Design forum

Discussion concerning the design of the NetatchiveSuite. This includes 'micro' code constructs.

Topics 13 Replies 11

	Topic	Author	Re
	Release tests screaming for automation first steps towards a better world may be taken with task NARC23 POC system test http://sbforge.statsbiblioteket.dk/jira/browse/NARC23	Mikis Seth Sørensen 06 Sep	1
	Introduce more powerful test frameworks Currently the NetarchiveSuite project uses an older Junit 3 for automatic testing. We should consider introducing more powerful testing frameworks to improve the value of the automatic tests. We should have a look at the following frameworks: TestNG http ...	Mikis Seth Sørensen 03 Sep	0
	Increase max linelength I've started a crusade for less wrapping of code lines here: APP:Linie length should be more than 80 characters. The readability of the code in my IDE would be greatly increased, eg. Line length = 80 Line length = 120 ...	Mikis Seth Sørensen 26 Aug	0
	Switch to a more modern mock framework currently used mock framework Mockobjects http://sourceforge.net/projects/mockobjects/ latest release is from 2003, which means this is a pretty dead project. Let's try to use a more modern mockup framework like Mockito http://code.google.com/p ...	Mikis Seth Sørensen 24 Aug	0
	Enforcing a nice 3-tier architecture One of the most know architectural patterns is the 3tier separation of applications into: GUI layer: Contains the code for user interactions Application/Server layer: Server side logic Data layer: The data storage The idea here is the reduce the complexity of the complete system ...	Mikis Seth Sørensen 19 Aug	0
	Consistency in equals and hashcode implementations Overriding the equals/hashcode methods defined on the Object http://downloadllnw.oracle.com/javase/6/docs/api/java/lang/Object.html#hashCode() class is very difficult to implement correctly (see Effective Java http://java.sun.com/developer/Books/effectivejava/Chapter3.pdf by Josh ...	Mikis Seth Sørensen 19 Aug	0
	More informative naming, less inline comments Yes, I've just added a new topic NAS:Reviewing test code, where I try to argue that it might be an good idea to lift the quality of the test code by including it in the normal review process. I used the code ...	Mikis Seth Sørensen 11 Aug	3
	CleanupIF vs. Singleton Good idea. As I mentioned earlier I'm planning on creating a sort of precommit review to validate the refactoring path I'm taking, before I put any of the major changes into svn. I'll see if I can't add Nicolas to the Crucible ...	Mikis Seth Sørensen 11 Aug	3
	Rename HARVEST_CONTROLLER_PRIORITY setting I agree somewhat, the HARVESTSERVERJOBTYPE name is much closer to the actual usage of the setting. But, as far as I can see the setting is only used on the server side to identify the channel to fetch HarvestJob messages from. The Harvest server doesn't use the setting ...	Mikis Seth Sørensen 09 Aug	3
	Reviewing test code I think we should seriously consider start review the test code on a continuous based as part of the general reviewing process. The reasons for this are: The quality of the unit test code is not very high. 3/4 of the refactorings I have done so far ...	Mikis Seth Sørensen 18 Aug	0
	Keeping the classes clean part of my ongoing effort to solve NARC11 Stop using the JMS queues for queuing snapshot harvests NARC11@jira I've it the HarvestDefinition class http://kbprodudv001.kb.dk:8060/browse/netarchiveSuite/trunk/src/dk/netarkivet/harvester/datamodel/HarvestDefinition.java?r ...	Mikis Seth Sørensen 18 Aug	0

	The weakness of the CleanupIF course of fixing the Singleton/Cleanup entaglement problem described in the CleanupIF vs. Singleton viewpage.action?pageId=1835079&focusedCommentId=1835090#comment1835090 thread, I removed the singleton aspect of the HarvestScheduler http://kbprodudv001.kb.dk:8060/browse/netarchiveSuite/trunk/src/dk/netarkivet ...	Mikis Seth Sørensen 12 Aug	0
	ChannelID access code way too complicated code has now been committed on a branch (NetarchiveSuite is now 170 lines smaller :) and a review has been created. The review is free for all, so feel free to give your input	Mikis Seth Sørensen 12 Aug	1

Add Topic

ChannelID access code way too complicated

I've created a issue NARC-17 Refactor ChannelId access to simplify the code implementing harverster ChannelID access.

Have a look.

