```

```
    @Override
    public Samples getSamples() {
        return hits;
    }

    @Override
    public String toCsv() {
        return String.format("\%d ; \%d", hits.getSize(), misses.getSize());
    }

    @Override
    public MonteCarloResults join(final MonteCarloResults results) {
        // This is where the results of the producers are dealt with.
        final MyResultsConsumer r = (MyResultsConsumer) results;
        this.hits.add(r.hits);
        this.misses.add(r.misses);

        return this;
    }

    public void addHit() {
        hits.add(1);
    }

    public void addMiss() {
        misses.add(1);
    }

    @Override
    public void reset() {
    }

    @Getter
    private final Samples hits = new Samples();
    @Getter
    private final Samples misses = new Samples();
}
```

These two classes are utilised thus

```
    MonteCarlo mc = new MonteCarlo(new Simulation(), 1000);
    mc.setResultsConsumer(new MyResultsConsumer());
    mc.run();

    final MyResultsConsumer results = (MyResultsConsumer) mc.getResults();
    log.info("Hits : Misses = {}", results.toCsv());
    log.info("Estimation of Pi = {}", 4 * results.getExpectedValue());
```

3.2.1 Markov Chain Monte Carlo

Markov chains can be combined with Monte Carlo simulation to explore a parameter space by using the Markov chain to 'walk' through the parameter space while Monte Carlo simulation is to determine the state of the system and each step in the walk. Thus, Markov chain Monte Carlo (MCMC) methods can be used to sample from a probability distribution by using the Markov chain (coupled with a rejection function to accept or reject propsed steps) to find the desired

distribution.

A MarkovChainMonteCarlo object will create a Markov chain and run the Monte Carlo simulation at each step using the Metropolis-Hastings algorithm (the MetropolisHastings class) to accept successive steps based on the results of the Monte Carlo simultaion at the given step.

A MarkovChainObserver object can be added to the MarkovChainMonteCarlo object which will be informed when a step has been completed. This observer can be used to save the results of the simulation.

By default a Metropolis-Hastings, but it is easy to implement an alternative (in the following example we will use the Metropolis algorithm with a log-likelihood).

```
final MonteCarloStep step = new MonteCarloStep(<initial step>);
final MarkovChainMonteCarlo mcmc = new MarkovChainMonteCarlo(
                myModel, <numScenarios>,
                myMonteCarloScenarioResults,
                myMarkovStepGenerator);

mcmc.setAcceptor(new MonteCarloAcceptor() {
            @Override
            public boolean accept(final MonteCarloResults oldResult, final
                MonteCarloResults newResult) {
                final double ratio = newResult.getExpectedValue() -
                    oldResult.getExpectedValue();
                return Math.log(generator.getDouble()) < ratio /
                    smoothingRatio;
            }
        });

MyMarkovChainObserver myMcmcObserver = new MyMarkovChainObserver(mcmc);
mcmc.addObserver(myMcmcObserver);

mcmc.run();
```

We can attach a MarkovChainObserver to the MarkovChainMonteCarlo object that observes the state of the chain (at each step the chain object informs the observer of its state which can be used, e.g., to save it to file).

```
final MarkovChainObserver observer = new MyMCObserver();
mcmc.addObserver(observer);
```

3.3 Approximate Bayesian Computation (ABC)

Approximate Bayesian computation (ABC) is a class of computational methods based on Bayesian Statistics [1–4]. It bypasses the evaluation of a likelihood function , which is often computationally expensive or even impossible.

At the heart of the ABC method is the ‘distance function’ which is a measure of close the proposed sample is to the desired (posterior). The distance function in Broadwick is specified by overriding the AbcDistance class and adding it to an ApproxBayesianComp object, by default a simple absolute value of the difference between the proposed and observed value is used (as can be seen in the following example).

An AbcController object is used to control (i.e. determine when the calculation should end) the ABC process.

The ApproxBayesianComp class runs the bayesian computation. It is constructed using

observed data (in the form of a AbcNamedQuantity object, which is a simple name-value map), an AbcModel object, an AbcPriorsSampler object (which specifes how samples are drawn from a prior distribution) and a sensitivity.

As a very simple example, we will sample from a posterier that is normally distributed around π (assuming a standard deviation of 1.0). Using a uniform prior (for the mean of the posterior distribution) in [3,4] we will sample from this using a simple abs() function as the distance function. We should observe a posterior distribution [normally] distributed around π.

