

# Tutorial: Java Distributed Refreshable Objects 3

Vladimir Ovchinnikov<sup>1</sup>  
© Fusionsoft, 2007

Distributed Refreshable Objects (DRO) 3 is a software library for Java developers enhancing inter-object communication in local as well as distributed environments, which can be considered as more effective and flexible alternative to CORBA and other distributed-object infrastructures. The corner stone of the library is the concept of refreshable object. A refreshable object provides change tracking for object properties, proper as well as calculated from other objects' properties, and it provides caching resource-intensive calculated properties without stale data. All this is done transparently for programmers; however there is an interface for programmers to control the process.

This functionality is supported in distributed environment with no any messaging server, peer-to-peer connections are organized instead, which gives very high scalability. While DRO 2.0 supported it through CORBA, DRO 3 has additional remote call packaging functionality by means of its own distributed object intercommunication layer. The layer provides much more effective network data interchange since several remote calls are collected and sent as one package; a bunch of iterator elements is transmitted at a time. The layer operates transparently for programmers; and an interface for programmers to control remote call packaging also exists.

All distributed objects in DRO 3 are cached on client side, so repeated use of object properties does not result in network exchange. However, when a server object is modified, all its cached client copies become updated; no stale data are possible.

In sum, DRO 3 gives means to create distributed as well as local applications with high flexibility and efficiency. No code structure modification is necessary to add resource-intensive calculated property caching, and to migrate an application into distributed environment efficiently.

This library allows for class providers to avoid programming of object property change notification, including within distributed environment; it is a ready-to-use mechanism being tuned up with [annotations](#) on classes and methods. Class users are able to track changes of any object properties by means of callbacks. Proper and calculated properties can be cached by adding the annotation `@Cached` on its getters, which results in flexible optimization of performance with no interference with the main code. The proposed architectural pattern "refreshable caching" is an alternative to [the pattern "publish/subscribe"](#) and an addition to [the architectural pattern MVC \(Model-View-Controller\)](#). Features of the approach let reduce programmers' efforts in programming complex interaction of objects noticeably.

---

<sup>1</sup> Please, report all discovered errors and inaccuracies, suggestions and requests to [info@fusionsoft-online.com](mailto:info@fusionsoft-online.com)

# CONTENTS

Problem to Solve .....	3
Running Example .....	3
Example with the Pattern Publish/Subscribe.....	4
Example with Refreshable Objects.....	7
Local Refreshable Objects: Development and Use .....	9
Getters.....	10
Getter Declaration.....	10
Logic of Getters .....	11
Setters .....	12
Setter Declaration .....	12
Logic of Setters.....	13
Restrictions of Clear-Methods.....	14
Methods for Change Event Interception.....	15
Freeing and Removal Methods.....	15
Tuning Up the Refresh Mechanism.....	16
Exclusion of Methods from the Refresh Mechanism .....	16
Caching of Getters.....	17
Switching off of Property Subscription.....	17
Setters Unchanging Properties .....	18
Refreshable Object Usage .....	19
Efficiency Questions .....	19
Distributed Refreshable Objects: Development and Use .....	20
Declaring Distributed Classes .....	20
Declaration and Use of Remote Interfaces .....	20
Declaration of Distributed Object Identifiers .....	21
Declaration of Distributed Classes .....	21
Caching Remote Objects on Client Side .....	22
Distributed Object's Life Cycle.....	22
Request Brokers.....	22
Interaction with Distributed Objects.....	22
Distributed Object Use .....	22
Invocation Packaging .....	23
Optimization of Invocation Packaging.....	24
Postponed Invocations.....	24
Cast Postponing .....	24
Invocation Reordering .....	24
Local Object Wrapping .....	25
Null Comparison Avoidance .....	25
One-way invocations and back request packages.....	25
Iterator Packaging.....	25
Architectural Pattern of Refreshable Caching.....	26
Starting up Request Brokers.....	28

## Problem to Solve

Consider an object some properties of which are calculated from other objects' properties. We don't want to recalculate such properties each time since it will consume resources, so we are going to cache them. But after caching, we reveal that the property becomes stale in time after some changes to underlying properties made. We should clear the cache every time the cached property is going to change as a result of changing of underlying properties this one is calculated from. So, there is a task to track changes of properties, including calculated ones, for instance, to recalculate other (dependent) object properties or to refresh user interface. Of course, the task can be solved by means of [the pattern "publish/subscribe"](#), but there is another solution more elegant and less tedious.

The task becomes more complex in distributed environment since a property can be calculated from properties of objects residing in other hosts (JVMs). We don't want to have additional remote calls to support our behavior, so we should embed the feature into distributed object infrastructure.

Also we wish to reduce network traffic to make distributed applications work more efficient since they are known to work very slowly in case of intense inter-object interaction. We can do two things to conquer the problem. First, it's reasonable to pack several remote calls to the same host into one package to reduce the count of network interchange cycles. Second, we can cache properties of remote objects on client side, using property change notification mechanism to know when to clear the cache. Being done transparently for programmers, all this gives a powerful tool to create distributed systems. The packaging-featured solutions are 3-100 times faster than without the feature. However it requires that we use our own distributed object infrastructure implementation, not CORBA-compatible. The possibility to use CORBA (or other solution) at the same time as our solution still remains, but without the packaging feature.

Then, we start from local refreshable objects, and then go to distributed ones.

## Running Example

Consider the class `NumberSum` as a running example. The class sums a set of integers up. First of all, let us describe the classes `Number` and `NumberList` to complete the picture<sup>2</sup>. The class `NumberSum` has the property `Sum` which is calculated with `getSum` on basis of the number list passed to its constructor. The question we answering is how to implement notification of `NumberSum` users about changes of the property `Sum` in the most practically feasible way.

```
public class Number {
    private int number;
    private final NumberList numberList;

    Number (int number, NumberList numberList){
        this.number = number;
        this.numberList = numberList;
    }

    public int getNumber(){return number;}

    public void setNumber(int number){this.number = number;}

    public void remove(){
        numberList.removeNumber(this);
    }
}
```

---

<sup>2</sup> The example has no practical sense; it is created to illustrate the maximum of product possibilities in one obvious and simple example.

```

public class NumberList {
    private final List<Number> numbers = new ArrayList<Number>();

    public Iterator<Number> getNumbers(){ return numbers.iterator(); }

    public Number addNumber(int number){
        final Number result = new Number (number, this);
        numbers.add(result);
        return result;
    }

    public void removeNumber(Number number){
        numbers.remove(number);
    }
}

public class NumberSum {
    private final NumberList numberList;

    NumberSum(NumberList numberList){
        this.numberList = numberList;
    }

    public int getSum(){
        int sum = 0;
        Iterator<Number> numbers = numberList.getNumbers();
        while (numbers.hasNext())
            sum += numbers.next().getNumber();
        return sum;
    }
}

```

Further the example will be used to illustrate adding of some property change notification functionality.

### ***Example with the Pattern Publish/Subscribe***

As we mentioned above, [the pattern "publish/subscribe"](#) let solve the task of object property change notification with no usage of this library, but in labor-intensive way. Let us show it.