CleanupIF vs. Singleton

In extension of my work on refactoring the HarvestScheduler assignment in the unit test class, which I described in the More informative naming, less inline comments thread, I've struggled with the instantiation, assignment and cleanup of the HarvestSchedule'er. It has finally dawned on me that the the problem is 2 conflicting characteristics of the HarvestScheduler:

1. The class is meant to be a Singleton . This means that only one instance of the class should exist a one time, which the Singleton pattern enforces by hiding the possibility of construction/creating instances from users of the class.
2. The class extends the CleanupIF: This interface implies that it is possible to invoke this operation when closing down the class. This works fine in normal class, where the operation is symmetric to the construction of the class. Eg. an objects life cycle is: create -> living -> cleanup.

The conflict between the 2 aspects of the HarvestScheduler arises because a Singleton pattern hides the object creation control from its API control, where the CleanupIF is logically a functionality associated with the destruction of a object.

The problem can be seen in the follow HarvestScheduler code:

```
public void cleanup() {
  hsmon = null;
  instance = null;
}
```

This code enables a user to create multiple instances of the HarvestScheduler as simple as this:

```
HarvestScheduler h1 = HarvestScheduler.getInstance();
h1.cleanup();
HarvestScheduler h2 = HarvestScheduler.getInstance();
```

This means the that the contract for the Singleton-pattern has been broken. It also means that application behaviour will now depend on when the created HarvestScheduler instances are garbage collect.

What is even worse is that this is used in the unit test to emulate contruction of a new HarvestScheduler instance in each test case. Eg.

```
public class HarvestSchedulerTester extends TestCase {
...
public void tearDown() {
if (harvestScheduler \!= null) {
            harvestScheduler.close();
        }
...
}

public void testGetHoursPassedSince() throws Exception {
        harvestScheduler = submitNewJobsAndGetSchedulerInstance();
  }
```

Note: The HarvestSchedulers close call delegated to the Cleanup method.

Conclusion: The CleanupIF should never be used on Singletonic classes as this leads to complicated and brittle code, and may lead to inconsistent application behavior. A solution could be to split the HarvestScheduler into a plain thread execution scheduler singleton, and a 'JobManager' class containing the business logicextending the CleanupIF. Another solution might be to replace the singleton pattern with a Inversion-of-control pattern, delegation the creation control to a class/container external to the HarvestScheduler and its users.

Consistency in equals and hashcode implementations

Overriding the equals/hashcode methods defined on the *Object* class is very difficult to implement correctly (see Effective Java by Josh Block for a description of the minefield buried here). We should instead try to delegate the responsibility of overriding these methods to a framework to insure correct and consistent implementations. Possibilities her are:

Solution	Pros	Cons
Use the IDE's built-in code generators to generate equals and hashcode methods	• No new framework need to be added to the project	• Will not be consistent across the system because of the different IDE's and configurations used by the developers. • May also be difficult to maintain, as a developer needing the update the attributes used in the equals and hashcode methods will properly have to generate to regenerate the full implementation, thereby cluttering the SCM with bloated changeds. • Developers need to remember to add any new attributes to both the hashcode and equals methods.
Use a framework with hashcode and equals generators, like the EqualsBuilder and HashCodeBuilder from the Apache Commons Lang library.	• Short and concise hashcode and equals implementation code. • Consistent implementations. • Relatively easy to maintain.	• May need inclusion of a new framework. • More dependent on the developers using the *EqualsBuilder* and *HashCodeBuilder* correctly. • Developers need to remember to add any new attributes to both the hashcode and equals methods.

Use a data object generation framework, which would take care of many of the common data object challenges. An example of such a framework can be found in the JEntity project, which is maintained by our own Mikis Seth Sørensen.	• No developer involvement in implementing any data object code, nor any test code. • Consistent data object implementation. • Correct data object implementations • Formal domain model. • Data objects will be auto generated, decreasing the source base.	• Introduction of new framework. • Some JEntity development work might be needed.

Which solution do you think we should use for eqauls() and hashcode() maintainance (Log In to vote.)	
Choices	**Your Vote**
Continue as usual	
Use IDE geenration	
Use Apache framework	
Use Jentity	

Enforcing a nice 3-tier architecture

One of the most know architectural patterns is the 3-tier separation of applications into:

- GUI layer: Contains the code for user interactions
- Application/Server layer: Server side logic
- Data layer: The data storage

The idea here is the reduce the complexity of the complete system by partitioning the code into 3 separate sub system. The interfaces between the component should be as clean as possible with a loose directional coupling GUI -> Server -> Database.

