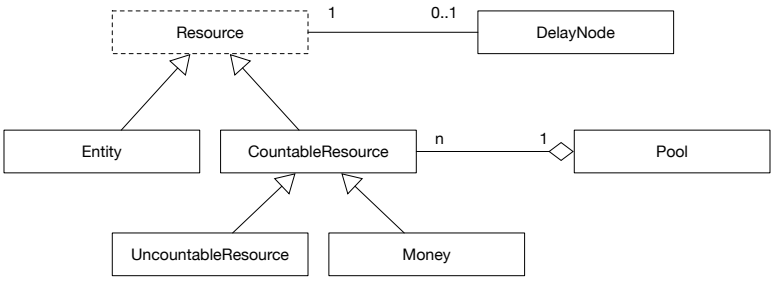
MASON Discrete Event Simulation Extension

Sean Luke
Department of Computer Science
George Mason University

Manual Version 2
June 2023

Contents



0 Introduction

0.1 Installing the DES Extension

0.2 What You Need to Know about MASON

0.3 About Discrete Event Simulation

0.4 About Hybrid ABM and DES Simulation

1 Resources and Related Objects

The DES Extension has two basic kinds of classes: **Resources** and **Processes**. Processes perform actions in response to events. In most cases, this action involves Processes handing off Resources to other Processes. This could be for many reasons. Perhaps a Resource moving from Process A to Process B to another represents A paying B for some service. Perhaps Processes A and B are warehouses and the Resource represents a truck moving from one to the other. Perhaps A and B are countries and the Resource represents a family migrating. Maybe A and B are spies and the Resource holds a critical communication between them.

Resources are also held in **Pools**. A Pool can be dipped into by certain Processes to allocate a Resource for some temporary function. For example, imagine if a factory floor had four assembly lines but only two lathes. Widgets moving through an assembly line at some point had to be lathed, and if a lathe was not available, the widget (and its assembly line) would have to wait until one came available.

To do this, there are two special kinds of Processes, copies of which are part of each assembly line, called **Lock** and **Unlock**. For a widget to pass through a Lock, the Lock must first allocate a Resource (representing the lathe) from a shared Pool. If it cannot, the widget must wait until the Lock can allocate the Resource. Then the widget may pass through to other Processes. Ultimately when widget is finished with the lathe, the widget can pass through an **Unlock** Process, which gives a Resource back to the Pool. The Pool only holds two lathe Resources.[1]

Resources are often passed through **Delays**, Processes which hold them up for a certain amount of time. To manage this time, Delays will store Resources in **DelayNodes** and associate them with timestamps.

MASON provides several kinds of Resources, depending on your modeling needs:

- **Entities** are atomic elements, like tickets or tokens or cars. They cannot be broken up into smaller Entities, nor joined together to form larger ones. They are not worth different amounts: and Entity is an Entity.
- **Composite Entities** are Entities which hold a collection of other Resources inside them, plus a manifest. For example, a Composite Entity might be used to model a shipping container filled with teddy bears. The manifest can indicate anything you'd like: expiration dates, country of origin, who manufactured

[1] These are called Lock and Unlock because they are reminiscent of locking or unlocking on a mutex or a semaphore in computer science terminology. So sue me, I'm a computer science professor. Others in the Discrete Event Simulation world might instead refer to **seizing** and **releasing** resources from a pool.

each teddy bear, the amount of love that went into each teddy bear, and so on. There is a special Process called a **Composer** which takes a collection of Resources and builds a Composite Entity

- **Countable Resources** are Resources which can be divided, but only as integers. For example, packages of medicine pills are countable resources: the package could be 0 pills, or 1 pill, or 2 pills, or 1500 pills. But it cannot be 1.5 pills. You cannot have a negative or infinite value.
- **Money** is an obvious example of a Countable Resource. In US currency, the fundamental unit of money is the penny. You cannot have a half a penny, and you cannot have 3.279 pennies.[2] Because money is so common, MASON has a dedicated kind of Countable Resource just to represent it: it prints it out in a pleasing fashion (like "$14.23" for 1423 pennies).
- **Uncountable Resources** are Resources which can be divided indefinitely. An example of an uncountable resource is water. You can have 3.239122 liters of water, and you can divide that into five parts any way you like, including some parts holding 0 liters. You cannot have a negative or infinite value.

Resources are typed, and you can have as many different types of Resources as you like. For example, you could have a type of Countable Resource representing pills and another one representing punches in the face. You could have a type of Uncountable Resource representing water and another one representing love. You could have both US dollars and Hong Kong dollars. You could have both cargo containers and suitcases of stuff. These types are not exchangeable: you can't give a Process water when it's expecting love. However there exists a special Process, called a **Transformer** which can convert certain Resources into Countable Resources: for example, it could be used as a currency converter.

If Resources are typed, how do you make a new type of Resource, and how do you make more of that same type? Each Resource type has a unique name. To make a new type of resource, just instantiate a Resource class with that new name. Later Resources of that type are allocated by copying them from earlier Resources of the same type. This can be done either by calling duplicate() or by calling a copy constructor on the earlier Resource, indicating the amount of the new Resource.

Processes deal with Resources differently based on the kind of Resources. For example, many Processes store Resources to eventually hand off to downstream Processes. If the Resource is an Entity, then it is atomic and the Process must store it in a collection of Entities. If the Resource is a Countable or Uncountable Resource, then the Process can simply merge it into one pile or blob of Countable (or Uncountable) Resource, and dole it out as appropriate later on.

All Resources have **amounts**. For Countable or Uncountable resources, the amount is simply the size of the resource: such as 1500 pills or 1423 pennies or 3.239122 liters of water. For Entities, including Composite Entities, the amount is always 1.

1.1 Resource

Resource is the abstract superclass of all resources. All Resources have a **unique type** (an integer) shared by resources of that type, and a **name** (a String) which you can stipulate. For example, all Electricity resources might have type 0, and all Gas resources might have type 1 and all Cars might have type 2 and all Dollars might have type 3. It is possible to construct two Resources with the same name, but they will have different types even so: don't do that. For your own sanity, make sure that you only construct a Resource once with a given name, and copy Resources from it to make more of the same type.

sim.des.Resource Constructor Methods

public Resource(String name)
: Builds a new Resource, of a new unique type, with the given name.

protected Resource()
: Builds a new Resource, but does not set the name or type. This exists to permit copy constructors in subclasses. Normally you'd leave it alone.

[2] Except on the stock exchange.

sim.des.Resource Methods

public void clear()
Clears the resource. What this does varies depending on the subclass.

public void toString()
Prints the resource in a pleasing manner.

public boolean equals(Object other)
Returns true if *other* is a Resource, is not null, and if it is both the same type and amount as this Resource. The storage and info elements inside Composite Entities are not checked for equality.

public int hashCode()
Returns an appropriate hash code for hash tables.

public double getAmount()
Returns the amount of the Resource. Note that this is always double, even for Countable Resources and for Entities, both of which return integers.

public boolean isSameType(Resource other)
Returns true if *other* is not null and the same type as this Resource.

public int getType()
Returns the type of this resource.

public String getName()
Returns the name of this resource.

protected void setName(String name)
Sets the name of this resource. You shouldn't call this probably.

public Resource duplicate()
Exactly duplicates the Resource once and returns the result. The storage and info objects are not copied — just pointer-copies. You'll have to deep copy them as you see fit.

public Resource[] duplicate(int times)
Exactly duplicates the Resource *times* times and returns the result. The storage and info objects are not copied — just pointer-copies. You'll have to deep copy them as you see fit.

1.2 Entity

An Entity is a Resource that cannot be subdivided into smaller Resources of the same time. For example: a Car might be an Entity. But Water is not, as you can divide Water up into smaller amounts. Entities can **store** other resources inside them: that is, they can be **composed** of them. For example, a given Car might contain Wheels, and Engine, and some amount of Gasoline. Different Cars are permitted to store different things. Entities can also contain **info** objects, essentially manifests.

sim.des.Entity Constructor Methods

public Entity(String name)
Builds a new Entity, of a new unique type, with the given name.

public Entity(Entity other)
Makes a copy of the Entity, including its type and name. The storage and info objects are pointer-copied, not deep-copied. This is the standard copy constructor for Entity.

Here are the custom methods for Entity. It also implements the methods defined in Resource.

sim.des.Entity Methods

public Resource[] getStorage()
Returns the actual storage array of the Entity, or null if there isn't one.

public void setStorage(Resource[] val)
Sets the storage array of the Entity. This can be set to null.

public Object getInfo()
Returns the actual info object of the Entity, or null if there isn't one.

public void setInfo(Object val)
Sets the info object of the Entity. This can be set to null.

public boolean isComposite()
Returns true if the storage is non-null.

public void clear()
Sets the storage to null.

public double getAmount()
Always returns 1.0.

public boolean equals(Object other)
Returns true if *other* is an Entity, is not null, and if it is the same type as this Entity. The storage and info elements inside Composite Entities are not checked for equality.

public int hashCode()
Returns an appropriate hash code for hash tables, not considering the storage or info elements.

1.3 CountableResource

A CountableResource is a Resource that has an **amount** (an integer) and can be subdivided into CountableResources with smaller integer amounts. For example: a Population might be divided into smaller subpopulations. There is an atomic, smallest, non-divisible amount of CountableResources: 1. You can also set the amount to 0.

Even though CountableResource only stores integers for amounts, it stores them as doubles. This means that its maximum integer value is larger than that of an int. Specifically, it is equal to sim.des.CountableResource.MAXIMUM_INTEGER

sim.des.CountableResource Constructor Methods

public CountableResource(String name, double intialAmount)
Builds a new CountableResource, of a new unique type, with the given name and amount.

public CountableResource(String name)
Builds a new CountableResource, of a new unique type, with the given name and zero amount.

public CountableResource(CountableResource other)
Makes a copy of the CountableResource, including its type, name, and amount. If the provided object is actually an UncountableResource, this will throw an exception.

public CountableResource(CountableResource other, double initialAmount)
Makes a copy of the CountableResource, including its type and name, but with the new amount provided. If the provided object is actually an UncountableResource, this will throw an exception.