There can be different implementations of the notification functionality for our example with [the pattern "publish/subscribe"](#); here is one of them:

- declare listener interfaces, one for each of the properties: `Number.Number`, `NumberList.Numbers`, `NumberSum.Sum` (see the interfaces `NumberChangeListener`, `NumbersChangeListener`, `SumChangeListener` below);
- implement subscription and notification mechanism (see the methods `subscribe`, `unsubscribe`, and private methods `numberChanged`, `numbersChanged`, `sumChanged` below);
- implement listeners, including cascading notification according to dependencies (see the implementation of the interfaces `NumberChangeListener`, `NumbersChangeListener`, `SumChangeListener` below);
- modify constructors so that initial subscription would be made (see the constructor of the class `NumberSum`);
- modify setters so that subscription on all necessary events would be supported, including addition and removal of subscription when necessary (see the methods `add*`, `remove*`, and `set*` in all classes below).

```

public interface NumberChangeListener{
    void numberChanged();
}

public interface NumbersChangeListener{
    void numbersChanged();
}

public interface SumChangeListener{
    void sumChanged();
}

public class Number {
    private int number;
    private final NumberList numberList;
    private Set<NumberChangeListener> subscription =
        new HashSet<NumberChangeListener>();

    Number (int number, NumberList numberList){
        this.number = number;
        this.numberList = numberList;
    }

    public int getNumber(){return number;}

    public void setNumber(int number)
    {
        this.number = number;
        numberChanged();
    }

    public void remove(){
        numberList.removeNumber(this);
        numberChanged();
    }

    public void subscribe(NumberChangeListener
        numberChangeListener){
        subscription.add(numberChangeListener);
    }

    public void unsubscribe(NumberChangeListener
        numberChangeListener){
        subscription.remove(numberChangeListener);
    }

    private void numberChanged() {
        Iterator<NumberChangeListener> iterator =
            subscription.iterator();
        while (iterator.hasNext())
            iterator.next().numberChanged();
    }
}

public class NumberList implements NumberChangeListener {
    private final List<Number> numbers = new ArrayList<Number>();
    private Set<NumbersChangeListener> subscription =
        new HashSet<NumbersChangeListener>();

    public Iterator<Number> getNumbers(){ return numbers.iterator(); }

    public Number addNumber(int number){
        final Number result = new Number (number, this);
        numbers.add(result);
    }
}

```

```

        result.subscribe(this);
        numbersChanged();
        return result;
    }

    public void removeNumber(Number number){
        numbers.remove(number);
        number.unsubscribe(this);
        numbersChanged(); /*It can be omitted since
        it is called indirectly from numbers.remove(number)*/
    }

    public void subscribe(NumbersChangeListener
        numbersChangeListener){
        subscription.add(numbersChangeListener);
    }

    public void unsubscribe(NumbersChangeListener
        numbersChangeListener){
        subscription.remove(numbersChangeListener);
    }

    private void numbersChanged() {
        Iterator<NumbersChangeListener> iterator =
            subscription.iterator();
        while (iterator.hasNext())
            iterator.next().numbersChanged();
    }

    public void numberChanged() {
        numbersChanged();
    }
}

public class NumberSum implements NumbersChangeListener{
    private final NumberList numberList;
    private Set<SumChangeListener> subscription =
        new HashSet<SumChangeListener>();

    NumberSum(NumberList numberList){
        this.numberList = numberList;
        numberList.subscribe(this);
    }

    /*This is for explicit unsubscription when necessary.*/
    public void remove(){
        numberList.unsubscribe(this);
    }

    public int getSum(){
        int sum = 0;
        Iterator<Number> numbers = numberList.getNumbers();
        while (numbers.hasNext())
            sum += numbers.next().getNumber();
        return sum;
    }

    public void subscribe(SumChangeListener sumChangeListener){
        subscription.add(sumChangeListener);
    }

    public void unsubscribe(SumChangeListener sumChangeListener){
        subscription.remove(sumChangeListener);
    }
}

```

```

    private void sumChanged() {
        Iterator<SumChangeListener> iterator =
            subscription.iterator();
        while (iterator.hasNext())
            iterator.next().sumChanged();
    }

    public void numbersChanged() {
        sumChanged();
    }
}

```

Analyzing the result, we can conclude the following:

- we had to add some auxiliary code having no direct relation with application domain logic, which results in cluttering the main code to a certain extent;
- the amount of the added auxiliary code is practically equal to the amount of the main code (see in comparison with the first listing);
- the auxiliary code is made according to one scenario, but it is not so trivial as we wanted to; as a result, amount of work has increased noticeably;
- the similar coding is necessary for every property which requires change notification support;
- the auxiliary code could contain some mistakes hard to reveal.

An alternative to [the pattern "publish/subscribe"](#), having no the described restrictions, is this library of refreshable objects.

### ***Example with Refreshable Objects***

In order to place objects under control of the library and to implement object property change tracking, a class provider should do the following:

- mark classes with the annotation `@Refreshable`;
- in case of getters and setters have nonstandard names, mark them with the annotations `@Gets` and `@Sets` accordingly; standard names are `get<PropertyName>`, `set<PropertyName>`, `is<PropertyName>`, `has<ProprtyName>`;
- use the static method `WrapManager.newInstance` to create new instances of the classes managed by the mechanism.

Using the library of refreshable objects, our task can be solved as follows:

```

import com.fusionsoft.wrapper.*;

@Refreshable
public class Number {
    private int number;
    private final NumberList numberList;

    Number (int number, NumberList numberList){
        this.number = number;
        this.numberList = numberList;
    }

    public int getNumber(){return number;}

    public void setNumber(int number){this.number = number;}

    public void remove(){
        numberList.removeNumber(this);
    }
}

```

```

        public static Number newNumber (int number,
            NumberList numberList){
            return (Number)WrapManager.newInstance(Number.class,
                new Class[]{int.class, NumberList.class},
                new Object[]{new Integer(number), numberList});
        }
    }

    @Refreshable
    public class NumberList {
        private final List<Number> numbers = new ArrayList<Number>();

        @Gets("Numbers")
        public Iterator<Number> getNumbers(){ return numbers.iterator(); }

        @Sets("Numbers")
        public Number addNumber(int number){
            final Number result = Number.newNumber(number, this);
            numbers.add(result);
            return result;
        }

        @Sets("Numbers")
        public void removeNumber(Number number){
            numbers.remove(number);
        }

        public static NumberList newNumberList(){
            return (NumberList)WrapManager
                .newInstance(NumberList.class);
        }
    }

    @Refreshable
    public class NumberSum {
        private final NumberList numberList;

        NumberSum(NumberList numberList){
            this.numberList = numberList;
        }

        public int getSum(){
            int sum = 0;
            Iterator<Number> numbers = numberList.getNumbers();
            while (numbers.hasNext())
                sum += numbers.next().getNumber();
            return sum;
        }

        public static NumberSum newNumberSum(NumberList numberList){
            return (NumberSum)WrapManager.newInstance(NumberSum.class,
                new Class[]{NumberList.class},
                new Object[]{numberList});
        }
    }
}

