Problem

How are components initialized? What about cyclic dependencies?

Solution

Component activation and initialization as currently implemented in ACS

- Initialization happens inside the container as part of component creation (="activation"). To the outside (both to manager as well as to prospective clients) creation+initialization form an atomic transaction. This means that once a client has obtained a component reference, it can call methods from the functional component interface, without worrying about the initialization state. A visible component is never un-initialized or half-initialized.
- There are two methods in the ComponentLifecycle (Java) interface which are used for initialization: `initialize` and `execute` (C++ and Py ComponentLifecycle similar)
  - `execute` is called by the container directly after `initialize`.
  - At the moment it does not make sense to have 2 separate methods here.

Component activation and initialization as originally foreseen in ACS

- The two lifecycle methods `initialize` and `execute` from the beginning on have been documented with distinct semantics (here copied from Javadoc):
  - `initialize`: Called to give the component time to initialize itself. For instance, the component could retrieve connections, read in configuration files/parameters, build up in-memory tables, ...
    Called before `execute()`. In fact, this method might be called quite some time before functional requests can be sent to the component. Must be implemented as a synchronous (blocking) call.
  - `execute`: Called after `initialize` to tell the component that it has to be ready to accept incoming functional calls any time.
    Examples: last-minute initializations for which `initialize` seemed too early, or component could start actions which aren't triggered by any functional call, e.g. the Scheduler could start to rank SBs in a separate thread. Must be implemented as a synchronous (blocking) call (can spawn threads though).
- The idea has been to later change the manager-container communication so that only the `initialize` method gets called inside the component creation transaction. Then the manager receives the component reference, but knows that the component is not yet fully initialized. When the component is actually needed (e.g. when the first client requests its reference), the manager makes a separate call to the container so that the component's `execute` method is called.
  - This change would be transparent to existing component code.
  - This change would split component initialization in order to spread it out in time. Aside from a change in the manager all container /component behavior would remain as we have it already. Especially issues of cascading creation/initialization and cyclic references (see below) would not change.

The problem

- In general there is no problem with 2 components holding mutual references to each other, or with a group of more components having some cycle in their references.
    - Probably due to a misunderstanding between Gianluca and Matej, **since ACS 4.1.1 all cyclic dependencies among components are prevented by the manager**. This will require some discussions soon, which may take into account how CCM deals with cyclic dependencies.
      Here is the change request, which in view of HSO refers only to dependencies during activation: GCH: also think that the circular dependencies should be catched before there is a request for activation to the Container of a Component. The Manager should know if it is waiting for an activation to be completed and oculd generate an exception.
- However, a component can only be retrieved once it has been initialized. That means that a component `A` which gets (and thus creates) another component `B` during its own initialization is not available for `B` before both components are fully initialized. Thus we get an error from manager if `B.initialize` calls `getComponent(A)`, for example
  ```
  Stack trace: AssertionFailed = { message='Cyclic dependency detected: [CONTROL_MASTER ->
  CONTROL_Array001 -> CONTROL_DataCapture001 -> CONTROL_MASTER].' values=
  {requestedComponentName=CONTROL_MASTER, ...
  ```
- Mutual dependencies can not be resolved using the component lifecycle methods.

Solutions

Subsystem master components were thought up as an additional higher-level construct in late 2003. They are meant to coordinate component creation and initialization, see FAQGeneralCompSubsysMaster.

- A master component has two separate initialization phases, during both of which it can create or manipulate the components it manages.
- The master component can be used to resolve cyclic dependency issues in different ways. In our example, we could either
    1. use the master component only to create the component instances `A` and `B`. Then the references to the other component can be obtained on demand. Since both components are guaranteed to already exist at that time, getting the reference through container and manager will be very fast.
       (Actually one reference could be established during initialization, and the othe one on demand.)
    2. especially if "on demand" creation of references should not be possible for some reason, the master component could create both components during the first initialization sweep ("initPass1"). Assuming that `A` has already gotten `B`, the master component could call a method such as `B.initCompA` during the `initPass2` stage. Then all references will be established before the components go functional. The drawback of the latter approach is that the method `initCompA` would show up in B's functional interface where it may not look good.

If this discussion grows, we should move it to a separate page...

- In the new light of having a subsystem master component in charge of every normal component in the system, it seems that we should revisit the original design.
    - Especially the `execute` method may be removed. It has caused a lot of confusion so far, since many people have thought that its purpose was to resolve cyclic dependencies.
- We could abandon the principle that only fully initialized components are visible to clients.
    - This would easily resolve all initialization problems with cyclic dependencies: components become available immediately when they are created. Initialization would happen as a separate step. `A.initialize` can call `getComponent(B)` and vice versa; both could get a reference to their un-initialized counterpart.
    - we would have to deal with the "un-initialzed" or "semi-initialized" states though, and here lies the problem.
        - so far everyone has agreed that it should not be the responsibility of every single component implementation to decline functional method calls while initialization is still pending.
        - the ACS framework could only do this with a "tight" container as we have it only in Java at the moment. For C++ and Python we would have to introduce call-intercepting containers, either based on CORBA interceptors or on code-generated skeletons that enforce the initialization state model.
- With any dependency cycle among components (set up during activation or later), it is not immediately clear in which order such components should be deactivated.
    - Should manager detect a cloud of unreferenced components, which may still reference each other, similar to Java GC? For this case we'd have to make sure there are no longer references from non-component clients nor from immortal components.
    - should manager randomly pick one component from such a cloud and deactivate it first?

-- HeikoSommer - 08 Aug 2005

Related articles

- How can more people do development with ACS on the same machine without disturbing each other?
- Which ports are used by ACS?
- Problems connecting to ACS servers on a remote machine: bad /etc/hosts
- Why does the getComponent method of ZLegacy/ACS.ContainerServices return an object of type None?
- Why are some of my print statements not showing up in the container output section of acscommandcenter?