Here are the custom methods for CountableResource. It also implements the methods defined in Resource.

sim.des.CountableResource Methods

public double doubleValue()
Returns the amount as a double value.

public boolean isUncountable()
Returns false (note that this is overridden by UncountableResource).

public boolean isCountable()
Returns true (note that this is overridden by UncountableResource).

public double getAmount()
Returns the amount stored in the CountableResource, which will always be a nonnegative integer (but possibly larger than an int).

public void clear()
Sets the amount to 0.

public double getAmount()
Returns the amount.

public void setAmount(double val)
Sets the amount

public void bound(double min, double max)
Bounds the resource to be no more than max and no less than min. It must be the case that max $\geq$ min ≥ 0.

public void bound(double max)
Bounds the resource to be no more than max and no less than zero. It must be the case that max ≥ 0.

public boolean increase(double val)
Increases the amount by the given value and returns true, unless val is not an integer, or unless this would exceed sim.des.CountableResource.MAXIMUM_INTEGER, in which case nothing happens and false is returned.

public boolean decrease(double val)
Decrements the amount by the given value and returns true, unless val is not an integer, or unless this would drop below zero, in which case nothing happens and false is returned.

public boolean increment()
Increments the amount by 1 and returns true, unless this would exceed sim.des.CountableResource.MAXIMUM_INTEGER, in which case nothing happens and false is returned.

public boolean decrement()
Decrements the amount by 1 and returns true, unless this would drop below zero, in which case nothing happens and false is returned.

public CountableResource reduce(double atLeast, double atMost)
Subtracts at least a certain amount and at most a certain amount from this CountableResource, placing that amount into a new CountableResource and returning it, unless there is not enough amount to do so, in which case nothing happens and false is returned. atLeast and atMost must be integers, with atMost $\geq$ atLeast ≥ 0.

public CountableResource reduce(double byExactly)
Subtracts exactly a certain amount from this CountableResource, placing that amount into a new CountableResource and returning it, unless there is not enough amount to do so, in which case nothing happens and false is returned. byExactly must be an integer ≥ 0.

public void add(CountableResource other)
Adds the other CountableResource's amount into this one, setting the other CountableResource's amount to zero afterwards.

public void add(CountableResource other, double atMostThisMuch)
Adds atMostThisMuch of the other CountableResource's amount into this one, setting the other CountableResource's amount to the remainder afterwards.

public void add(CountableResource[] other)
Adds all the other CountableResources' amounts into this one, setting the other CountableResources' amounts to zero afterwards.

public boolean greaterThan(CountableResource other)
Returns true if this amount is greater than the other resource's amount.

public boolean greaterThanOrEquals(CountableResource other)
Returns true if this amount is greater than or equal to the other resource's amount.

public boolean lessThan(CountableResource other)
Returns true if this amount is less than the other resource's amount.

public boolean lessThanOrEquals(CountableResource other)
Returns true if this amount is less than or equal to the other resource's amount.

public int compareTo(Object other)
Returns 0 if the other object's amount is equal to, -1 if it is greater than, and 1 if it is less than my amount.

1.4 Money

Money is a CountableResource with a cute printing function which prints it with a currency sign. For example, a Dollar is Money, hence a CountableResource, where the smallest amount (1) would be the Cent. Perhaps it should be renamed Cents.

sim.des.Money Constructor Methods

public Money(String name, double intialAmount)
Builds a new Money, of a new unique type, with the given name and amount. The name will be used as a currency symbol during printing.

public Money(String name)
Builds a new Money, of a new unique type, with the given name and zero amount. The name will be used as a currency symbol during printing.

public Money(Money other)
Makes a copy of the Money, including its type, name, and amount.

public Money(Money other, double intialAmount)
Makes a copy of the Money, including its type and name, but with the new amount provided.

Here are the custom methods for Money. It also implements the methods defined in CountableResource.

sim.des.Money Methods

public String toString()
Returns the amount in decimal format as $CX.Y$, where C is the name (a currency symbol), X is the amount / 100, and Y is the amount mod 100. For example, if "$" was the name, and the amount was 1423, then this would print as "$14.23"

1.5 UncountableResource

An UncountableResource is a CountableResource which can be subdivided infinitely and has real-valued amounts. Thus while you can only have a CountableResource with values 0, 1, 2, 3, ..., an UncountableResource could be any positive amount, such as 0 or 2.34129 or 92.3 or Infinity.

If you have a variable holding a CountableResource, how do you know it is a CountableResource versus an UncountableResource? You could use `instanceof`, or you could call `isCountable()` or `isUncountable()`.

sim.des.UncountableResource Constructor Methods

public UncountableResource(String name, double intialAmount)
Builds a new UncountableResource, of a new unique type, with the given name and amount.

public UncountableResource(String name)
Builds a new UncountableResource, of a new unique type, with the given name and zero amount.

public UncountableResource(UncountableResource other)
Makes a copy of the UncountableResource, including its type, name, and amount.

public UncountableResource(UncountableResource other, double intialAmount)
Makes a copy of the UncountableResource, including its type and name, but with the new amount provided.

Here are the custom methods for UncountableResource. It also implements the myriad of methods defined in CountableResource.

sim.des.UncountableResource Methods

public UncountableResource[] divide(int times)
Divides the amount by *times*, which must be > 0, and builds *times* -1 new UncountableResources, each with the new amount. Also reduces the amount of this UncountableResource to the amount as well.

public UncountableResource halve()
Builds a new UncountableResources with half the amount, leaving the other half as the current amount.

public void scale(double value)
Changes the amount by multiplying it by the given value.

public boolean increase(double val)
Increases the amount by the given value and returns true.

public boolean decrease(double val)
Decrements the amount by the given value and returns true, unless this would drop below zero, in which case nothing happens and false is returned.

1.6 Pool

A **Pool** is a storage of a single kind of CountableResource, UncountableResource, or Money. Various Process objects, namely **Lock** and **Unlock**, dip into a shared Pool to add or remove resources or to wait until resources have come available.

Pool is very simple: it has a **maximum** amount of resource, a **minimum** (always zero), and an **initial resource allocation**. Beyond that, its current resource can be set and queried. And that's it!

A Pool extends sim.des.portrayal.DESPortrayal, so it can serve as a MASON SimplePortrayal. DESPortrayal objects are also sim.des.Named, so the Pool can be given a name via getName() and setName(...).

A Pool is sim.des.Resettable, meaning that it can be reset to its original state via a method called reset(...).

sim.des.Pool Constructor Methods

public Pool(CountableResource resource, double maximum)
Builds a Pool of the given type of CountableResource, with an initial amount copied from the CountableResource, and the provided maximum.

public Pool(CountableResource resource)
Builds a Pool of the given type of CountableResource, with an initial amount copied from the CountableResource, and a maximum of infinity (if it's an UncountableResource), or CountableResource.MAXIMUM_INTEGER (if it's a CountableResource or Money).

public Pool(double initialResourceAllocation)
Builds a Pool of a new type of CountableResource, with an initial amount as specified, and a maximum of infinity (if it's an UncountableResource), or CountableResource.MAXIMUM_INTEGER (if it's a CountableResource or Money).

public Pool(double initialResourceAllocation, double maximum)
Builds a Pool of a new type of CountableResource, with an initial amount as specified, and the specified maximum.

A Pool only has a few methods:

sim.des.Pool Methods

public void reset(SimState state)
Resets the Pool. This simply resets its current amount to the initial amount: the name and maximum stay as you had last set them.

public CountableResource getResource()
Returns the current available resource.

public void setResource(CountableResource val)
Sets the current available resource.

public double getMaximum()
Returns the maximum allowed resource.

public void setMaximum(double value)
Sets the maximum allowed resource.

public String getName()
Returns the name (which can be null).

public void getName(String name)
Sets the name (which can be null).

public String toString()
Returns the Pool and its state in a pleasing fashion.

1.7 DelayNode

A **DelayNode** holds a Resource and associates it with a timestamp and a **Provider** (a Process which provided the Resource, discussed later). DelayNodes can be strung together into small Linked Lists. DelayNodes are used by various delays (SimpleDelay, Delay, and BoundedDelay) to describe how long a Resource must be delayed before the delay Process makes it available to others. Additionally, a DelayNode can be marked **dead**, meaning that its Resource has been destroyed and will no longer be available to others after it leaves the delay.

sim.des.DelayNode Constructor Methods

public DelayNode(Resource resource, double timestamp, Provider provider)
Builds a DelayNode with the given resource, timestamp, and provider.