```
// first set up the observed data (the mean value of my unknown
    distribution).
final Map<String, Double> observed = new LinkedHashMap<>();
observed.put("value", 3.142);
final AbcNamedQuantity observedData = new AbcNamedQuantity(observed);

// Next create a simple controller (we will sample 20000) points from
    the prior distribution.
final AbcController controller = new AbcController() {
    @Override
    public boolean goOn(final ApproxBayesianComp abc) {
        return abc.getNumSamplesTaken() <= 20000;
    }
};

// Create a dummy model to run
final AbcModel myModel = new AbcModel() {
    @Override
    public AbcNamedQuantity run(final AbcNamedQuantity parameters) {
        // this is trivially simple model. Instead of doing any
            calculations we just return the parameters.
        return parameters;
    }
};

// Create a method for sampling our priors.
final AbcPriorsSampler priors = new AbcPriorsSampler() {
    @Override
    public AbcNamedQuantity sample() {
        // Another dummy method here, we uniformly sample 'value' in
            the range [3,4]
        final LinkedHashMap<String, Double> sample = new
            LinkedHashMap<>();
        sample.put("value", generator.getDouble(3.0, 4.0));
        return new AbcNamedQuantity(sample);
    }
    private final RNG generator = new RNG(RNG.Generator.Well19937c);
};

final ApproxBayesianComp abc = new ApproxBayesianComp(observedData,
    myModel, priors, 0.05);
abc.setController(controller);
abc.run();
```

3.4 Ordinary Differential Equations (ODEs)

Ordinary differential equations can be solved in Broadwick using the 4^{th} order Runge-Kutta method. The RungeKutta4 object is constructed using an Ode objectlindexClass!Ode (which specifies the ODEs by implementing methods to specify the initial alues and derivatives of each variable), start and end times and the step size.

An observer can be attached to keep track of the results generated by the solver and a controller can be used to ensure that the calculation stops (if, for example, you specify a negative step size resulting in infinite loops or obtain a negative value for a population size).

```
final Ode myOde = new SirModel(beta, rho, s0, i0, r0);
final OdeSolver solver = new RungeKutta4(myOde, tStart, tEnd, stepSize);

// we will need an observer to 'observe' our simulation and record the
    simulation states.
solver.getObservers().clear();
final MyObserver observer = new MyObserver(solver, outputFile);
solver.addObserver(observer);

// Create a simple controller object to tell the simulator when to stop.
final OdeController controller = new MyOdeController(tEnd);
solver.setController(controller);

solver.run();
```

We can also add triggered events to the solver. These events occur at predetermined times and can be used to model, e.g. immigration events, vaccination or culling strategies in a population.

```
// Register theta events, these are fixed events that will be triggered at set
    times.
solver.registerNewTheta(observer, 20.0, new MyThetaEvent(solver));
```

Appendix

Maven as a Build Tool

There is no requirement to use maven as a build tool but as Broadwick and it's examples are built using maven this section will give a brief outline of how maven is used to create a simple model.

Maven uses an xml file to describe the classes to be built as well as the dependencies, dynamically downloading required libraries as needed. It uses the 'convention over configuration' paradigm imposing the directory structure given in the table below.

Directory	**Purpose**
Project home	Contains the pom and all the subdirectories.
src/main/java	Contains the java source code for the project.
src/main/resources	Contains the xsd file for configuring the project.
src/test/java	Contains any [Junit or TestNG] test cases for the project.
src/test/resources	Contains resources necessary for testing.

Maven's directory structure

Maven's equivalent to a makefile or Ant's build.xml is a 'project object model' which is stored in a pom.xml file. A good reference for the maven pom is http://maven.apache.org/pom.html.

Bibliography

[1] T Toni et al. “Approximate Bayesian computation scheme for parameter inference and model selection in dynamical systems”. In: *J R Soc Interface* 6 (2007), pages 187–202 (cited on page 20).

[2] J-M Marin et al. “Relevant statistics for Bayesian model choice”. In: *ArXiv:11104700v1 [mathST]* (2011) (cited on page 20).

[3] P Marjoram et al. “Markov chain Monte Carlo without likelihoods”. In: *Proc Natl Acad Sci USA* 100 (2003), pages 15324–15328 (cited on page 20).

[4] Excoffier L (2009) Wegmann D Leuenberger C. “Efficient approximate Bayesian computation coupled with Markov chain Monte Carlo without likelihood.” In: *Genetics* 182 (2009), pages 1207–1218 (cited on page 20).

Index

N

O

P

R

T

U