The reason for bringing this up is that I have run into difficulties fixed the JobDAOTester#testGetStatusInfo test, because of a creep of gui related logic into the data layer code. The problem is located in the HarvestStatusQuery constructor:

```
...
public HarvestStatusQuery(ServletRequest req) {
  String[] statuses = (String[]) UI_FIELD.JOB_STATUS.getValues(req);
    for (String s : statuses) {
      if (JOBSTATUS_ALL.equals(s)) {
        this.jobStatuses.clear();
        break;
      }
    this.jobStatuses.add(JobStatus.parse(s));
  }
...
```

First of all, the data - server layer decoupling is broken because this data layer specific method contains a server type argument. This means the data layer has knowledge of the server implmentation (it is a web service).

But even worse is that the GUI logic is exposed in this code. Both by the type of data used (f.ex. UI_FIELD), but also in the methods from these classes which contains functionality specific for the gui. In case of the UI_FIELD.JOB_STATUS.getValues(ServletRequest)

```
public String[] getValues(ServletRequest req) {
  String[] values = req.getParameterValues(name());
  if (values == null || values.length == 0) {
    return new String[] { this.defaultValue };
  }
  return values;
}
```

where undefined values are assumed to mean some kind of default values should be used. This is properly driven by some GUI intrinsic behavior, but appears confusing seen from a purely data model point of view.

The fundamental problem here is that you can not work with the data layer code without understand both the server and GUI usage of this functionality. This makes is much more difficult to understand the full context of this code and makes the code very brittle.

The reason for the added problem with exposing gui related functionality, is that it very difficult to validate gui code with automatic testing.This is one of the motivations for the many GUI centric design patterns like the MVC pattern, which are attempts to isolate the presentation specific code even further compare to a simple 3-tier architecture.

An example of the dangers of working with such closely couple code would be the changing of the HarvestStatusQuery constructor to better reflect only data layer concerns. This will certainly break the application in some way, but the precise implications are difficult to isolated and the automatic regression test will not be very helpful, because is properly doesn't affect the GUI code.

Increase max linelength

I've started a crusade for less wrapping of code lines here: Linie length should be more than 80 characters.

The readability of the code in my IDE would be greatly increased, eg.

Line length = 80

```
assertEquals(
                "Should have created one job after starting job dispatching",
                1, dao.getCountJobs());
        ((JMSConnectionMockupMQ) JMSConnectionFactory.getInstance())
        .waitForConcurrentTasksToFinish();
        assertEquals("The job should still be marked as new", 1,
                IteratorUtils.toList(dao.getAll(JobStatus.NEW)).size());
        assertEquals("The job should not have been marked as submitted", 0,
                IteratorUtils.toList(dao.getAll(JobStatus.SUBMITTED)).size());
        TestMessageListener normalMessageListener = new
TestMessageListener();
        JMSConnectionMockupMQ.getInstance().setListener(
                JobChannelUtil.getChannel(JobPriority.HIGHPRIORITY),
                normalMessageListener);
```

Line length = 120

```
assertEquals("Sho
        ((JMSConn
        assertEqu
                1
        assertEqu
                1

        TestMessa
        JMSConnec
                r
```

Note the indention cause by class + method blocks. Additional try/if/for/while block will eat further into the available line.

What looks pretty is of course a very individual thing, and will also vary by which interface is used, eg. do you view the code on paper, side-by-side in your editor, many windows, etc.

Introduce more powerful test frameworks

Currently the NetarchiveSuite project uses an older Junit 3 for automatic testing. We should consider introducing more powerful testing frameworks to improve the value of the automatic tests. We should have a look at the following frameworks:

- TestNG: Created to fix the deficiencies found in JUnit3 (see details here and here). testNG is supported by all major tools, can coexist with JUnit test and has tools for automatic JUnit -> TestNG migration. The major advantage of switching to TestNG is the flexibility it introduce compared to Junit3 (and somewhat JUnit4). See NARC-31 for proof-of-concept task.
- Selenium: We should start automating our system tests (see NARC-23), and here we will need to find some kind of web GUI testing framework to help us access the Netarchive GUI. Selenium is a very popular chose for doing this. See NARC-32 for proof-of-concept

task.
- JAccept: See NARC-33 for proof-of-concept task.
- GreenPepper: We might also consider using for automatic system testing and integrated test specification and issue tracking. This depends on whether SBForge will provide this functionality.