```

A class user can intercept property changes by defining a new derived class and declaring a method `clear<PropertyName>` for every property to track (the annotation `@Clears` can be also used). In the following example, the method `clearSum` will be called every time when the property `Sum` is changed (in other words, when the result of the method `getSum` is changed):

```

public class NumberSumCatcher extends NumberSum {

```

```

    public NumberSumCatcher(NumberList numberList) {
        super(numberList);
    }

    public void clearSum(){
        /*DO SOMETHING HERE*/
    }

    public static NumberSum newNumberSum(NumberList numberList){
        return (NumberSum)WrapManager
            .newInstance(NumberSumCatcher.class,
                new Class[]{NumberList.class},
                new Object[]{numberList});
    }
}

```

Since clear-methods have several serious restrictions described in the section "Restrictions of Clear-Methods", there is another way to get property change notifications having no such restriction. The annotation `@OnChange` described in the section "Restrictions of Clear-Methods" is used for it. In order to print a new value of the property `Sum` when it changes in the example above, one should define the following method in the class `NumberSumCatcher`:

```

    @OnChange("Sum")
    public void onSumChange(){
        System.out.println("The sum is: " + getSum());
    }

```

Comparing the result with the initial code, we conclude the following:

- the applied code remains unchanged, clear, and transparent; the auxiliary code is replaced by annotations completely;
- the amount of work to add notification functionality is minimal, it is reduced to specifying annotations;
- there can be no coding mistakes for the notification mechanism itself; the mechanism is hidden from the application programmer, well-tested in multiple reuse.

Thus, the solution based on this library is much simpler than the solution based on [the pattern "publish/subscribe"](#). It is more reliable and can be developed in the shortest possible time.

## Local Refreshable Objects: Development and Use

In order to make a class refreshable, one should give the annotation `@Refreshable` to it:

```

@Refreshable
public class NumberSum {...

```

For the mechanism of refreshable objects to work correctly, all class methods should be categorized on getters, setters, clear-methods, free-methods, remove-methods, on-change-methods, and other methods not participating in the mechanism.

The mechanism takes into account only public non-static methods since its implementation requires the methods be called from a separate auxiliary object. Presence of any annotations on non-public or static methods is ignored completely.

## Getters

It is well known that getters are methods returning some object property. Getters can be named in the standard way (`get<PropertyName>`, `is<PropertyName>`, `has<Property>`) as well as in the non-standard way (for instance, `position`, `iterator` и т.п.).

### Getter Declaration

For the refresh mechanism to take into account all standard-named public getters, the class annotation `@Refreshable` should have the parameter `usePropertyDefaultNaming=true`:

```
@Refreshable(usePropertyDefaultNaming=true)
public class NumberSum {...
```

If the parameter is `false`, the refresh mechanism will take into account only those getters which declared explicitly with the annotation `@Gets`. Other getters will not participate in the mechanism, and tracking for the appropriate properties will not be possible. The default value for the parameter is `true`.

The annotation `@Gets` is used to designate getters explicitly and requires one parameter which is the name of the property being requested. The annotation prevails over method naming: first of all, a method annotation will be taken into account, and only if it is absent, the name of the method is interpreted. For instance, the annotation of the method and not its name is used in the following example:

```
@Gets("Numbers")
public Iterator<Number> getNumbers()...
```

A name of a property being requested is assigned reasoning from property uniqueness: two properties are different only if they have different names. But if the same name of properties is used in different classes, the properties will be obviously different.

The property name for getters named in the standard way, and having no the annotation `@Gets`, is taken as the name of the method without the standard prefix. For example, in the case of the method `getSum` the property has the name `Sum`:

```
public int getSum(){...
```

One property can be requested with several getters. For instance, the same property is returned by both getters in the following case. One method returns all numbers (through an iterator), and another method returns the specific number located by the given index.

```
@Gets("Numbers")
public Iterator<Number> getNumbers()...

@Gets("Numbers")
public Number getNumber(int index)...
```

If we omit the annotation `@Gets` on the second method, the method will be associated to the property `Number`, not the property `Numbers` (according to the standard method naming rule). In this case, logic of work for the refreshable object mechanism will be somewhat different.

It is a class provider who is responsible for dividing a class on properties, to his discretion. In extreme case, all getters of a class can return the same property reflecting an object state in the whole, if a developer supposes that details are not necessary.

It is important to note that change notification concerns a property itself, and not a getter which returns the property. Therefore, if several properties are combined into one, a change of any of them results in the change of the combined property as a whole.

## Logic of Getters

As a rule, when a complex getter is calculated, some other getters, possibly of other objects, are called. The refresh mechanism uses the actual method call sequence to subscribe on changes of properties corresponding to the getters being called. In our running example, the subscription can be illustrated as follows:

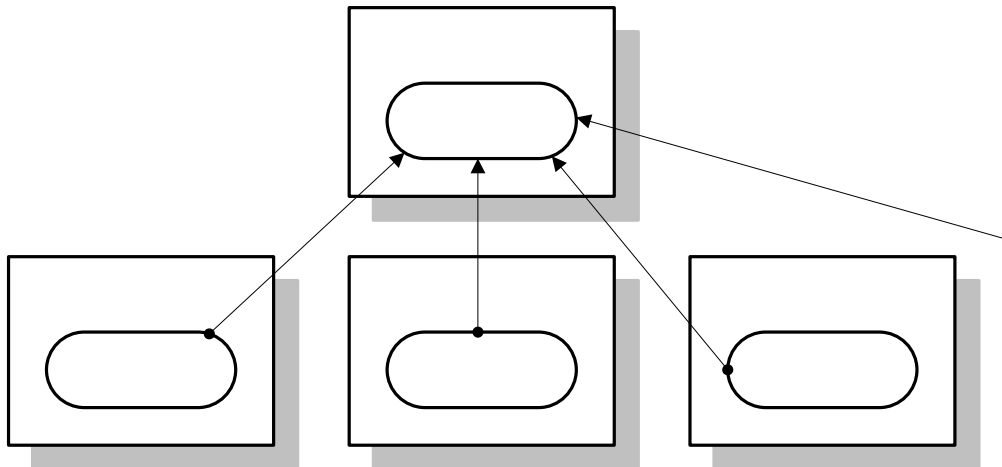


Figure 1 Property Subscription Illustration

The arrows here show the direction of property change event propagation. The direction is opposite to the subscription direction. The property `Sum` of an instance of the class `NumberSum` subscribes to the property `Numbers` of an instance of the class `NumberList`, since the body of the getter `NumberSum.getSum` contains the invocation for the getter `NumberList.getNumbers`:

```
public int getSum(){
    int sum = 0;
    Iterator<Number> numbers = numberList.getNumbers();
    while (numbers.hasNext())
        sum += numbers.next().getNumber();
    return sum;
}
```

Subscription on properties here is made dynamically in the process of method invocations. Such approach let track changes in object properties with no resorting to additional programming by class developers. For instance, in the case of the method `getSum`, no additional efforts required even for annotating.