---

sim.des.Delay Methods

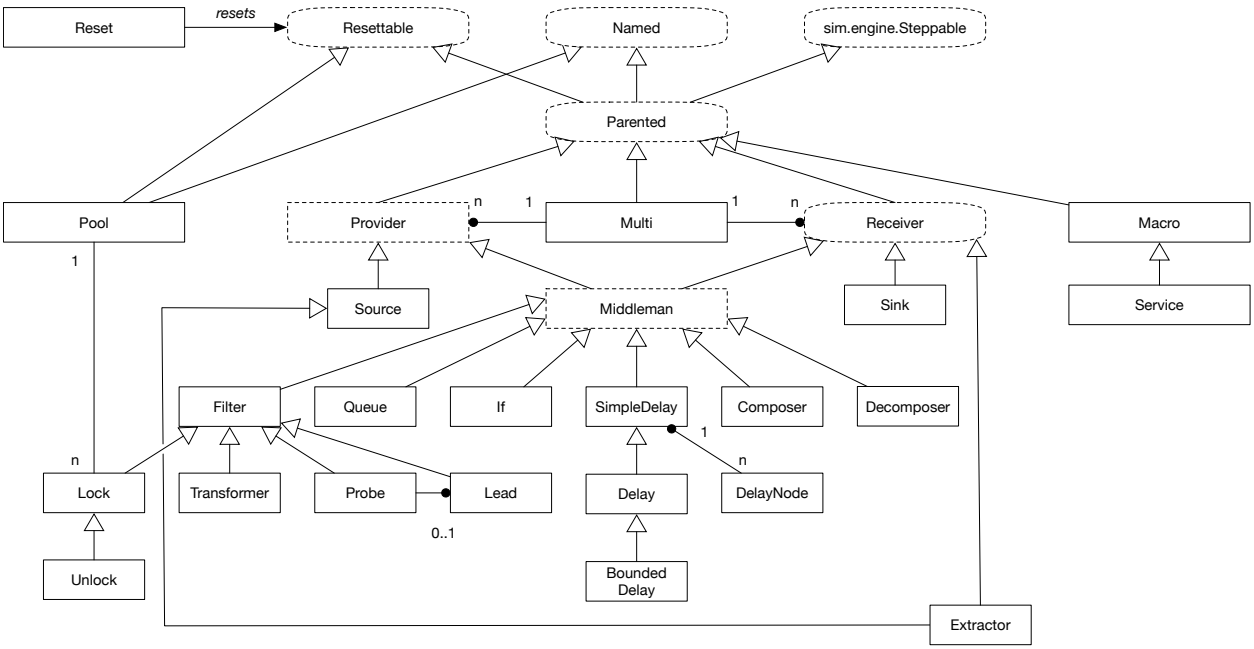
public Resource getResource()
Returns the resource.

public double getTimestamp()
Returns the timestamp.

public Provider getProvider()
Returns the provider.

public boolean isDead()
Returns whether the Resource has been marked dead.

public void setDead(boolean val)
Marks the Resource as dead.



2 About Processes

Processes are the verbs of the DES system. They perform actions in response to *events* (or if you like, *signals*) they receive.

An event can be one of four things typically:

- Another Process can *offer* some Resources to the Process.
- Another Process can *request* that the Process offer it a resource.
- Another Process can *offer a transaction* (an exchange) of a Resource for some other Resource.
- MASON's Schedule can send an event to the Process to wake it up to do some work.

Various Process objects can be **Receivers** or **Providers**, or they can be both (known as a **Middleman**). A special subclass of Middleman, called a **Filter**, is designed to pass received resources from Providers directly to downstream Receivers in zero time. Subgraphs of Process objects can be packed into a single object called a **Macro**.

Indeed most Process objects do their work in zero time, the notable exception being various kinds of **Delays**.

Most Process objects are meant to receive and/or provide a resource of a single type. If you need a Process which works with several types of Resources, you can use a **Multi**, which provides several Receivers and Provides that work in concert. For example, you might use a Multi to represent a factory which takes steel, labor, and electricity and produces bicycles and waste (five different Resource types).

Process objects inherit from a variety of high-level abstract classes and interfaces, so let's start there.

3 Abstract Process Objects

3.1 Resettable and Reset

All Processes are Resettable, meaning they can be **reset**. A convenience Reset object is provided which can reset multiple Resettable objects at once.

sim.des.Resettable Methods

public void reset(SimState state)
Resets the object to its initial state.

sim.des.Reset Constructor Methods

public Reset(SimState state)
Constructs a Reset object.

sim.des.Reset Methods

public void reset()
Resets all the objects stored in the Reset.

public void add(Resettable r)
Stores an object in the Reset.

public void remove(Resettable r)
Removes an object from the Reset.

3.2 Named

All Processes are Named, meaning that they have a **name** (a String).

sim.des.Named Methods

public String getName()
Returns the name, or null.

public void setName(String name)
Sets the name.

3.3 sim.engine.Steppable

A Steppable is an object which can be placed on MASON's schedule to be stepped in the future. All processes are Steppable, but only a few need it. [Perhaps in the future we may be more selective in this regard].

sim.engine.Steppable Methods

public void step(SimState state)
Steps the Steppable to do work.

3.4 Parented

Processes are Parented, meaning that they have a parent who "owns" the Process and will take care of it, stepping it and resetting it as necessary. All Parented objects are Resettable, Named, and sim.engine.Steppable.

sim.des.Parented Methods

public Object getParent()
Returns the parent, or null if no parent.

public void setParent(Object parent)
Sets the parent. Pass in null to remove the parent.

3.5 Receiver

An object which can *receive* an offer from a Provider. That is, the Provider will offer the Receiver some amount of Resource, asking it to **accept** the offer. Offers have minimum and maximum amounts that the Receiver may select from. The receiver accepts some appropriate amount (between the min and max) of the Resource being offered by the Provider, or it refuses it. A Receiver can also go to a Provider and **ask** that the Provider make an offer to the Receiver. Receivers have a **typical Resource type** of the resource that they accept. Rarely, a Receiver can receive more than one resource.

sim.des.Receiver Methods

public Resource getTypicalReceived()
Each Receiver is designed to receive a certain type of Resource called the *typical* Resource. This method returns a Resource of this type for type comparison purposes.

public void setRefusesOffers(boolean value)
Sets whether this Receiver should unilaterally refuse all offers.

public boolean getRefusesOffers()
Returns whether this Receiver is presently unilaterally refusing all offers (that is, setRefusesOffers(...) was turned on).

public boolean accept(Provider provider, Resource resource, double atLeast, double atMost)
This method is called to ask the Receiver to **accept an offer** of some amount of a given resource from a given provider. If the Receiver accepts some of the resource, it returns true, else it returns false if it refuses the resource. This method may throw a RuntimeException if the resource does not match the typical resource of the receiver, or if a cycle was detected in accepting offers (A offers to B, which offers to C, which then offers to A). It should be the case that $0 \leq atLeast \leq atMost \leq$ resource.getAmount(), else a RuntimeException will be thrown. Though atLeast may be set to 0, Receivers must never accept 0 of any resource: they must always accept more than 0, or else refuse. If the Receiver is currently refusing all offers, this method should always return false.

If the Resource is an Entity, the Receiver takes control of the entire Resource. *atLeast* and *atMost* may be ignored (and in general should be set to 0 and 1 respectively).

If the Receiver is a CountableResource (or an UncountableResource or Money etc.), then the Receiver may remove between *atLeast* and *atMost* (inclusive) amount of the provided Resource.

it returns true, else it returns false if it refuses the resource. If the Resource is an Entity, the Receiver takes control of the entire Resource. *atLeast and atMost* may be ignored (and in general should be set to 0 and 1 respectively).

3.6 Provider

An object which can make offers to one or more Receivers. Receivers are **registered** with the Provider. When the Provider has a resource to offer, it will go to each of its Receivers according to some **offer policy** and ask them to **accept** offers of the Resource until it is depleted. Providers can also be asked by a Receiver to make an offer to it. Providers have a **typical Resource type** of the resource that they offer. Rarely, a Provider can provide more than one resource.

sim.des.Provider Constructor Methods

public Provider(SimState state, Resource typicalProvided)
Constructs a Provider object given the provided SimState and typical provided resource.

---

sim.des.Provider Methods

public SimState getState()
Returns the SimState model object.

public Resource getTypicalProvided()
Each Provider is designed to provide a certain type of Resource called its *typical* Resource. This method returns a Resource of this type for type comparison purposes.

Storing Resources A Provider maintains a pool of resources to offer to downstream Receivers. This pool can be one of two types: either a single collected Resource, or an ordered list of available Entities. Accordingly Providers have two **protected** variables:

- `protected CountableResource resource;`
- `protected LinkedList<Entity> entities;`

One of these will be null. This is determined by the typical provided Resource type of the Provider. If the type is a subclass of CountableResource, then entities will be nulled out. If the type is a subclass of Entity, then resource will be nulled out. Subclasses of Provider are free to add or remove from the non-null variable: but you should not change which one is null and which is non-null.

sim.des.Provider Methods

public SimState clear()
Clears the stored resources.

public double getAvailable()
Returns the total amount of stored resources. If the resources are CountableResources, this returns resource.getAvailable() If the resources are Entities, this returns entities.size()

public Entity[] getEntities()
Returns in an array all the current Entities the Provider can provide. You can modify the array (it's yours), but do not modify the Entities stored inside, as they are the actual Entities stored in the Provider. If this Provider does not provide Entities, then null is returned.

public Entity getEntity(int entityNumber)

Returns the available entity with the given number for inspection, but does not remove it from the available pool. Numbers range from 0 to (int)getAvailable(); You probably should not modify this Entity: it is the actual Entity in the Provider and is owned by the Provider. If this Provider does not offer entities, or if the entityNumber is invalid, an exception is thrown.

Registering Receivers A Provider maintains a list of Receivers who have registered themselves to receive offers from it. Note that Receivers do not maintain a corresponding list of Providers.

sim.des.Provider Methods

public boolean addReceiver(Receiver receiver)
Registers a Receiver. Receivers may not be registered multiply with the same Provider. Returns false if the Receiver had already been registered with this Provider.

public boolean removeReceiver(Receiver receiver)
Deregisters and removes a Receiver. Returns false if the Receiver was not registered.

public ArrayList<Receiver> getReceivers()
Returns all registered receivers.

Making Offers This is what Providers do: they make offers to Receivers. To do this, a Provider subclass would call offerReceivers() or [rarely] offerReceivers(...). Then the Provider will offer resources to its Receivers according to its **offer policy** and **offer order**.