Keeping the classes clean

As part of my ongoing effort to solve NARC-11 Stop using the JMS queues for queuing snapshot harvests I've it the HarvestDefinition class, and I'm a bit confused about what this classes purpose is. The name of the class and the first part of the class description implies this a clean data entity, but the last part of the description and the implementing code contains functionality for generation Job objects based on the HarvestDefinition and write these job to the database. This set of extra 'useful' functionality might seem like a good idea, but the problem is that responsibility of the HarvestDefinition class isn't any longer clearly defined.

One of the very most efficient ways to lose control of your OO design, is to muddle the responsibilities of the components/classes making up your system. In OO classes are meant to function as logical partitions of the different problem areas of the system, enabling us to reduce the system complexity and allow developers to focus on small isolated sections of code when manipulating the code. Th resposibilities of a class can be categorized into two disjunct sets:

- Business domain responsibilities: In the case of the NetarchiveSuite system examples would be HarvestDefinitions, HarvestJob, BatchJobs, Indexes, Archives etc. Here the class properly started out as a pure harvest definition data object, but was somewhere down the road extended to also know about HarvestJobs which should be distinct data object compared to HarvestDefinitions.
- Software domain responsibilities: This might be database access, communication, mapping, gui, model entity, etc.: In the case of the HarvestDefinition it properly started out as a plain model entity class, but was again a some point bloated with a functionality for mapping harvestdefinitions to HarvestJobs and database access for updating the persistent model.

Have a look at the Monster object antipattern or God object antipattern for further discussions on what to look for.

More informative naming, less inline comments

A lot of the NetarchiveSuite source code is written with very compact naming of variables and methods, which doesn't include much information about the purpose of the variable or methods. The result is that the code can become very difficult to read. This has sometimes has been mitigated by adding inline comments. Writing readable code is in my opinion is fare superior to incomprehensible code riddled with comments.

Concrete example can be found in the HarvestSchedulerTester#testSubmitNewJobsMakesDuplicateReductionInfo method:

```
public void testSubmitNewJobsMakesDuplicateReductionInfo() {
    Method m = ReflectUtils.getPrivateMethod(HarvestScheduler.class, "submitNewJobs");
    hsch = submitNewJobsAndGetSchedulerInstance();
    //Get rid of the existing new jobs
m.invoke(hsch);
    ...
}
```

A much better way to write this code would be:

```
public void testSubmitNewJobsMakesDuplicateReductionInfo() {
    clearExistingJobs();
    ...
}

/**
 * Clear the existing jobs by running a <code>submitNewJobs</code> on the scheduler.
 *
 * @throws Exception
 */
 private void clearExistingJobs() throws Exception {
    Method submitNewJobMethod = ReflectUtils.getPrivateMethod(HarvestScheduler.class, "submitNewJobs"
);
    submitNewJobMethod.invoke(harvestScheduler);
 }
```

The following changes has been made:

- The 'Get rid of the existing new jobs' implementation and documentation has been encapsulated in a dedicated method. The method is reusable so all the duplicate occurrences of the first code snippet can be replaced by a *clearExistingJobs();*

- The m variable has been renamed to a more information name.
- hsch variable has been renamed.
- The hsch variable assignment has been moved to the setUp method to be symmetric to the usage at the tearDown level. The HarvestScheduler is a singleton, so there is no need to wait until the concrete test method is called. The functionality of the HarvestScheduler should be not be dependent upon the exact instantiation time (singleton should be robust towards such invariants), so the the individual unit tests assume they have control over the HarvestScheduler instantiation, and therefore first scheduled run.

This is much more readable piece of code IMHO.

Release tests screaming for automation

I'm currently on my 4. day of (manual) release testing here in I45, and I feel there is a number of critical problems with the current practices of manual release test at the end of iterations, which could be solved by converting these tests to automatic tests. Here is a list of the deficiencies (IMHO) of the current release test approach.

