ABng

Active Badge – next generation
version 0.9
Implementation Description

D. Mencnarowski, M. Steinder, J. Szymaszek, A. Uszok, K. Zieliński
e-mail:{mencnar,gosia,jasz,uszok,kz}@ics.agh.edu.pl

Distributed Systems Research Group
Institute of Computer Science
University of Mining and Metallurgy in Cracow
Al. Mickiewicza 30, 30-059 Kraków, Poland

April 1998

Table of Contents

Chapter 1

Introduction

This documents contains a description of software implementation of the ABng core for the Active Badge System originally developed by ORL in Cambridge, UK. The ABng core is a middleware layer, consisting of a set of servers, on which ABng applications can be built. It is the first completely object-oriented software for the Active Badge hardware and is constructed with the CORBA distributed object paradigm. The design of the system is presented in a separate document: "ABng - Next Generation Active Badge System".

The ABng software has approximately 40 000 lines of code written mostly in C++. Only two modules have been implemented in Java. The worldwide industry's most commonly used ORB implementation - Orbix 2.2 - has been chosen as the system middleware platform for ABng.

The construction of the software exploits two C++ templates:

1. Repository - an object that collects CORBA objects of the same type and which provides basic operations for manipulating the collection.

2. Notification - a general and scalable notification mechanism of any object's in the repository property change, built according to the observer-observed design pattern. The changes are notified to any object which is registered as an observer of the given repository or repository item.

The repository template is used for implementation of the following modules, built as separate servers in Orbix 2.2:

1. Badge Repository,

2. Badge Holder Repository,

3. Meta Equipment Repository,

4. User Repository,

5. Sensor Repository,

6. Location Type Repository,

7. ABng Location Description Repository.

All these repositories have persistent states, saved on disc in separate data files. Two repositories: Location Description and Equipment Repository are implemented in Java and they are not using the repository templates. The reason is that serialisation of type *any* is implemented in Java but causes serious portability problems in C++.

The repository servers are registered in Name Service via a dedicated server called Server Manager. This server provides notification mechanisms of the servers' existence, necessary to support easy start of the ABng system. In result, ABng servers started in any order are able to find each other.

The sighting processing is performed by the Main Sighting processor and the View processor. They communicate with each other and with poller using the notification mechanisms mentioned above.

The documentation specifies also Scheduler that is a server dedicated for badge and sensor commands processing and general template of the client useful for repository testing programs implementation.

The structure of the document is as follows. Second Chapter describes Server Manager implementation. Next the Scheduler server is specified. Chapter 4 describes Repository and Notification templates. Next 9 chapters are dedicated to the repositories implementation description. Next Main Sighting processor and View processor is described. The document is ended with Client template description.

Chapter 2

Server Manager

2.1 Functional Specification

2.1.1 Functionality description

This module is intended as a synchronization mechanism among servers in *ABng_Core*. It is set atop of CORBA Name Service. Unfortunately the Name Service does not provide a functionality of a notifier, thus it was necessary to create an additional layer.

The *server_manager* provides an interface one can use to register its reference in associated *NS*. This a most common part of the *server_manager*, the other allows you to register your interest in being informed when a particular server denoted by scoped name is start up, or shut down, one can also register its interest in being informed about servers starting up, or shutting down in a specified context (a subdirectory).

The module is essential for work of the *ABng_Core* thus individual servers may be start up (and shut down) in any order you like, because when a server depends upon work of other servers it is called back by *server_manager* when any of these servers starts up, thus it can reestablish all connection, refresh information associated with a particular server.

Although the *server_manager* is not in any way restricted to *ABng_Core* system; it is believed not to have hidden dependencies on *ABng_Core*, it was implemented as a part of *ABng_Core* system.

It should not be used for large systems with many servers and observers, because it scales badly, since it creates a single point of control in a distributed system, vulnerable to a failure.

2.1.2 Interfaces description

```
#ifndef __ABNG_SERVER_MANAGER_IDL__
#define __ABNG_SERVER_MANAGER_IDL__

module ABng_Server_Manager {
  exception Server_Still_Active {};

  exception Unknown_Server_Name {};
```

```
    exception Invalid_Server_Name {};

    exception Invalid_Server_Reference {};

    interface Callback;

    typedef sequence<string> Name;
    typedef unsigned long uid;

    interface Register {
      void register_server( in Object server_reference,
                            in Name complex_name)
        raises( Server_Still_Active,
                Invalid_Server_Name,
                Invalid_Server_Reference);

      void unregister_server( in Name complex_name)
        raises( Invalid_Server_Name,
                Unknown_Server_Name);

      uid register_callback( in Callback callback_object,
                             in Name server_name);

      uid register_context_callback( in Callback callback_object,
                                     in Name context_name);

      void unregister_callback( in uid identifier);
    };

    interface Callback {
      oneway void server_startup( in Object server_reference,
                                  in Name complex_name,
                                  in uid callback_id);

      oneway void server_shutdown( in Name complex_name,
                                   in uid callback_id);
    };
  };


  #endif                                   // __ABNG_SERVER_MANAGER_IDL__

  //
```

The *server_manager* implements interface `Register`:

register_server puts a reference `server_reference` in name server context denoted by all but last components of `complex_name` using name specified by last component. The sequence of string is translated into `CosNaming::NameComponent` in such a way that `id == kind` (see COSS - Name Service Spec.) All callbacks registered for this name are activated with `server_startup` message.

Exceptions raised:

Server_Still_Active Operation failed because there is another server running that uses that name. Terminate the other one to register properly.

Invalid_Server_Name The name consist of 0 elements.