The offer policy governs the way and order in which offers are made to Receivers. The options are:

```
/** Offer Policy: offers are made to the first receiver, then the second, and so on,
     until available resources or receivers are exhausted. */
public static final int OFFER_POLICY_FORWARD = 0;

/** Offer Policy: offers are made to the last receiver, then the second to last,
     and so on, until available resources or receivers are exhausted. */
public static final int OFFER_POLICY_BACKWARD = 1;

/** Offer Policy: offers are made to the least recent receiver, then next, and so on,
     until available resources or receivers are exhausted. */
public static final int OFFER_POLICY_ROUND_ROBIN = 2;

/** Offer Policy: offers are made to the receivers in a randomly shuffled order,
     until available resources or receivers are exhausted. */
public static final int OFFER_POLICY_SHUFFLE = 3;

/** Offer Policy: offers are made to only one random receiver, chosen via an
     offer distribution or, if the offer distribution is null, chosen uniformly. */
public static final int OFFER_POLICY_RANDOM = 4;

/** Offer Policy: offers are made to only one receiver, chosen via selectReceiver. */
public static final int OFFER_POLICY_SELECT = 5;
```

Note the RANDOM option. In this option, a receiver may be drawn from an **offer distribution** which you can provide. Also note the SELECT option. Here, a method called selectReceiver(...) is called to determine which Receiver to make an offer to: you override this method to provide the Receiver.

The offer order governs the order in which Entities are extracted from the entities pool and offered to Receivers. If the typical resource is instead a CountableResource subclass, then the offer order had no effect. The options are:

```
/** First in First Out Offer Order for entities. */
public static final int OFFER_ORDER_FIFO = 0;

/** Last in First Out Offer Order for entities. */
public static final int OFFER_ORDER_LIFO = 1;
```

If the Provider is offering Entities, by default it only offers a single Entity in an offer period. Alternatively you can set it to attempt to offer everything it has.

Offers can be made **take it or leave it**: the whole offer must be accepted or none at all. You can also temporarily turn off the Provider's ability to make any offers at all.

Offers should not be cyclic: that is, Provider A should not offer to Provider B, which in zero time turns around and offers to Provider A. To check for this, while a Provider is making offers, it sets a flag to indicate it is offering. This flag, isOffering(), is examined by subclasses to break cycles.

sim.des.Provider Methods

protected boolean offerReceivers()
: Offers registered Receivers existing Resources.

protected boolean offerReceivers(ArrayList<Receiver> receivers)
: Offers the provided Receivers existing Resources.

protected boolean offerReceiver(Receiver receiver, Entity entity)
: Offers the given entity to the given receiver, returning true if it was accepted. You probably should not override this method; instead you probably want to override offerReceiver(Receiver, double) if at all.

protected boolean offerReceiver(Receiver receiver, double atMost)
: Makes an offer of up to the given amount to the given receiver. If the typical provided resource is an ENTITY, then atMost is ignored. Returns true if the offer was accepted.

  If the resource in question is an ENTITY, then it is removed according to the current OFFER ORDER. If the offer order is FIFO (default), then the entity is removed from the FRONT of the entities linked list (normally entities are added to the END of the linked list via entities.add()). If the offer order is LIFO, then the entity is removed from the END of the entities linked list. Then this entity is offered to the receiver by calling offerReceiver(receiver, entity).

  The only real reason for the atMost parameter is so that receivers can REQUEST to be offered atMost resource from a provider.

protected boolean isOffering()
: Returns true if the Provider is currently making offers and so should not be receiving any offers (no cycles).

public void setOffersTakeItOrLeaveIt(boolean val)
: Sets whether receivers are offered take-it-or-leave-it offers. A take-it-or-leave-it offer requires the Receiver to accept all of the offered Resource, or else reject it all.

public boolean getOffersTakeItOrLeaveIt()
: Returns whether receivers are offered take-it-or-leave-it offers. A take-it-or-leave-it offer requires the Receiver to accept all of the offered Resource, or else reject it all.

public void setOffersAllEntities(boolean val)
: Sets whether the Provider will, during offerReceivers(...), attempt to offer every single entity that it has available, until offers start to be refused by downstream receivers. By fault this is FALSE: the Provider offers only one Entity. This only matters if the Provider provides entities. This capability is largely useful for Delays and SimpleDelays rather than other kinds of Providers.

public boolean getOffersAllEntities()
: Returns whether the Provider will, during offerReceivers(...), attempt to offer every single entity that it has available, until offers start to be refused by downstream receivers. By fault this is FALSE: the Provider offers only one Entity. This only matters if the Provider provides entities. This capability is largely useful for Delays and SimpleDelays rather than other kinds of Providers.

public Receiver selectReceiver(ArrayList<Receiver> receivers, Resource resource)
: If the offer policy is OFFER_POLICY_SELECT, then when the receivers are non-empty, this method will be called to specify which receiver should be offered the given resource. Override this method as you see fit. The default implementation simply returns the first one.

protected void selectedOfferAccepted(Receiver receiver, Resource originalResource, Resource revisedResource)
If the offer policy is OFFER_POLICY_SELECT, then if a receiver accepts an offer of a resource, this method is called, with (a copy of) the original resource, the revised resource after then receiver accepted it. If the resource was an ENTITY, then the revised resource will likely be unchanged. If the resource was a COUNTABLE RESOURCE, then the revised resource will be reduced by the amount that the receiver accepted (relative to the original resource).

public void setOfferOrder(int offerOrder)
Sets the offer order as provided.

public int getOfferOrder()
Returns the offer order.

public void setOfferPolicy(int offerPolicy)
Sets the receiver offer policy as provided.

public int getOfferPolicy()
Returns the receiver offer policy

public void setOfferDistribution(double[] distribution)
Sets the receiver offer policy to OFFER_POLICY_RANDOM, and sets the appropriate distribution for selecting a receiver. If null is provided for the distribution, receivers are selected randomly. Selection via an offer distribution works as follows: a random integer is selected from the distribution. An offer is made to the registered receiver corresponding to the index of the selected slot in the distribution. The distribution must exactly match the size of the number of registered receivers.

public void setOfferDistribution(sim.util.distribution.AbstractDiscreteDistribution distribution)
Sets the receiver offer policy to OFFER_POLICY_RANDOM, and sets the appropriate distribution for selecting a receiver. If null is provided for the distribution, receivers are selected randomly. Selection via an offer distribution works as follows: a random integer is selected from the distribution. An offer is made to the registered receiver corresponding to the index of the selected slot in the distribution. The distribution must exactly match the size of the number of registered receivers.

public sim.util.distribution.AbstractDistribution getOfferDistribution()
Returns the current offer distribution, or null if none.

public void setMakesOffers(boolean value)
Sets whether the Provider will make offers (or refuse to do so in all situations).

public boolean getMakesOffers()
Returns whether the Provider will make offers (or refuse to do so in all situations).

Requesting Offers You can request a Provider to make an offer to a Receiver. The Provider may or may not attempt it (and the Receiver may or may not accept it).

sim.des.Provider Methods

public boolean provide(Receiver receiver)
Asks the Provider to make a unilateral offer to the given Receiver. This can be used to implement a simple pull. The Receiver does not need to be registered with the Provider. Returns true if the offer was accepted; though since the Receiver itself likely made this call, it's unlikely that this would ever return anything other than TRUE in a typical simulation.

public boolean provide(Receiver receiver, double atMost)
Asks the Provider to make a unilateral offer of up to the given amount to the given Receiver. If the typical provided resource is an ENTITY, then atMost is ignored. This can be used to implement a simple pull. The Receiver does not need to be registered with the Provider. Returns true if the offer was accepted; though since the Receiver itself likely made this call, it's unlikely that this would ever return anything other than TRUE in a typical simulation.

atMost must be a positive non-zero, non-NAN number.

public boolean requestEntity(Receiver receiver, int entityNumber)
Asks the Provider to offer to the given receiver entity #entityNumber in its entities list. You can get this entity number by requesting getEntities(), then returning the index of the entity of interest in the resulting array. If you want to grab multiple entities, call this method multiple times; but beware that as you pull entities out, the entity list shrinks and the indexes change. The easiest way to deal with this is to call getEntities() once, and then request entities one by one going *backwards* through the resulting list. If this Provider does not offer entities, or if the entityNumber is invalid, an exception is thrown.

Offer Statistics Offers can be accepted or rejected by Receivers. The Provider maintains statistics on the most recent accepted offers, their receivers, and the time were accepted. The Provider also maintains statistics on the total amount of accepted offer resource and the rate over time that it was accepted.

sim.des.Provider Methods

public ArrayList<Resource> getLastAcceptedOffers()
Returns the most recent offers accepted.

public ArrayList<Receiver> getLastAcceptedOfferReceivers()
Returns the receivers for the most recent offers accepted.

public double getLastAcceptedOfferTime()
Returns the timestamp for the most recent offers made.

public double getTotalOfferResource()
Returns the total amount of resource accepted when offered to downstream Receivers so far.

public double getOfferResourceRate()
Returns the rate of accepted resource so far.

3.7 MiddleMan

An object which is both a Provider and a Receiver. You can make your own Provider+Receiver combination, but MiddleMan is a nice abstract superclass. Middlemen can have different typical received and provided resources. Middlemen also have a special trick up their sleeves: because they are both Providers and Receivers, they can perform **transactions**, that is, exchanging one kind of Resource for possibly another.

sim.des.Middleman Constructor Methods

public Middleman(SimState state, Resource typical)
Builds a Middleman with the given state, and using the same Resource for both typical provided and typical received resources (subclasses may deviate from this).

---

sim.des.Middleman Methods

public boolean accept(Provider provider, Resource resource, double atLeast, double atMost)
Offers a resource from a Provider to the Middleman. By default it does nothing: it returns FALSE, indicating that the offer is refused. You can override this as you see fit. This isn't abstract because you might wish to use a custom Middleman to conduct transactions only, rather than accepting offers. For more details, see Receiver.accept(...). Implementation suggestions are provided in the Javadocs for Middleman.java

protected Resource performTransaction(Provider provider, Receiver receiver, Resource provided, double atLeast, double atMost, Resource requestedType, double atLeastRequested)

Received by the Middleman when a Provider and Receiver are asking for a transaction of one resource for another. The Provider would provide a resource to the Middleman and a Receiver would receive the transacted returned Resource. Very commonly this Provider and Receiver are one and the same: they are also a Middleman or perhaps a Multi. But this does not have to be the case. If the transaction is agreed to, you should modify the provided resource and return the requested resource. Otherwise, return null. The default form simply returns null.