- The test are very time consuming: Test I've run through so far (Test1,2, 10) took me around 2 days each to run through. Even though the time isn't dedicated 100% to release testing the constant switching back and forth between the release test makes is difficult to work efficiently on other tasks.
- The precision of the manual test are low. The manual process of reading through many pages of 'do this and verify that' are error prone, and very difficult to reproduce on a consistent manor.
- The test specifications are not very precise. As nearly all documentation written with the purpose of being read by another human being, the test specification are open to interpretation. This again means that the tests will be run in slightly different ways each time they are run.
- Lot's of redundancy/inconsistentcy: A lot of the same test functionality can be found in different tests. Small variations have been introduced across these similar bits of code, properly because of historical copy/pastes. This phenomenon often arises from attempt to 'code' extensive functionality through documentation.
- Tests are rarely run : Because of the timeconsuming nature of the current acceptance test/relase test they are only run at the end of iterations and often only a subset of the tests are run.
- Running extensive manual tests can be very boring and drain the motivation of the development team.

All these problems could be solved by writing automatic system test to replace the current manual tests. This would change the release test so:

- The automated parts of the release test, would provide the test status free-of-cost for at the end of a iteration. Eg. in I43 I spent the better part of 3 days trying to get a picture of the state of the unit test (TEST10), where in I44 the reference test result could be read directly from the Hudson continuous integration server.
- Test specification written directly as code are very precise and therefore reproducible.
- It is much easier to reuse code and avoid redundancy and inconsistency compared to manual test specifications.
- Tests can be run on a continuous integration server, providing fast feedback on changes cause the acceptance tests to break. The current unit test suite is a good example of the value of such a quick feedback functionality. After the unit test have been added to the continuous integration server, it can now be used to get a real time, reference status (unit tested) functionality of the NetarchiveSuite code, which in turn leads to much quicker detection and fixes of broken commits, which again leads to faster code changes and bolder design maintenance (refactorings).
- The energy of the development team can be switched from laboring through exhausting manual test, to improving the quality of the tests.
- Stress testing, performance test, regression test, multiple platform tests, etc. becomes feasible because of the cheapness of running tests.

There is of course some short-comes of automatic system tests. Among these are:

- System tests can be pretty time consuming to implement because of the rich fixtures/interfaces they have to work with. This could be GUI's, databases, webservices, OS levels stuff, etc.
- System tests are also notoriously expensive to maintain: Because of the complicated environment making of a system test, it is prone to stop working as soon as parts of the system changes. This can on the other hand be a good thing, as these changes may be cause for further attention.
- Care should be taken to maintain human documentation of the automatic tests (readable test specifications. This is because the test specification are the primary means of QA'ing the system tests, and essential in enabling the usage of the system test as acceptance tests/release tests. See Introduce more powerful test frameworks for a couple of solutions to this challenge.
- Automatic test can not validate things like documentation and look & feel, so QA of these project aspects still need to be performed manually.

Nevertheless, I think a automated system test would great help increase the development efficiency of the NetarchiveSuite project and make it more fun to be a part of the development team. In my experience a highly automated QA foundation is a critical ingredient in the creation of a motivated, high performance teams. Without investing the work needed to implement a automatic system test framework it will become increasing difficult to make introduce new functionality and QA the application.

Rename HARVEST_CONTROLLER_PRIORITY setting

The current *HARVEST_CONTROLLER_PRIORITY* setting implies that the harvest controller architecture is based on a number of harvest controller instances, each with a specific priority. This does not reflect the current setup, where there only exists one 'HarvestControllerClient' which dispatches HarvestJob to the harvester via. 2 message queues. The HarvestControllerClient choses which queue to use based on the *JobPriority* of the concrete job. This means that the *HARVEST_CONTROLLER_PRIORITY* setting isn't used on the client side. On the server side

the setting is used to find the relevant queue to consume messages from. So if we wish to use the name of the setting to describe the settings purpose, is should be called something like: *HARVEST_SERVER_JOB_CHANNEL* and the type should change to a ChannelID.

The current code is ridelled with code like:

```
String priority = Settings.get(
                HarvesterSettings.HARVEST_CONTROLLER_PRIORITY);
  ChannelID result;
  if (priority.equals(JobPriority.LOWPRIORITY.toString())) {
    result = Channels.getAnyLowpriorityHaco();
  }
  else {
    if (priority.equals(JobPriority.HIGHPRIORITY.toString())) {
      result = Channels.getAnyHighpriorityHaco();
    }
    else throw new UnknownID(priority + " is not a valid priority");
  }
NetarkivetMessage naMsg = new DoOneCrawlMessage(theJob, result, TestInfo.emptyMetadata);
```