It is important to note that subscription among properties is made on the level of specific instances. For example, every instance of the class `Number` has the property `Number`, which differs from other properties named `Number` existing in other instances of the class (see the figure above). Therefore, when a getter is called for different objects, the call sequence is processed in the standard way as described above; so, subscription of some properties to others is made, taking no notice to the fact that they have the same names.

## Setters

Setters are methods modifying some object property. Setters can be named in the standard way (`set<PropertyName>`) as well as in the non-standard way (for instance, `addElement`, `removeElement`, etc.).

## Setter Declaration

For the refresh mechanism to take into account all standard-named public setters, the class annotation `@Refreshable` should have the parameter `usePropertyDefaultNaming=true`:

```
@Refreshable(usePropertyDefaultNaming=true)
public class NumberSum {...
```

If the parameter is `false`, the refresh mechanism will take into account only those setters which are declared explicitly with the annotation `@Sets`. Other setters will not participate in the mechanism, and tracking for the appropriate properties will not be possible. The default value for the parameter is `true`.

The annotation `@Sets` is used to designate setters explicitly and requires one parameter which is the name of the property being changed with it, for example:

```
@Sets("Numbers")
public Number addNumber(int number) {
    final Number result = Number.newNumber(number, this);
    numbers.add(result);
    return result;
}
```

The annotation `@Sets` prevails over method naming: first of all, a method annotation is taken into account, and only if it is absent, the name of the method is interpreted. The property name for setters named in the standard way, and having no the annotation `@Sets`, is taken as the name of the method without the standard prefix. For instance, in the case of the method `setNumber`, the property has the name `Number`:

```
public void setNumber(int number) {this.number = number;}
```

The property name indicated in the annotation `@Sets` should be equal to the name of the property indicated on the getters returning the property. In the following example, the method `addNumber` changes a property returned by the method `getNumbers`, and so annotations on both the methods should refer to the same property name `Numbers`.

```
public Iterator<Number> getNumbers() { return numbers.iterator(); }

@Sets("Numbers")
public Number addNumber(int number) {
    final Number result = Number.newNumber(number, this);
    numbers.add(result);
    return result;
}
```

If it is not so, the refreshable object mechanism will not be able to understand when the property `Numbers` is changed, and as a result, it will not be able to notify subscribers that the property has changed.

One property can be changed by means of several setters. For instance, in the following case, both methods change the same property, but one of them adds an element to the property, and the other removes.

```
@Sets("Numbers")
public Number addNumber(int number){
    final Number result = Number.newNumber(number, this);
    numbers.add(result);
    return result;
}

@Sets("Numbers")
public void removeNumber(Number number){
    numbers.remove(number);
}
```

## Logic of Setters

Setters are known to change object properties. Therefore, when a setter ends working, the property the setter belongs to will be changed and the notification will be propagated. Notification is made by the refresh mechanism and it does not require any additional efforts from a developer's side.

Notification process is executed in cascade way according to subscription existing among properties on the moment of notification. Consider the example on the page 11. When the property `Number` is changed for some instance of the class `Number`, the change notification will be generated concerning the property `Sum` of an instance of the class `NumberSum`, which uses the changed property. The figure uses arrows to show direction where notification is propagated to.

Properties subscribed on some changing property directly or indirectly form a change tree (see the figure on the page 11). When a property being in the root of the tree is changed, for instance the property `Number`, all properties being in this tree will be changed inevitably (`Sum` in our case).

During change notification process, a clear-method is called for every property of the change tree. A clear-method is a method serving for processing a fact of change for some property. Clear-methods can be described with the annotation `@Clears(<PropertyName>)` as well as with standard names `clear<PropertyName>` (the annotation can be omitted in this case). The annotation has one parameter that is a property name. For example, a clear-method for the property `Sum` can be declared with one of the ways below:

```
public void clearSum(){...

@Clears("Sum")
public void clearSum(){...
```

If no clear-method is specified for some property, the notification process is continued as usual with no method called. When specified for changed properties, clear-methods are invoked even if one of earlier called clear-methods raised an unhandled exception.

Call order is not defined for clear-methods. Actually, the order is the traverse order for the change tree, but how it will be done exactly is not known a priori.

Property names specified for clear-methods should coincide with names specified for appropriate getters and setters. For instance, the example below shows a getter and a clear-method, both related to the property `Sum`, since they are named using the default naming rule and they have `Sum` in their names after the standard prefixes.

```
public int getSum(){...
```

```
public void clearSum(){...}
```

It is the method `clearSum` which will be called as a result of changes of the property `sum`. Since our example provides only indirect changes to the property, by changing number sets or number values in the list, then there should be no setters for the property.

Besides invocations of clear-methods, a change notification process for some property resets subscription of the property to other properties. It is necessary since if some property changed, logic of getters using this property can change too. More precisely, the logic remains unchanged, but the execution sequence can change since some of properties used by getters can have taken other values. And as a result, it could require the property subscribe to another set of properties. The subscription will get renewed when a getter for the property is executed first time after reset. Both reset and renewal of subscription are made in transparent way for application programmers.

The change notification process can be executed by calling a clear-method for a property changed. In this case, the initiated change tree notification process is absolutely the same as in the case a setter for this property ends working. This feature can be used in getters as well as in setters without restrictions. But if one used this feature, it likely tells that there are problems in system architecture. Their resolution let significantly simplify tasks to solve, and not only in this aspect.

Logically, there can be only clear-method for one property, but the refresh mechanism does not impose restrictions on declaration of several clear-methods for one property. In this case, during the notification process, all the clear-methods will be called in arbitrary order. Here is an example of such declaration:

```
public int getSum(){...  
  
public void clearSum(){...  
  
@Clears("Sum")  
public void clearSum2(){...}
```

## Restrictions of Clear-Methods

Clear-methods represent the mechanism of property change notification. It is not assumed that clear-methods would have some "heavy" functionality, these methods are exclusively for:

- clearance of a cache, if getters of the appropriate property use caching, for instance, in the following manner:

```
private Integer area=null;  
  
public int getArea(){  
    if (area == null)  
        area = new Integer(getAreaCalculated());  
    return area;  
}  
  
private int getAreaCalculated(){  
    return width*height;  
}  
  
public void clearArea (){  
    area = null;  
}
```

- marking of some flag for subsequent processing (for example, invalidation of a user interface part for subsequent redrawing).

There can be no cached getter called in the scope of clear-methods, since there is no guarantee that such getter returns actual state of the property requested. The reason of it is that no order is defined for clear-method execution, and as a result, there is no guarantee that the cache is already reset by the time the getter is called.

Since caching is an optimization mechanism and it can be introduced for any getter in time, when the library used, it is not recommended to call any getter from clear-methods at all. This prevents errors connected with the dirty property state completely.

Setters being under control of the refresh mechanism also should not be called from clear-methods since it can result in endless recursion: when a setter is called, all clear-methods of the current change tree are executed immediately after, and if the tree intersects the source change tree, results could be unpredictable.