By the time this method has been called, refuses-offers, cyclic, and type compatibility checks have already been performed, but you might still benefit from knowing the requestedType, so it is provided: but you should not modify this resource nor return it.

The transaction is offering atLeast and atMost a certain amount of provided resource in exchange for (from you) a requested resource. atLeastRequested is the amount of requested resource to be provided in exchange for the *least* amount of provided resource. If you decide to take some X provided resource where X is between atLeast and atMost, then the resource amount you provide in return X * atLeastRequested / atMost. For requested CountableResources, I suggest that the amount returned in response to a request would best be (int)(X * atLeastRequested / atMost) but you can do as your model deems appropriate.

For Entities, only a single Entity can be provided. If an Entity is being provided, then atLeast = atMost = 1.

For Entities, only a single Entity can be requested. If an Entity is being requested, atLeastRequested = 1 and exactly one Entity should be returned regardless of its value.

public Resource transact(Provider provider, Receiver receiver, Resource provided, double atLeast, double atMost, Resource requestedType, double atLeastRequested)

You may call this method in order to request a transaction of one resource for another. The Provider would provide a resource to the Middleman and a Receiver would receive the transacted returned Resource. Very commonly this Provider and Receiver are one and the same: they are also a Middleman. But this does not have to be the case. If the transaction is agreed to, your provided resource will be accordingly modified (reduced) and the requested resource will have been provided. Otherwise null will be returned.

The transaction is offering atLeast and atMost a certain amount of provided resource in exchange for (from you) a requested resource. atLeastRequested is the amount of requested resource to be provided in exchange for the *least* amount of provided resource. If you decide to take some X provided resource where X is between atLeast and atMost, then the resource amount you provide in return is X * atLeastRequested / atMost. For requested CountableResources, I suggest that the amount returned in response to a request would best be (int)(X * atLeastRequested / atMost) but you can do as your model deems appropriate.

For Entities, only a single Entity can be provided. If an Entity is being provided, then atLeast = atMost = 1.

For Entities, only a single Entity can be requested. If an Entity is being requested, atLeastRequested = 1 and exactly one Entity should be returned regardless of its value, and atLeast = atMost.

Don't override this method. Instead, override performTransaction().

public Resource transact(Middleman middleman, Resource provided, double atLeast, double atMost, Resource requestedType, double atLeastRequested)

You may call this method in order to request a transaction of one resource for another. The other Middleman would provide a resource to this Middleman and would receive the transacted returned Resource. If the transaction is agreed to, your provided resource will be accordingly modified (reduced) and the requested resource will have been provided. Otherwise null will be returned.

The transaction is offering atLeast and atMost a certain amount of provided resource in exchange for (from you) a requested resource. atLeastRequested is the amount of requested resource to be provided in exchange for the *least* amount of provided resource. If you decide to take some X provided resource where X is between atLeast and atMost, then the resource amount you provide in return is X * atLeastRequested / atMost. For requested CountableResources, I suggest that the amount returned in response to a request would best be (int)(X * atLeastRequested / atMost) but you can do as your model deems appropriate.

For Entities, only a single Entity can be provided. If an Entity is being provided, then atLeast = atMost = 1.

For Entities, only a single Entity can be requested. If an Entity is being requested, atLeastRequested = 1 and exactly one Entity should be returned regardless of its value, and atLeast = atMost.

Don't override this method. Instead, override performTransaction().

---

4 Basic Process Objects

4.1 Sink

A Receiver which accepts all offers (of the appropriate Resource type) and throws the resulting Resource away. **By default, this class's step() method does nothing, so there's no need to schedule it.**

sim.des.Sink Constructor Methods

public Sink(SimState state, Resource typicalReceived)
Builds a Sink with the given state and typical received resource.

4.2 Source

A Provider which generates Resources and offers it to downstream registered Receivers. You can customize your Source however you want, but the default form works as follows. Each time the Source is ready to **produce** some Resource, the amount it produces is determined either by a **distribution** or a **deterministic amount**. It then adds this Resource to a pile, and offers all the Resources currently in the pile to its receivers. Sources have a maximum **capacity** for their pile and will not generate more than this. After making offers, the Source then uses *another* distribution or deterministic amount to determine the **next time** it will generate Resources and make offers. It schedules itself appropriately.

sim.des.Source Constructor Methods

public Source(SimState state, Resource typicalProvided)
Builds a Source with the given state and typical provided resource.

---

sim.des.Source Methods

public double getCapacity()
Returns the maximum available resources that may be built up.

public void setCapacity(double d)
Set the maximum available resources that may be built up.

public void setRateDistribution(AbstractDistribution rateDistribution)
Sets the distribution used to determine the rate at which the source produces resources. When the source is update()ed (via a step() method), it draws from this distribution the next time at which it should schedule itself to be stepped() again. If this distribution is null, it instead uses getRate() to deterministically acquire the next timestep. Note that the if the time is currently Schedule.EPOCH, no resources will be produced this timestep.

public AbstractDistribution getRateDistribution()
Returns the distribution used to determine the rate at which the source produces resources. When the source is update()ed (via a step() method), it draws from this distribution the next time at which it should schedule itself to be stepped() again. If this distribution is null, it instead uses getRate() to deterministically acquire the next timestep. Note that the if the time is currently Schedule.EPOCH, no resources will be produced this timestep.

When a value is drawn from this distribution to determine delay, it will be put through Absolute Value first to make it positive. Note that if your distribution covers negative regions, you need to consider what will happen as a result and make sure it's okay (or if you should be considering a positive-only distribution).

public void setRate(double rate)
Sets the deterministic rate for producing resources. If the rate distribution is null, then the deterministic rate is used instead as follows.

Throws a runtime exception if the rate is negative or NaN.

public double getRate()
Returns the deterministic rate for producing resources. If the rate distribution is null, then the deterministic rate and random offset are used instead as follows. If the Source is initially scheduled for Schedule.EPOCH, then at that time it does not produce any resources, but rather determines the initial time to reschedule itself. If the random offset is TRUE then the initial time will be the EPOCH plus a uniform random value between 0 and the deterministic rate. If the random offset is FALSE then the initial time will simply be the EPOCH plus the deterministic rate. Thereafter the next scheduled time will be the current time plus the rate.

public void setProductionDistribution(AbstractDistribution productionDistribution)
Sets the distribution used to determine how much resource is produced each time the Source decides to produce resources. If this is null, then the determinstic production value is used instead.

public AbstractDistribution getProductionDistribution()
Returns the distribution used to determine how much resource is produced each time the Source decides to produce resources. If this is null, then the determinstic production value is used instead.

Depending on your needs, you might wish to select a discrete distribution rather than a continuous one.

When a value is drawn from this distribution to determine delay, it will be put through Absolute Value first to make it positive. Note that if your distribution covers negative regions, you need to consider what will happen as a result and make sure it's okay (or if you should be considering a positive-only distribution).

public void setProduction(double amt)
Sets the deterministic production value used to determine how much resource is produced each time the Source decides to produce resources (only when there is no distribution provided).

Throws a runtime exception if the rate is negative, zero, or NaN.

public void getProduction()
Returns the deterministic production value used to determine how much resource is produced each time the Source decides to produce resources (only when there is no distribution provided).

protected Entity buildEntity()
Produces ONE new entity to add to the collection of entities. By default this is done by duplicating the typical provided entity. You can override this if you feel so inclined.

protected void buildEntities(double amt)
Builds *amt* number of Entities and adds them to the entities list. The amount could be a real-value, in which it should be simply rounded to the nearest positive integer ≥ 0. By default this generates entities using buildEntity().

protected void buildResource(double amt)
Builds *amt* of Countable or Uncountable Resource and adds it to the resource pool. By default this simply adds resource out of thin air.

protected void update()
This method is called once every time this Source is stepped, and is used to produce new resources or entities and add them to the available pool in the Source. You can override this method to add them as you see fit (you can check to see if you should add entities by seeing if the entities variable is non-null: otherwise you should be adding to the existing resource).

The modeler can set the RATE at which production occurs by setting either a deterministic RATE (via setRate()) or by setting a distribution to determine the rate (via setRateDistribution() – you may find sim.util.distribution.Scale to be a useful utility class here).

The modeler can also change the AMOUNT which is produced by setting either a deterministic PRODUCTION (via setProduction()) or by setting a distribution to determine the production (via setProductionDistribution() — again you may find sim.util.distribution.Scale to be a useful utility class here). Note that if the distribution produces a negative value, the absolute value is used.

By default update() then works as follows.

1. First, if we are autoscheduling, we need to reschedule ourselves.
   (a) If we have a rate distribution, select from the distribution and add to our current time to get the rescheduling time. The distribution value should be ¿= 0, else it will be set to 0.
   (b) If we have a fixed rate, add it to our current time to get the rescheduling time.
   (c) If the resulting rescheduling time hasn't changed (we added 0 to it, probably an error), reschedule at the immediate soonest theoretical time in the future. Else schedule at the rescheduling time.
2. Next, if we're $\geq$ capacity, return.
3. Othrwise we determine how much to produce.
   (a) If we are producing a deterministic production amount, then we use the production amount.
   (b) If we are producing an amount determined by a distribution, then we select a random value under this distribution.

New entities are produced by calling the method buildEntity().

Total production cannot exceed the stated capacity.