Which could be replaced by the much shorter and clearer:

```
NetarkivetMessage naMsg = new DoOneCrawlMessage(
    theJob, Settings.getChannel(HARVEST_SERVER_JOB_CHANNEL), TestInfo.emptyMetadata);
```

Another renaming we might consider is renaming the *HarvestControllerClient* to *HarvestJobDispatcher* to better describe what this class is doing. It really isn't controlling anything, and using 'Client' as a marker in a distributed architecture doesn't say much, many components are acting as clients and servers, often both at the same time. The *HarvestControllerServer* could then be renamed to just HarvestServer. Colin mentioned that the Controller part may have come from the classes responsibility of controlling a Heritrix instance, but in my opinion this is just a ordinary delegation, which shouldn't be reflected in the classes public name (a classes name should primarily reflect the services it exposes, not how it implements them, eg. controls Heritrix to do the actual crawls).

Reviewing test code

I think we should seriously consider start review the test code on a continuous based as part of the general reviewing process. The reasons for this are:

- The quality of the unit test code is not very high. 3/4 of the refactorings I have done so far on the Nertarchive project has been on test code.
- The the automatic tests are a powerful component in the development process QA, and the best way to ensure the quality of these tests are to review them. Compared to the main code which get's verified though the various test, the automatic tests needs some other QA mechanism.
- Defining functionality: Test code is a powerful mechanisme for defining functionality/acceptance criteria, and this aspect of the test code should be part of the general feature complete reviews.
- Describe functionality: I can be difficult for developers and reviewers to understand the the higher level meaning of classes and methods. Here the test code can again help us, by providing a different specification of what a class or method does/should do. If the test code is included in the general feature reviews, the reviewer would gain a better understanding of the ideas and overall design of the main code of documentation.
- Highlight design: The test cases is a very good example of how the main class/unit may be used. A bad test code design or readability can in many cases be a symptom of a badly designed main class interface. In this way the test code can be a powerful tool for spotting class which should have their api refactored (I stumbled into the CleanupIF vs. Singleton issue this way).

Switch to a more modern mock framework

The currently used mock framework Mockobjects latest release is from 2003, which means this is a pretty dead project.

Let's try to use a more modern mockup framework like Mockito. I'll try out Mockito on the harvestscheduler-refac branch.

The weakness of the CleanupIF

In the course of fixing the Singleton/Cleanup entaglement problem described in the CleanupIF vs. Singleton thread, I removed the singleton aspect of the HarvestScheduler class, but I'm still struggling with implementing a logically consistent lifecycle of the class.

I think the reason for this is that the current CleanupIF design is broken, because it introduces a asymmetrical object lifecycle where it is difficult to control what the state of the object is. The default object lifecycle is the symmetric:

```
Creation/Construction
  Reference
     ->State: Active
  Dereference
Destruction
```

Now the CleanupIF introduces a new step in this lifecycle so the lifecycle becomes:

```
Creation/Construction
  Reference
     ->State: Active
    CleanupIF
  ->State: Inactive
  Dereference
Destruction
```

It isn't exactly clear from the CleanupIF name that the functionality defined here is actually a lifecycle step, but the cleanup functionality will make the object unusable (and no way to 'restart' the object), which implies that the idea properly it to dereference the object quickly afterwards. The CleanupIF javadoc says this more directly *"Interface for classes which can be cleaned up by a shutdown hook"*. Many of the CleanupIF classes tries to clarify this by defining a close() method, which implies that we are starting to end the objects life.

In my opinion a more clean design would be to define something like a *ComponentLifecycle* interface defining 2 operations:

- start: Starts the component, creates connection, starts threads etc.
- shutdown/stop: Does the opposite of the start implementation.

This means the life of a *ComponentLifecycle* would become:

```
Creation/Construction
  Reference
  -> State: Inactive
    Start
     ->State: Active
    Shutdown
  ->State: Inactive
  Dereference
Destruction
```

This a more symmetric design and will enable a clear design of *ComponentLifecycle* objects state where they are used. The 'start' operation is actually found in many *CleanupIF* classes under different names, but they should be promoted to the same logical level as the current cleanup, to make it clear that they are symmetric lifecycle steps. A lot of the functionality found in the *CleanupIF* classes constructors might also be moved to a new *start* method to support the symmetry of the classes.

This is by the way a common component lifecycle pattern which can f.ex. be found in application servers, component frameworks etc.