The "lightness" requirement for clear-methods, absence of calling getters and setters from them, is also imposed by execution atomicity and indivisibility for all clear-methods of a change tree. If clear-methods address to property objects, especially if they change them, then the atomicity will be violated.

In spite of the fact that the requirement sounds like a rigid architecture restriction, in fact it makes the code more transparent and extensible, the maximum amount of applied logic shifts to getters. It reduces duplication of logic which often happens in other architectures. In the last caching example all logic of calculation of the property `Area` was concentrated in the getter, and the clear-methods serves for cache clearance. Therefore, we assume that all actions concerning recalculation of object properties are made directly in getters according to [the pattern "lazy initialization"](#), and not in clear-methods.

Change notification, calling a clear-method, does not contain any additional information except the fact that the specific object property has changed. If some additional data is necessary to accompany a change event, there can be created separate object properties providing this data.

## ***Methods for Change Event Interception***

Sometimes there is necessity to process changes with "heavy" functionality by getting and setting object properties. The mechanism provides additional methods to solve the task. The methods should be described with the annotation `@OnChange` and are called immediately after the root clear-method of the current change tree finishes its work. These methods allow execution of any getters and setters with no restriction, including cached getters. For instance, the following code shows such method which is used for printing a sum value on the screen after every change of it.

```
static public class NumberSumCatcher extends NumberSum {
    ...

    @OnChange("Sum")
    public void onSumChange(){
        System.out.println("The sum is: " + getSum());
    }
}
```

## ***Freeing and Removal Methods***

There is a special category of methods: methods freeing or removing objects they belong to. Freeing objects means removing it from RAM with keeping the entity the object represents, for instance, in an external storage. Removing objects means removing it from RAM with the entry the object represents. Let us call the methods as free-methods and remove-methods accordingly.

If such objects are under control of the refresh mechanism, we recommend you mark free-methods with the annotation `@Frees` without parameters, and remove-methods with the

annotation `@Removes` without parameters also. It allows for the refresh mechanism to make some additional work when such methods are called.

When a free-method called, clear-methods for all properties of the object being freed are invoked. Also the object goes into the special state forbidding requests and changes to its properties.

When a remove-method called, the same work is made as for the free-method, and the change notification is generated for the property the object being removed was created earlier. For instance, if a set of objects is created within a getter and all they are returned in the form of collection (see below),

```
public Collection getPropertyNames() { ...
```

then removal of any of the created objects will initiate change notification for the property `PropertyNames`. One should take into account that this behavior takes place only if created objects are under control of the refresh mechanism and if a remove-method is used when removing.

If the class annotation `@Refreshable` has the parameter `useFreeingDefaultNaming=true`, then the method having the name `free` is categorized as a free-method with no necessity to specify the annotation `@Frees` for it. If the class annotation `@Refreshable` has the parameter `useRemovalDefaultNaming=true`, then the method having the name `remove` is categorized as a remove-method with no necessity to specify the annotation `@Removes` for it, for instance:

```
public void remove() { ...
```

Remove-methods get the special role when getters return newly created objects being under control of the refresh mechanism. Removal of such objects, as a rule, is convenient to implement as remove-methods in the classes of returned objects. When such remove-method is called, change notification for the initial property the getter belongs to will be initiated.

Use of remove-methods is obligatory only in the rare cases when removal of an object earlier created in a getter is not accompanied with execution of some methods of the initial object the getter belongs to. The situation can arise only within different factories, though property change tracking is not relevant for factories. The following example shows the method `remove` which can be taken off control of the refresh mechanism since the method calls the setter of the initial object which does all the work:

```
public void remove() {  
    numberList.removeNumber(this);  
}
```

Therefore, free- and remove-methods are auxiliary for class developers and are not obligatory to use. Above all, they allow for the refresh mechanism to clear the whole property cache of the object being freed or removed, and to block further use of the object, with no any efforts from a class developer's side.

## ***Tuning Up the Refresh Mechanism***

### **Exclusion of Methods from the Refresh Mechanism**

A method can be excluded from the refresh mechanism in an explicit way using the annotation `@NoRefresh`. The annotation is required when a method is categorized as

participating in the refresh mechanism by its name, but in fact this participation is not required from the point of view of a class developer, for instance:

```
@NoRefresh
public int getNumber(){...}
```

## Caching of Getters

The section "Restrictions of Clear-Methods" contains a caching example where caching is implemented by a class developer himself. The refreshable object library has a solution for the caching task which does not require programming; it is the annotation `@Cached`. The annotation is used on getters to cache. For instance, instead of the code in the section "Restrictions of Clear-Methods", a developer can specify the annotation `@Cached` for the method `getArea`.

```
@Cached
public int getArea(){
    return width*height;
}
```

The cache is cleared by the refresh mechanism automatically when receiving change notification for the appropriate property. An application programmer has to neither declare variables, nor implement caching in getters, nor track property changes, nor clear a cache.

The approach lets reduce significantly amount of work necessary to implement caching, increase tuning up flexibility for caching, free application code from auxiliary code connected with caching, avoid mistakes during caching implementation.

If a getter having the annotation `@Cached` should not be cached after its current execution has finished, for instance, if an exception occurs or for other reasons, the static method `RefreshManager.noCache` should be called before exiting the getter. It prevents the getter from caching and the subsequent invocation of the method will certainly result in the getter's code complete execution.

## Switching off of Property Subscription

Sometimes a getter does not fulfill its work and does not return a value requested. This can happen if a program exception arises during its execution. In this case, a class developer may prefer switch property subscription off for the getter. To do this, the static method `RefreshManager.noGet` should be called before the getter finishes its work, which switches off subscription within the current execution of the getter, for example:

```
@Gets("Numbers")
public Iterator<Number> getNumbers() throws NullPointerException
{
    try {
        return numbers.iterator();
    } catch (NullPointerException e) {
        RefreshManager.noGet();
        throw e;
    }
}
```

Subscription switching off results in breaking the property subscription tree on unconnected parts. In our example, a property calling the method `getNumbers` from its getter will not subscribe on the property `Numbers` if the exception `NullPointerException` occurs during execution of the getter `getNumbers`.

There is also possibility to switch subscription off for some getter permanently, using the annotation `@NoUsage` with no parameter. If we switch subscription off for the method `getNumbers` in our example permanently,

```
@Gets("Numbers")
@NoUsage
public Iterator<Number> getNumbers(){...
```

then no property calling the method `getNumbers` from its getter will subscribe on the property `Numbers`. Every execution of the method `getNumbers` will result in breaking subscription tree on parts.

We strongly recommend you do not use the annotation `@NoUsage` for getters returning values as in the example above. This mode is only for initialization methods which do not return values and serve for solving tasks unconnected with the context the methods are executed within. Switching subscription off permanently allows exclude unnecessary subscription for properties calling initialization methods from their getters.

```
@NoUsage
void initialize(){...
```

Specifying the annotation `@NoUsage` for an initialization method, one may not mark the method as a getter, as in the example above.