Note that when a value is drawn from either the RATE or PRODUCTION distributions, it will be put through Absolute Value first to make it positive. Note that if your distribution covers negative regions, you need to consider what will happen as a result and make sure it's okay (or if you should be considering a positive-only distribution).

public void step(SimState state)
Upon being stepped, the Source updates its resources (potentially building some new ones), then makes offers to registered receivers. If you are automatically rescheduling, you don't have to schedule the Source at all; it'll handle it.

public void autoScheduleAt(double time)
A convenience method which calls setAutoSchedules(true), then schedules the Source on the Schedule using the current rescheduleOrdering. The Source is initially scheduled at the given time. See also autoScheduleNow() and autoSchedule(...) for other options.

public void autoScheduleNow()
A convenience method which calls setAutoSchedules(true), then schedules the Source on the Schedule using the current rescheduleOrdering. The Source is scheduled for the next possible time within epsilon, or if we're currently before the simulation epoch, as you probably should be, then the time is set to Schedule.EPOCH, that is, 0.0. You should only call this method ONCE at the beginning of a run. See also autoSchedule(...) and autoScheduleAt() for other options.

public void autoSchedule(boolean offset)
A convenience method which calls setAutoSchedules(true), then schedules the Source initially on the Schedule using the current rescheduleOrdering. The Source is scheduled initially in one of two ways. If OFFSET is FALSE, then the source is scheduled by selecting the next rate value, either fixed, or from a distribution, and using that as the time. For example, if you have a fixed rate of 2.0, and you are at the beginning of a simulation run, then the Source is scheduled for 2.0; or if you have a rate distribution, then a value is selected at random, from this distribution and the Source is scheduled for that.

However if OFFSET is TRUE, then the Source attempts to be scheduled at a random offset so as to simulate a Source whose process is ongoing as of the commencement of the simulation. This is done by once again selecting the next rate value, either fixed or from a distribution. However then we select a random time from between 0 and that value inclusive. If you are using a distribution and it is a sim.util.distribution.AbstraceDiscreteDistribution, then this random time will be an integer, else it will be a real value.

See also autoScheduleNow(...) and autoScheduleAt() for other options.

public void setAutoSchedules(boolean val)
Sets whether the Source reschedules itself automatically using either a deterministic or distribution-based rate scheme. If FALSE, you are responsible for scheduling the Source as you see fit. If TRUE, then the ordering used when scheduling is set to 0.

public boolean getAutoSchedules()
Returns whether the Source reschedules itself automatically using either a deterministic or distribution-based rate scheme.

public int getRescheduleOrdering()
Returns the reschedule ordering.

public void setRescheduleOrdering(int ordering)
Sets the reschedule ordering.

4.3 Extractor

Both a Receiver and a Source (sort of). An Extractor is works like a Source, except that it "builds" its resources by extracting them from another Provider. You could, for example, attach an Extractor to a Queue and every once i a while the Extractor would pull out of the Queue and offer resources to downstream Receivers. Extractors ignore capacity.

Extractors make requests of their Providers according to a **request policy**. The policies are:

```
/** Request Policy: requests are made to the first provider, then the second, and so on. */
public static final int REQUEST_POLICY_FORWARD = 0;
/** Request Policy: requests are made to the last provider, then the second to last,
     and so on. */
public static final int REQUEST_POLICY_BACKWARD = 1;
/** Request Policy: requests are made to the providers in a randomly shuffled order. */
public static final int REQUEST_POLICY_SHUFFLE = 2;
/** Request Policy: requests are made to only one random provider, chosen via an
     offer distribution or, if the offer distribution is null, chosen uniformly. */
public static final int REQUEST_POLICY_RANDOM = 3;
/** Request Policy: requests are made to only one provider, chosen via selectProvider. */
public static final int REQUEST_POLICY_SELECT = 4;
```

In the case of the FORWARD, BACKWARD, and SHUFFLE policies, Extractors can make requests of their Providers in different ways. First, an Extractor can go through every one of its providers, providing those Resources any of them are willing to offer. Second, an Extractor can rummage through its providers until it finds one willing to provide the request Resources, then provide only that one. Third, an Extractor can go through its providers, providing the Resources they provide *until* one of them refuses. These are:

```
public static final int REQUEST_TERMINATION_EXHAUST = 0;
public static final int REQUEST_TERMINATION_FAIL = 1;
public static final int REQUEST_TERMINATION_SUCCEED = 2;
```

In the case of the RANDOM policy, an Extractor will choose a Provider at random either uniformly or using a provided distribution.

In the case of the SELECT policy, an Extractor will choose a Provider by calling its selectProvider(...), which you may override to provide the Provider of interest.

Requesting from the Provider should not be cyclic: that is, Provider A should not offer to the Extractor, which in zero time turns around and offers to Provider A. To check for this, while an Extractor is making requests, it sets a flag to indicate it is requesting. This flag, isRequesting(), is examined by subclasses to break cycles.

sim.des.Extractor Constructor Methods

public Source(SimState state, Resource typical)
Builds a Source with the given state and typical provided and received resource.

public Source(SimState state, Resource typical, Provider provider)
Builds a Source with the given state and typical provided and received resource, plus a single initial Provider.

---

sim.des.Extractor Methods

public boolean addProvider(Provider provider)
Registers a provider with the Extractor. Returns false if the receiver was already registered.

public ArrayList¡Provider¿ getProviders()
Returns all registered providers.

public boolean removeProvider(Provider provider)
Unregisters a provider with the Extractor. Returns false if the provider was not registered.

public void setRequestPolicy(int requestPolicy)
Sets the request policy.

public int getRequestPolicy()
Returns the request policy.

public void setRequestDistribution(AbstractDiscreteDistribution distribution)
Sets the receiver request policy to REQUEST_POLICY_RANDOM, and sets the appropriate distribution for selecting a provider. If null is provided for the distribution, providers are selected randomly. Selection via an request distribution works as follows: a random integer is selected from the distribution. If this integer is < 0 or $\geq$ the number of providers registered, then a warning is produced and no request is made (this should NOT happen). Otherwise a request is made to the registered provider corresponding to the selected integer.

public int getRequestPolicy()
Sets the receiver offer policy to REQUEST_POLICY_RANDOM, and sets the appropriate distribution for selecting a provider. If null is provided for the distribution, providers are selected randomly. Selection via an offer distribution works as follows: a random integer is selected from the distribution. An offer is made to the registered provider corresponding to the index of the selected slot in the distribution. The distribution must exactly match the size of the number of registered providers.

public int getRequestPolicy()
Returns the current offer distribution, or null if none.

public void setRequestTermination(int requestTermination)
Sets the request termination type.

public int getRequestTermination()
Returns the request termination type.

protected boolean isRequesting()
Returns true if the Extractor is currently requesting an offer (this is meant to allow you to check for offer cycles.

protected boolean requestProviders(double amt)
Requests the given amount from providers according to the request policy and termination type

public Provider selectProvider(ArrayList<Provider> providers)
If the provider policy is REQUEST_POLICY_SELECT, then when the providers are non-empty, this method will be called to specify which provider should be asked to offer a resource. Override this method as you see fit. The default implementation simply returns the first one.

public boolean provide(Receiver receiver)
Makes a request of the upstream Providers, then provides it if possible.

public boolean provide(Receiver receiver, double atMost)
Makes a request of the upstream Providers, then provides it if possible.

protected void buildEntities(double amt)
Builds a single entity, ignoring the amount passed in, by asking the provider to provide it. See Source.

protected void buildResource(double amt)
Builds resource by asking the provider to provide it. See Source.

public Resource getTypicalReceived()
Returns the typical resource received: this is the same as the typical resource provided.

public boolean accept(Provider provider, Resource amount, double atLeast, double atMost)
Accepts resource, normally from an upstream provider in response to a request, and immediately offers it.

4.4 Queue

A Middleman. When a Queue receives a Resource, it adds it to a pile, then (usually) immediately offers this pile to its own receivers. The Queue's pile has a maximum capacity. If it cannot add to the pile, it will refuse the offer.

In addition to maintaining statistics on total provided resources, a Queue also maintains statistics on total received resources.

sim.des.Queue Constructor Methods

public Queue(SimState state, Resource typical)
Builds a Queue with the given state and typical provided and received resource.

---

sim.des.Queue Methods

public void setCapacity(double d)
Set the maximum available resources that may be aquired by the Queue. Throws a runtime exception if the capacity is negative or NaN.

public double getCapacity()
Returns the maximum available resources that may be aquired by the Queue.

public void setOffersImmediately(boolean val)
Sets whether the Queue offers items immediately upon accepting (when possible) in zero time, as opposed to when it is stepped.

public boolean getOffersImmediately()
Returns whether the Queue offers items immediately upon accepting (when possible) in zero time, as opposed to when it is stepped.

public double getTotalReceivedResource()
Returns the total amount of resource received to date.

public double getReceiverResourceRate()
Returns the average rate of resources received to date.

public void reset(SimState state)
Resets the total resource received to date.

public void step(SimState state)
Offers to receivers. You can use this to make the Queue occasionally offer to receivers in addition to doing so when receiving resources.

public boolean accept(Provider provider, Resource amount, double atLeast, double atMost)
Accepts resources up to the given capacity, then if offering immediately, turns around and offers them in zero time to downstream Receivers.

4.5 SimpleDelay

A Middleman. When a SimpleDelay receives a Resource, it adds it to a linked list and, after a delay of some amount of time, then offers the Resource to downstream receivers. If the receivers do not accept, then the Resource is (usually) discarded.

There are three kinds of delays. A *SimpleDelay* delays all resources by the same fixed amount of time, and is implemented internally with a LinkedList. A *Delay* allows resources to have variable delay times relative to one another, and in fact have random delay times. It is implemented with a binary heap, and incurs an O(lg n) insertion and removal cost. A *Bounded Delay* is like a Delay except that the delay times must be integers between 0 and some *m* exclusive. This is implemented as an array, and is as fast as SimpleDelay.