Absence of such annotation for initialization methods will result in surplus change notification as a result of surplus subscription for properties called such methods directly or indirectly. Though, program logic remains consistent in this case.

There is an important difference between `@NoRefresh` and `@NoUsage`. The difference is that `@NoRefresh` does not switch off property subscription; it only excludes method from controlling by the refresh mechanism. For instance, if we replace the annotation `@NoUsage` with the annotation `@NoRefresh` in the example above, then all properties calling the method `initialize` from its getters will subscribe on those properties getters of which are called in the method `initialize`. If the annotation `@NoUsage` is used, such subscription does not take place.

## Setters Unchanging Properties

Not every method execution gives rise to value change for an appropriate property. The value can remain unchanged because of different reasons: the property can already have the value being set or an exception can arise before a change happens.

When a setter did not change its property actually, a developer can decide not to notify about property change by calling the static method `RefreshManger.noChange` at the end of the setter, for instance:

```
public void setNumber(int number)
{
    if (this.number == number)
        RefreshManger.noChange();
    else
        this.number = number;
}
```

In this case, if a value to set does not differ from the current value, there will be no property change notification for the property `Number` and any property depending on it.

The method `RefreshManger.noChange` is not obligatory to execute every time a property value remains unchanged. If it is not executed, program logic remains untouched. But if it is

executed, it allows you to save some calculation resources which are subject to spend on the property change notification process.

## **Refreshable Object Usage**

Objects built with property change notification support can be used as any other objects without restriction. The only requirement for them is that they should be created using the static method `WrapManager.newInstance`. If the requirement is not met, and objects are created using the operator `new` directly, then the objects stay operable, but no feature of the refresh mechanism can be used: property changes can not be tracked, caching does not work, clear-methods are not called, etc.

In order to catch property change events, a class user should create a new derived class and define clear-methods or on-change-methods for every property to track changes. At the end of created clear-method, the same method of the base class (if any) should be called, as in the following example:

```
public class NumberSumCatcher extends NumberSum {
    ...
    public void clearSum(){
        /*DO SOMETHING HERE*/
        super.clearSum();
    }
}
```

Such derivation is necessary in rare cases, for instance, for setting flags or for invalidation of a user interface part.

If a class user develops its own class with notification support using this mechanism, it is not necessary to take some special actions about using other classes. The refresh mechanism will do it instead of the user.

## **Efficiency Questions**

Some of efficiency questions are considered in the section "Tuning Up the Refresh Mechanism" above. The rest of them are below.

First of all, it is important to note that every object property is accompanied with a separate auxiliary object containing information about subscription among this property and other properties. Therefore, the less quantity of properties is declared in a class, the less calculation resources are consumed when creating and controlling the auxiliary objects.

When a first object of a class is created with the help of the method `WrapManager.newInstance`, a wrapping class is generated for this class. Therefore, creation of first objects takes more time than creation of next objects.

One of advantages of the approach is that change tree notification takes place once for a sequence of changes of one property. Calculation resources are spent for the first change in the sequence only.

Caching mechanism requires more calculation resources at the first execution of a getter, immediately after its cache is cleared. If the cache already has the predetermined result, the getter requires minimum of calculation resources. Therefore, there is no necessity to avoid execution of one getter multiple times since caching of such getters result in minimization of resource consumption. The getters are calculated anew only if it is unavoidable.

## Distributed Refreshable Objects: Development and Use

All said before is true for local refreshable objects as well as for distributed ones. Let's clear the terms. An object is local if it exists in one host only. Being used by another host, the object is to be copied into another different object residing in the other host. So such object is never shared by hosts, a copy can be only created. When a local object is modified, the modification remains within this host only.

Distributed objects are like local ones in every other aspect, but they can be shared by several hosts. When such object is used by another host, a reference to the object will be transmitted from the server host, where the object resides, to the client host, where it is used. The same object can be shared with various client hosts.

Object property dependencies considered above are maintained locally as well as in distributed environment. If a local or distributed object's property is changed, all properties calculated from it will be changed whether they are local or distributed.

Below we describe the main aspects of distributed refreshable objects. The complete description can be found in javadoc.

### *Declaring Distributed Classes*

#### Declaration and Use of Remote Interfaces

Distributed objects can be accessed remotely only through remote interfaces. An interface is remote if it is annotated with `@Remote`, for instance:

```
@Remote  
interface EntityInterface  
...
```

Any class of distributed objects should implement such interfaces, for example:

```
class EntityClass implements EntityInterface  
...
```

When a distributed object is passed as a parameter or returned from a method as a result, it should be passed or returned as an appropriate interface, not as a class itself. It's necessary to be able to redirect calls to remote objects, which can be done in general case for interfaces only.

```
class SomeClass {  
    EntityInterface getEntity(){...}  
  
    void setEntity(EntityInterface entity){...}  
}
```

If a class is used instead of an interface, then its object will try to be passed by value (by copying it into a new object on remote side), and not by reference (by transmitting its identifier only; see identifier description in the next section). Passing by value requires the class itself be serializable. It is useful when an interface has both local and distributed implementation classes. In this case local implementation can be passed both as the class or the interface, but distributed implementation should be always passed as the interface.

## Declaration of Distributed Object Identifiers

DRO 3 introduces identifiers to refer to remote objects. The identifiers can be specific for every distributed class as well as the general identifier `IdentifierSequential` can be used. Specific identifiers are supposed to be used for every class which objects can be loaded on request and unloaded after some disuse time. If we want to use a specific identifier, we should define it first by extending the interface `IdentifierAbstract` and by defining or overriding the following methods in it:

- `findObject()` – finds the object having this identifier (it can be loaded if it was not loaded yet); returns null if no object exists for the identifier (maybe it never existed or was removed).
- `copy()` – makes a new copy of the identifier equivalent to this one;
- `getIdentifierClass()` – returns the class of the identifier (it's necessary to define whether an object is an identifier itself or an entity inheriting the identifier; see identifier inheritance in the next section);
- `equals(Object)` – compares two identifiers for equality (two identifiers should be equal if and only if they are for the same object);
- `hashCode()` – returns hash code for the identified object (it should be the same for all identifiers of the same object);
- `writeObject(ObjectOutputStream)` – serializes the identifier (see the interface `Serializable`);
- `readObject(ObjectInputStream)` – deserializes the identifier (see the interface `Serializable`).

```
class EntityIdentifier extends IdentifierAbstract
```

```
...
```

In case a persistent object has mirror copies on different hosts, the same identifier should be used for the copies on every of these hosts. But then DRO supposes that all the copies represent the same entity with the same state. So if one copy is changed, other copies should become synchronous with changed one. It's a programmer who should ensure this synchronization. However you may decide to use only a master copy and not to bother with mirror copies.

None of the previous is necessary if we use the general `IdentifierSequential`. But objects identified in this manner can not be unloaded on disuse. It's a programmer who should keep such objects in memory all the time while remote requests to them are possible.