The SimpleDelay has a maximum capacity of items it may store which are presently delayed. The Resources held up in the SimpleDelay are called the *delayed resources*, and the Resources being offered are called *available* or *ripe*. The SimpleDelay has a maximum capacity of delayed resources, or optionally of delayed and ripe resources together.

If after offering Resource to receivers, there is available capacity (*slack*), the SimpleDelay can immediately ask a **slack provider** to fill that capacity.

A SimpleDelay needs to be placed on the Schedule to do its work. To make things easy, it ca be **autoscheduled**, meaning it will place itself on the Schedule as needed.

In addition to maintaining statistics on total provided resources, a SimpleDelay also maintains statistics on total received resources.

It's possible to kill Resources while in the SimpleDelay (to simulate destruction in transit, for example). When a Resource is added to a SimpleDelay it is also (optionally) added to a *lookup table* so you can kill it (see DelayNode) or for some other reason reference it. This lookup table incurs an overhead, so don't turn it on unless you need to.

sim.des.SimpleDelay Constructor Methods

public SimpleDelay(SimState state, double delayTime, Resource typical)
Builds a SimpleDelay with the given state and typical provided and received resource, and the given delay time.

public SimpleDelay(SimState state, Resource typical)
Builds a SimpleDelay with the given state and typical provided and received resource, and a delay time of 1.0.

---

sim.des.SimpleDelay Methods

public void setCapacity(double d)
Set the maximum available resources that may be built up. Throws a runtime exception if the capacity is negative or NaN.

public double getCapacity()
Returns the maximum available resources that may be built up.

public DelayNode[] getDelayedResources()
Returns in an array all the Resources currently being delayed and not yet ready to provide, along with their timestamps (when they are due to become available), combined as a DelayNode. Note that this is a different set of Resources than Provider.getEntities() returns. You can modify the array (it's yours), but do not modify the DelayNodes nor the Resources stored inside them, as they are the actual Resources being delayed.

public boolean getAutoSchedules()
Returns whether the SimpleDelay schedules itself on the Schedule automatically to handle the next timestep at which a delayed resource will become available. If you turn this off you will have to schedule the SimpleDelay yourself.

public void setAutoSchedules(boolean val)
Sets whether the SimpleDelay schedules itself on the Schedule automatically to handle the next timestep at which a delayed resource will become available. If you turn this off you will have to schedule the SimpleDelay yourself.

public void clear()
Clears all resources currently in the SimpleDelay.

public double getSize()
Returns the number of items currently being delayed.

public double getDelayed()
Returns the AMOUNT of resource currently being delayed.

public double getDelayedPlusAvailable()
Returns the AMOUNT of resource currently being delayed, plus the current available resources.

public double getDelayTime()
Returns the delay time.

public void setDelayTime(double delayTime)
Sets the delay time. In a SimpleDelay (not a Delay) this also clears the delay queue entirely, because not doing so would break the internal linked list. In a Delay, the delay queue is not cleared, and you are free to call this method any time you need to without issues. Delay times may not be negative or NaN.

public int getRescheduleOrdering()
Returns the delay ordering.

public void setRescheduleOrdering(int ordering)
Sets the delay ordering and clears the delay entirely.

public double getTotalReceivedResource()
Returns the total amount of resource received to date.

public double getReceiverResourceRate()
Returns the average rate of resources received to date.

protected void buildDelay()
Builds the delay structure. Typically you wouldn't fool with this: it's used by Delay subclasses.

public void setUsesLookup(boolean val)
Sets whether lookup is used. If TRUE, then every time a Resource is added to the SimpleDelay it is also added to a HashMap so its DelayNode can be quickly looked up with lookup(). When the Resource exits the SimpleDelay it is removed from the HashMap. This is primarily used to make it fast to set a resource as "dead" (see DelayNode). However it incurs a constant overhead and so this feature is turned off by default.

public boolean getUsesLookup()
Returns whether lookup is used. If TRUE, then every time a Resource is added to the SimpleDelay it is also added to a HashMap so its DelayNode can be quickly looked up with lookup(). When the Resource exits the SimpleDelay it is removed from the HashMap. This is primarily used to make it fast to set a resource as "dead" (see DelayNode). However it incurs a constant overhead and so this feature is turned off by default.

public DelayNode lookup(Resource resource)
Looks up a resource, if lookup is presently being used (otherwise issues a RuntimeException). This is primarily used to make it fast to set a resource as "dead" (see DelayNode). However it incurs a constant overhead and so this feature is turned off by default (see setUsesLookup()).

public boolean getIncludesAvailableResourcesInTotal()
Returns whether the ripe resources (no longer in the delay queue) should be included as part of the delay's total resource count for purposes of comparing against its capacity to determine if it's full. Note that if setDropsResourcesBeforeUpdate() is TRUE, then the ripe resources will be included PRIOR to when drop() is called, but they disappear afterwards. drop() is called during offerReceivers(), which in turn may be called during step() after update().

public void setIncludesAvailableResourcesInTotal(boolean val)
Sets whether the available resources (no longer in the delay queue) should be included as part of the delay's total resource count for purposes of comparing against its capacity to determine if it's full. Note that if setDropsResourcesBeforeUpdate() is TRUE, then the available resources will be included PRIOR to when drop() is called, but they disappear afterwards. drop() is called during offerReceivers(), which in turn may be called during step() after update().

public boolean accept(Provider provider, Resource amount, double atLeast, double atMost)
Accepts up to CAPACITY of the given resource and places it in the delay, then auto-reschedules the delay if that feature is on.

public void setDropsResourcesBeforeUpdate(boolean val)
Sets whether available resources are cleared prior to loading new delayed resources during update(). By default this is TRUE. If this is FALSE, then resources will build potentialy forever if not accepted by downstream receivers, as there is no maximum capacity to the available resources.

public boolean getDropsResourcesBeforeUpdate()
Returns whether available resources are cleared prior to loading new delayed resources during update(). By default this is TRUE. If this is FALSE, then resources will build potentialy forever if not accepted by downstream receivers, as there is no maximum capacity to the available resources.

protected void drop()
Removes all currently available resources.

protected void update()
Deletes exiting available resources, then checks the delay pipeline to determine if any resources have come available, and makes them available to registered receivers in zero time.

public void step(SimState state)
Upon being stepped, the Delay calls update() to reap all available resources. It then calls offerReceivers to make offers to registered receivers. You don't have to schedule the Delay at all, unless you have turned off auto-scheduling.

public void reset()
Clears the Delay and sets the total received resource to 0.

public Provider getSlackProvider()
Returns the slack provider. Whenever a queue's offerReceivers(...) call is made, and it has slack afterwards, it will call the slack provider to ask it to fill the slack up to capacity.

public void setSlackProvider(Provider provider)
Sets the slack provider. Whenever a queue's offerReceivers(...) call is made, and it has slack afterwards, it will call the slack provider to ask it to fill the slack up to capacity.

public double getReceiverResourceRate()
Returns the average rate of resources received to date.

public void reset(SimState state)
Resets the total resource received to date.

public void step(SimState state)
Offers to receivers. You can use this to make the Queue occasionally offer to receivers in addition to doing so when receiving resources.

public boolean accept(Provider provider, Resource amount, double atLeast, double atMost)
Accepts resources up to the given capacity, then if offering immediately, turns around and offers them in zero time to downstream Receivers.

4.6 Delay

A subclass of SimpleDelay. This is just like a SimpleDelay, except that the delay time for each object received can be *different*, so some Resources can move through the Delay faster than others. Thus the SimpleDelay is implemented using a heap rather than a linked list.

Delay times can be fixed, or selected at random under a distribution. On insertion of a Resource, its delay is computed automatically depending on the settings you have provided, via the method getDelay(...)

There are three kinds of delays. A *SimpleDelay* delays all resources by the same fixed amount of time, and is implemented internally with a LinkedList. A *Delay* allows resources to have variable delay times relative to one another, and in fact have random delay times. It is implemented with a binary heap, and incurs an O(lg n) insertion and removal cost. A *Bounded Delay* is like a Delay except that the delay times must be integers between 0 and some *m* exclusive. This is implemented as an array, and is as fast as SimpleDelay.

Below we list only the Delay methods which are significantly different from SimpleDelay.

sim.des.Delay Methods

public void setDelayTime(double delayTime)
Sets the delay time. Unlike a SimpleDelay, a Delay does not also clear its delay queue when setting the delay time: and so you are free to call this method any time you need to without issues. The default delay time is 1.0. Delay times may not be negative or NaN.

public void setUsesLastDelay(boolean val)
Sets whether getDelay(...) should simply return the delay time used by the most recent resource added to the Delay. If there is no such resource, or if that resource has since been removed from the Delay, or if its delay time has passed, then a delay value of 1.0 will be used as a default.

public boolean getUsesLastDelay()
Sets whether getDelay(...) should simply return the delay time used by the most recent resource added to the Delay. If there is no such resource, or if that resource has since been removed from the Delay, or if its delay time has passed, then a delay value of 1.0 will be used as a default.

public void setDelayDistribution(AbstractDistribution distribution)
Sets the distribution used to independently select the delay time for each separate incoming resource. If null, the value of getDelayTime() is used for the delay time.

public AbstractDistribution getDelayDistribution()
Returns the distribution used to independently select the delay time for each separate incoming resource. If null, the value of getDelayTime() is used for the delay time. When a value is drawn from this distribution to determine delay, it will be put through Absolute Value first to make it positive. Note that if your distribution covers negative regions, you need to consider what will happen as a result and make sure it's okay (or if you should be considering a positive-only distribution).

protected double getDelay(Provider provider, Resource amount)
Returns the appropriate delay value for the given provider and resource amount. You can override this as you see fit, though the defaults should work fine in most cases. The defaults are: if getUsesLastDelay(), and there has been at least one previous resource entered into the Delay already, then the most recent previous delay time is used. Otherwise if the delay distribution has been set, it is queried and its absolute value is used to produce a random delay time under the distribution (delay times may not be negative or NaN). Otherwise the fixed delay time is used (which defaults to 1.0). Override this to provide a custom delay given the provider and resource amount or type.

protected void setLastDelay(double val)
Sets the last delay value. This is used internally by Delay and BoundedDelay, don't fool with it.

protected double getLastDelay()
Returns the last delay value. This is used internally by Delay and BoundedDelay, don't fool with it.

4.7 BoundedDelay

A subclass of Delay. This is just like a Delay, except that the delay time for each object is adjusted to some (different) integer n, $0 < n \leq m$. The value m is the **maximum delay steps**. This allows BoundedDelay to be implemented as an array and so incur an O(1) insertion and removal cost.

For example, let's imagine the maximum delay steps was 5. You submitted a delay of 2.3. The actual delay would be 3. If you submitted a delay of 5.2, it would be rejected.

BoundedDelay can further restrict delay times to fall along **delay intervals**. For example, BoundedDelay could force delay times to be one of 1, 2, 3, 4, 5, 6, 7; or it could force them to be one of 1, 3, 5, 7; or 1, 4, 7, etc. This is mostly done to allow the arrays to be smaller even for large delay times.

The actual delay time is computed as follows:

1. getDelay(...) divided by the delay interval, and rounded to its ceiling.
2. this is then multiplied by the delay interval again, and added to the current time.

For example, if you had a delay interval of 2, a current time of 7.0, and you selected a delay time of 4.342, you'd get $\lceil 4.342/2 \rceil \times 2 = 6$, and $6 + 7.0 = 13.0$. By default the delay interval is just 1.

There are three kinds of delays. A *SimpleDelay* delays all resources by the same fixed amount of time, and is implemented internally with a LinkedList. A *Delay* allows resources to have variable delay times relative to one another, and in fact have random delay times. It is implemented with a binary heap, and incurs an O(lg n) insertion and removal cost. A *Bounded Delay* is like a Delay except that the delay times must be integers between 0 and some m exclusive. This is implemented as an array, and is as fast as SimpleDelay.

Below we list only the BoundedDelay methods which are significantly different from SimpleDelay or Delay.

sim.des.BoundedDelay Constructor Methods

public BoundedDelay(SimState state, double delayTime, Resource typical, int maxDelaySteps, int delayInterval)
Creates a BoundedDelay with the given SimState, initial default delay time, typical resource, maximum delay steps, and delay interval.

public BoundedDelay(SimState state, double delayTime, Resource typical, int maxDelaySteps)
Creates a BoundedDelay with the given SimState, initial default delay time, typical resource, and maximum delay steps. The delay interval is 1.

public BoundedDelay(SimState state, Resource typical, int maxDelaySteps, int delayInterval)
Creates a BoundedDelay with the given SimState, typical resource, maximum delay steps, and delay interval. The delay time is 1.

public BoundedDelay(SimState state, Resource typical, int maxDelaySteps)
Creates a BoundedDelay with the given SimState, typical resource, and maximum delay steps. The delay time and delay interval are both 1.

---

sim.des.BoundedDelay Methods

public int getDelayInterval()
Returns the delay interval.

public void setDelayInterval(int val)
Sets the delay interval. This value must be ≥ 1 or an exception will be thrown.

public void setDelayTime(double delayTime)
Sets the delay time, which must be > 0 and $\leq$ the maximum delay steps.

protected double getDelay(Provider provider, Resource amount)
Computes the delay as usual, except restricts it to be > 0 and $\leq$ the maximum delay steps.

public void autoScheduleAt(double time)
A convenience method which calls setAutoSchedules(true), then schedules the BoundedDelay on the Schedule using the current rescheduleOrdering. The BoundedDelay is initially scheduled at the given time.

4.8 Composer

A Middleman. Upon receiving a Resource, it adds it to a collection of Resources. These Resources can be of different types. When it has all the necessary amounts of Resources for each type, it gathers them together and produces a Composed Entity holding these Resources, then offers the Entity to downstream receivers. If they do not accept, then the Entity is discarded. The Composer has both minimum and maximum (capacity) amounts for each of the Resources needed to compose into the Entity.

sim.des.Composer Constructor Methods

public Composer(SimState state, Entity typicalProvided, Resource[] minimums, double[] maximums)
Builds a composer which outputs composite entities of the given type. Each entity consists of resources with the given minimums and maximums. If a resource is an entity, and its maximum (which must be an integer) is larger than 1, this indicates that you want more than one of this entity present in the composition.

Throws a RuntimeException if there is a duplicate among the provided resources, or if a minimum is $>$ its maximum, or if a resource is an Entity but its maximum is not an integer.

---

sim.des.Composer Methods

public void setOffersImmediately(boolean val)
Sets whether the Composer offers Entities immediately in zero time upon accepting the last resource necessary to build them, as opposed to only when it is stepped. The default is TRUE.

public boolean getOffersImmediately()
Returns whether the Composer offers Entities immediately in zero time upon accepting the last resource necessary to build them, as opposed to only when it is stepped. The default is TRUE.

public boolean accept(Provider provider, Resource amount, double atLeast, double atMost)
Accepts offered resources as usual. If the resources accepted so far can be composed, offers the composed Resource immediately if setOffersImmediately() is TRUE. Otherwise offers them when step() is called.

public Resource getTypicalReceived()
Returns NULL because various resource types are received. To get the full list of legal received resource types, call getPermittedReceived()

public Resource[] getPermittedReceived()
Returns the full list of legal received resource types.

public void step(SimState state)
If stepped, offers the composed entity if it is ready.

4.9 Decomposer

A Middleman. Upon receiving a Composed Entity, it decomposes it and offers each of its Resources to various downstream receivers, one registered for each Resource type. If they do not accept, then that Resource is discarded. **By default, this class's step() method does nothing, so there's no need to schedule it.**

sim.des.Decomposer Constructor Methods

public Decomposer(SimState state, Entity typicalReceived)
Builds a Decomposer with the given state and typical received Entity type.

sim.des.Decomposer Methods

public boolean addReceiver(Receiver receiver)
Registers a receiver. Only one receiver may be registered for a given type. If a receiver cannot be registered because another has already been registered for that type, FALSE is returned.

public boolean accept(Provider provider, Resource amount, double atLeast, double atMost)
Accepts offered composed entities as usual, but then immediately breaks them up into Resources and offers them to each registered Receiver as appropriate.

public Resource getTypicalReceived()
Returns the typical received Entity.

public Resource getTypicalProvided()
Returns NULL because various resource types are provided.

public void step(SimState state)
If stepped, offers the composed entity if it is ready.

protected void processEntityInfoFor(Entity entity)
This is called when the Decomposer breaks apart a composite entity, immediately before extracting the elements in its Storage and offering them to downstream Receivers. It's meant to give you an opportunity to process the Entity's Info object if you need to. By default this does nothing.

4.10 If

A Middleman which receives and incoming offer, then chooses only one of several possible Receivers to hand the offer to. This choice is specified in a method defined by the modeler.

An If must have an offer policy (see Provider) of OFFER_POLICY_SELECT. It sets this automatically.

sim.des.If Constructor Methods

public If(SimState state, Entity typicalReceived)
Builds a If with the given state and typical received Entity type.

sim.des.If Methods

public void setOfferPolicy(int offerPolicy)
Throws an exception. You may not change the offer policy.

public boolean provide(Receiver receiver)
Always returns false and does nothing. If is push-only.

public abstract Receiver selectReceiver(ArrayList¡Receiver¿ receivers, Resource resource)
You must implement this. See Provider.

---

5 Filters

Filters are just Processes which accept offers and immediately turn around and offer them to a single downstream Receiver. Filters are both Providers and Receivers, and never respond to requests to make offers.

Important Note Filters may go away soon and be merged into Middlemen. I'm not sure yet.

5.1 Filter

The abstract superclass of Filters, which simply gathers together variables and methods common to them.

5.2 Transformer

Upon receiving a Resource, Transformer **transforms** the Resource into a different one according to a certain rule, then immediately offers it to downstream receivers. If they do not accept, then the Resource is discarded. **By default, this class's step() method does nothing, so there's no need to schedule it.**

5.3 Lock

Upon receiving a Resource, Lock attempts to **allocate** some amount of *a different* Resource from a **Pool**. If successful, it then offers the original Resource to downstream receivers. If they do not accept, then that Resource is discarded. **By default, this class's step() method does nothing, so there's no need to schedule it.**

5.4 Unlock

A subclass of Lock, but only for convenience's sake. Upon receiving a Resource, it **provides** some amount of *a different* Resource to a **Pool**. Regardless of whether the Pool accepts this generosity, Unlock then offers the original Resource to downstream receivers. If they do not accept, then that Resource is discarded. **By default, this class's step() method does nothing, so there's no need to schedule it.**

5.5 Probe

An object which gathers statistics on the resources which pass through it. A Probe can also be used in conjunction with a **Lead**.

5.6 Lead

An object detached from Probe with can be used in combination with it to determine the amount of resources in a DES subgraph between them.

6 Process Utilities

6.1 Macro

An object which can store a **subgraph** of the DES graph. This subgraph consists of some **Receivers**, some **Providers**, and some **in-between Processes**. When stepped, the Macro steps all of its stored objects in order. You can access the Receivers and Providers to make offers or register stuff as you see fit.

6.2 Service

A simple example of a common Macro consisting of a Lock, then a SimpleDelay, then an Unlock.

6.3 Multi

An object which contains multiple Providers and multiple Receivers, all with potentially different typical Resource types, and so which can work with a variety of Resources simultaneously. A Multi can also perform transactions via custom-made Middlemen called **Brokers**.