## Declaration of Distributed Classes

Now we are ready to complete declaration of our distributed class. Let's use the specific identifier introduced above, not the general one.

```
@Refreshable
```

```
class EntityClass extends EntityIdentifier implements EntityInterface
```

```
...
```

Distributed refreshable objects should be created with wrap manager as local ones. Their classes or interfaces may or may not have the annotation `@Refreshable`. If they do, then such objects are refreshable in local as well as distributed environment. Otherwise, the objects are considered as refreshable only from outside, when they are used by some remote host. Such objects should be built over other refreshable objects for property change notification to work correctly. If it's not true, no property change will be tracked and no cache will be cleared.

## ***Caching Remote Objects on Client Side***

When a distributed object is used by some remote host, all its properties used within the remote host are cached automatically. When a property is requested repeatedly, no remote invocation will be made; its value will be taken from the cache until the property is changed directly or through calculation dependencies. When the property is changed, all its caches will be cleared in every client host having it cached.

All this is done transparently to programmers. No additional arrangements should be made.

## ***Distributed Object's Life Cycle***

Life cycle includes the moments of creation, existence, and removal for an object. In DRO, life cycle is completely managed by programmers. A programmer can delegate the task to some other software, for instance to RDBMS, by implementing specific identifiers for classes. In this case, such identifier should be able to find its object (or report about its absence) whenever it is required for DRO. Specific identifiers can also be used to manage life cycles in custom way without delegation to any software.

The other way is to use the general identifier IdentifierSequential instead of a specific one and implement an object's life cycle anew in full. DRO does not require for a programmer to export an object to use it remotely, therefore DRO does not keep references on such objects. It's a programmer who should keep the references in memory to save the objects from removal in the IdentifierSequential case. Using of specific identifiers does not require such references be kept since objects are loaded on request.

## ***Request Brokers***

A request broker is a software module responsible for processing remote invocations. DRO creates one request broker for every JVM participating in distributed object interaction.

Request brokers are connected one to another in peer-to-peer way. Any pair of request brokers has only one direct connection between them if they communicate each other. The connection is created by initiative of that request broker which was first to send a request to another. As a result, the initiative request broker may not open any ports for incoming connections for DRO to work successfully.

## ***Interaction with Distributed Objects***

### **Distributed Object Use**

Distributed objects can be interacted through their remote interfaces. You can not get an object's implementation class in general case since if the object is remote its implementation is known for its remote host only.

```
{ EntityInterface entity; entity.method(); }  
...
```

Another way to interact is dynamic invocation through the interface RemoteObject which is implemented by all distributed objects. Using this interface, you can call any method on a distributed object:

```
((RemoteObject) entity).invoke(method, args);  
...
```

In both cases, if a distributed object resides in the local host, no remote invocations will be undertaken. However some packaging can take place, see the next section for details.

**IMPORTANT.** When using DRO, you should not check distributed objects for null with “!= null” or “== null”. A special method exists for this: `RequestManager.isNull(Object)`. And you cannot cast some distributed object to a type in any other way than by calling the method `RequestManager.cast(Object, Class<?>)`. The reason is that in both cases the distributed object passed as an argument can be not resolved yet as a result of packaging.

...

To understand is an object local or remote, you can use the method `RequestManager.isLocal(Object)`. An object is local if none of calls to it can produce remote invocation; but other objects which this one invokes can still produce remote invocation. The object which is not local is remote. Not every call to a remote object produces remote invocation: some methods can be cached and so executed locally. There is no way to understand whether a call requires remote invocation or not, except to analyze the appropriate program code. However you can understand whether a method is cached on a specific remote object using the method `RemoteObject.isCachedGetter(Method, Object[])`. The second argument is parameters of the method since caching is done separately for every set of parameters.

...

## Invocation Packaging

DRO applies packaging for distributed object invocations to decrease intension of network interchange between request brokers. Distributed applications become 3-100 times more efficient with packaging. It’s done transparently to programmers for the most part.

Packaging means that some calls may be not executed immediately and may be postponed for later execution. Such postponed invocations are collected into packages; every package contains an unbroken sequence of requests to one request broker, one host. Packages are being collected until a terminal condition arises, which results in execution of all packages collected. There are several types of terminal conditions. Here they are:

- a method returning a type not being a remote interface (for instance `String`, `int`, or a user-defined type) is called on a distributed object;
- `RequestManager.isNull(Object)`, `RequestManager.cast(Object, Class<?>)` or `RequestManager.invokeAll()` are invoked; the last method triggers execution of all collected packages explicitly;
- `RemoteObject.invokeImmediately(Method, Object[])` is called, which also triggers execution of all collected packages explicitly.

...

When packaging is used, exceptions will be raised at the point where packages are executed, not at the point where a method itself was called. Use `RequestManager.invokeAll()` to guarantee that all exceptions were raised up to this point. If a checked exception took place, you should catch `ExceptionUtils.WrapException` which contains details about the exception actually raised.

...

You can switch off packaging feature by calling `RequestManager.setPackaging(false)`.

## Optimization of Invocation Packaging

For better performance, you should aim at less quantity of bigger packages. It gives better efficiency of network interchange. To achieve the most possible efficiency, you should aim at execution of collected packages as late as possible (see execution criterions in the previous section). It means that you should postpone invocations resulting in execution of collected packages to the end of an algorithm.

---

## Postponed Invocations

One way to optimize is to invoke methods in postponed manner explicitly by means of the special dynamic invocation method `RemoteObject.invokePostponed(Method, Object[])`. When this way of invocation is used, no immediate invocation is undertaken if possible. This is only way to postpone invocations for methods returning types not being remote interfaces.

---

## Cast Postponing

This dynamic invocation call returns a distributed object, even if it's actually local. You should use `RequestManager.cast(Object, Class<?>)` method to get access to the object as a required type, not standard cast operations. The cast should be made as late as possible since it results in execution of all collected packages. So, to make better optimization, you can get all necessary result objects as distributed objects using this dynamic invocation method, and then at the end of an algorithm cast them to correct types.

---

## Invocation Reordering

DRO does some additional optimization by reordering call sequences when possible. Reordering is done by moving later requests to early packages to increase the size of packages and so to give more efficiency of network interchange. The reordering is possible only in cases which do not result in program logic modification:

- an unbroken sequence of getter methods, not having other types of methods inside, can be reordered in any way since no modification is done within it, and so all the getters address to the same state of objects;
  - a method can be marked as unordered explicitly with the annotation `@NoOrder(String)`; the parameter of the annotation is a tag: only an unbroken sequence of unordered methods with the same tag is subject to be reordered, usage of other tags breaks the sequence; the annotation is useful when some methods are known as not influencing each other: then all the methods get the annotation with the same tag.
-

## Local Object Wrapping

To include local objects in reordering process you should wrap them first with `RequestManager.wrap(Object)`. The result of the wrapping is a distributed object which should be used to call methods instead of the wrapped local object. If the object implements a remote interface, you can cast the wrapper to the interface and use it, otherwise you have to call methods dynamically over the distributed wrapper through the interface `RemoteObject`.

---

Reordering over local and distributed objects gives great possibility to optimize read-only algorithms, since, for instance, remote getter method invocations can be executed with one package even if they are dispersed over an algorithm. Also wrapping can be useful for postponed invocations over local objects (see postponed invocation description above).

## Null Comparison Avoidance

Another way of packaging optimization is to avoid null comparison for distributed objects when possible. In this case you catch `NullPointerException` instead of comparison with null by means of `RequestManager.isNull(Object)`. At the end of try block you should call `RequestManager.invokeAll()` to guarantee that the exception will be raised if necessary. The try block is supposed to include as much logic as possible.

---

This approach allows for request packages to become bigger and in less quantity. Logic, which is supposed to be done in case of not-null distributed object, goes with that operation requesting the object itself. As a result, in case the object is not null, no additional network interchange cycle is undertaken.

## One-way invocations and back request packages

You can mark a method returning nothing as one-way with the annotation `@OneWay`. In this case, the method is considered as not requiring for a request broker to wait a response. If a request package consists of one-way requests only, the package is considered as one-way too. Request brokers sent such packages do not wait for response packages, and request brokers received such request packages do not send response to them. As a result, we can save a half of network interchange cycle. But you should be careful since there is no way to ascertain whether the package was delivered or not, so the annotation can be used only when such outcome is possible.

---

One-way packages can go to a receiving request broker along with response as back request packages. In this case no additional network interchange cycle is undertaken at all: one-way packages are integrated into the standard request-response cycle between two request brokers. However, this, of course, requires such carrying cycle exist, or else a half of additional cycle will be certainly done.

## Iterator Packaging

Elements of iterators are transmitted in packages in spite of the fact whether packaging is switched on or not. It reduces network exchange intension in 1000 and more times. A first

package is got when hasNext() or next() method called on a remote iterator. Next package will be got when the last package got is completely iterated.

Size of every package is determined by the variable MarshalledIterator.packageSizeInitial, the default value is 1000 elements. You can also manage changes of size in time using packageSizeAddition, packageSizeMultiplier, and packageSizeMaximum variables. New package size is calculated as  $\langle \text{current\_size} \rangle * \langle \text{multiplier} \rangle + \langle \text{addition} \rangle$ , only if it is less than the maximum. Sizes are calculated individually for every iterator: every iterator starts from the default size and changes in time according to the rule.

## Architectural Pattern of Refreshable Caching

The architectural pattern "Refreshable Caching" is at the corner stone of the mechanism of refreshable objects. Let us illustrate the pattern on the following example:

```
public class CachePattern {
    private int width;
    private int height;
    private Integer area=null;

    public CachePattern(int width, int height){
        this.width = width;
        this.height = height;
    }

    public int getWidth(){
        return width;
    }

    public void    setWidth(int width){
        this.width = width;
        clearArea();
    }

    public int getHeight(){
        return height;
    }

    public void    setHeight(int height){
        this.height = height;
        clearArea();
    }

    public int getArea(){
        if (area == null)
            area = new Integer(getAreaCalculated());
        return area;
    }

    private int getAreaCalculated(){
        return width*height;
    }

    public void clearArea (){
        area = null;
    }
}
```

```
public int getArea2(){
    return width*height*2;
}
}
```

The core of the pattern is the following. All properties of an object are divided on the categories:

- unchangeable: they are defined in constructors or initialization methods and never change;
- proper: they can be changed by an object user directly;
- calculated: they can not be changed directly, but they can be changed in principle as a result of changes of those properties they are calculated from.

Proper and calculated properties can be cached in getters using [the pattern "lazy initialization"](#). In this case, when a proper property is changed, its cache and caches of all properties calculated from it are to be cleared, for example:

```
public void setHeight(int height){
    this.height = height;
    clearArea();
}
```

All application logic is divided into view logic and state transition logic in this pattern. View logic represents calculation of some derived object properties on basis of current object states with no changes in the states (all the getters in our example). On the contrary, state transition logic formulates changes to object states.

There is a requirement in this pattern which states that all logic which can be formulated as view logic should be formulated just in this way. Only that logic which can not be formulated in this way is subject to be formulated as state transition logic.

The requirement can be reformulated in terms of methods: all setters should be simple to the maximum, and all their logic which can be extracted to getters should be extracted. In this approach, the quantity of calculated getters is increasing and the maximum of application logic is placed exactly in them.

The approach frees application logic from a significant part of auxiliary code not related to an application domain; the rest auxiliary code becomes more uniform. And usage of the refreshable object library results in reducing the amount of auxiliary code to the minimum.

It is important to note suitability of the approach to efficiency optimization by means of gradual caching of most resource-consuming properties with no essential code structure reorganization, and with no reorganization at all when the refreshable object library is applied.

The approach imposes high requirements and encourages high quality for program architecture being extended flexibly without code reorganization. It is conditioned by high encapsulation and content segmentation of applied logic.

In conjunction with the refreshable object library, the pattern provides automated caching and tracking of changes for complex calculated object properties. An applied programmer does not take into account calculation dependencies among object properties; extension of applied code functionality can be done with no code reorganization and with no auxiliary coding. The programmer can focus on applied logic exclusively.

The pattern in conjunction, with the refreshable object library implementing it, can be considered as an addition to [the architectural pattern MVC \(Model-View-Controller\)](#) by providing more simple notification about object state changes.

## Starting up Request Brokers

Request brokers should have unique names necessary to understand what connection to use for communication. Use the environment variable `com.fusionsoft.BrokerName`, for instance, specify the parameter for JVM as

```
-Dcom.fusionsoft.BrokerName="/Companies/Some Company/Client/Victor's Notebook"
```

Every pair of request brokers can have at most one connection between them. One of the pair plays the role of client, another one is server. Which role each request broker plays is not predetermined: the first request broker to communicate to another one becomes client, and the answering request broker becomes server. After the connection is established, request broker's unique names are used to get it. Any request broker knows which connected request broker is accessible through which connection.

So, client-only request brokers may have no published listeners if it is known that the request brokers are supposed to initiate connection to remote request brokers and no remote request broker is supposed to initiate connection to them. Client-only request brokers may have no accessible IP/PORT listening for remote request brokers to connect.

The situation is different for server request brokers. To specify listening IP/PORT for request brokers being able to play the role of server, use the environment variables `com.fusionsoft.ListenerHost` and `com.fusionsoft.ListenerPort`, for instance, specify the parameter for JVM as

```
-Dcom.fusionsoft.ListenerHost="localhost" -Dcom.fusionsoft.ListenerPort=1060
```

You have to run listener to start listening incoming connections from other request brokers by calling `RequestBroker.startListener()`:

To get access to a remote request broker, use the method `RequestManager.getRequestBroker(String uniqueName, String listenerHost, int listenerPort)`. All the parameters should correspond to parameters of the appropriated server request broker executed earlier.

To print debug information on console, you use the environment parameter `com.fusionsoft.Debug`, for instance, as JVM parameter:

```
-Dcom.fusionsoft.Debug="transport,invocation,clearance"
```

The parameter “transport” shows packages transmitted, “invocation” shows all remote invocations made, and “clearance” shows all cache resettings taken place. You can specify a part of the debug parameters, if necessary.