Invalid_Server_Reference The server reference is nil.

unregister_server removes the entry from Name Service denoted by `complex_name`. All callback registered for this name are activated with `server_shutdown` message.

Exceptions raised:

Invalid_Server_Name The name consist of 0 elements.

Unknown_Server_Name There is no server registered for this name.

register_callback subscribes to all events regarding a server specified by `server_name`. The `callback_object` will be used to notify about server startups and shutdowns. The operation returns an unique identifier that may be used for future operation, it is also sent back for every callback so you can distinguish among calls if you have registered a few callbacks using the same `callback_object`.

register_context_callback subscribes to all events regarding all servers in a context denoted by `context_name`.

unregister_callback unsubscribes from previously registered callback using the identifier returned by `register_callback` or `register_context_callback`.

The `Callback` interface is implemented by clients of *server_manager*.

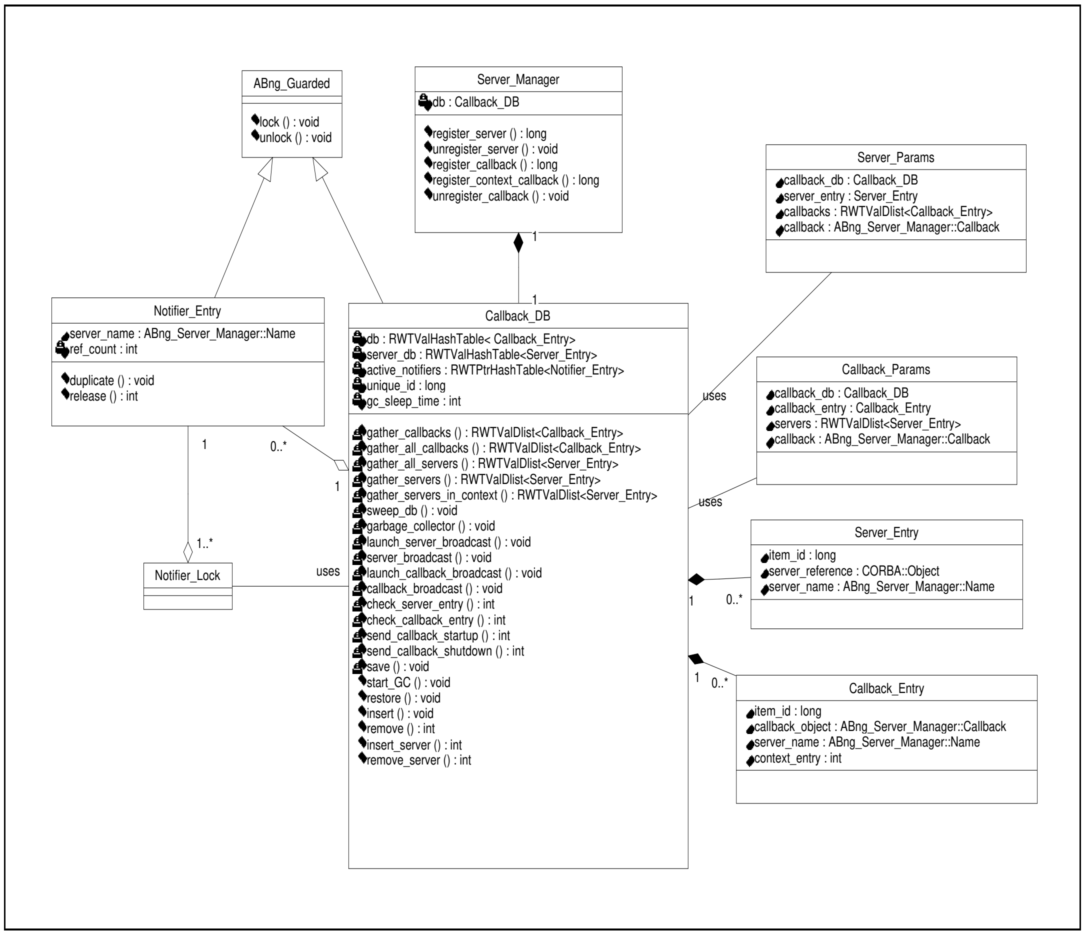
server_startup is called whenever the server registers itself with `register_server`. It is called also after registering a callback for a server that is already running, or when *server_manager* starts up.

server_shutdown is called whenever the server unregisters itself with `unregister_server`. It may by called also if *server_manager* detects a crash of a server.

2.1.3 Relationships

The module does not have inherit relationship to any of *ABng_Core* modules, altough it is used by them. On the other hand it is tightly coupled to the Name Service.

2.1.4 Object model



2.1.5 Implementation classes description

Server_Entry
encapsulates a registered server, consists of server name, its reference and a unique identifier assigned during `register_server` operation.

Callback_Entry
enapsulates a callback, consists of a server (or context) name, a reference of a callback object and an unique identifier. It holds normal callbacks and context callbacks.

Notifier_Entry
holds an information about a server name. The object exists as long as there is a callback being called with information about this server. The class is used by `Notifier_Lock`.

Notifier_Lock
prevents two or more callbacks being made with information about the same server, thus the callback for a specified server are serialized and as there are mutilple threads running notifiers the information is always accurate. It was introduced due to a problem that appeared: it was possible that one thread sent an information about server startup and later another thread sent an information about about server shutdown, but the actual sequence of events was counterwise.

Server_Params

holds all information necessary to inform all observers of a given server about a state change

Callback_Params

holds information about all servers running that the specified observer is interested in. It is used after a call to `regiser...callback`, when *server_manager* collects information about all servers in interest and spawns a thread to notify observer about running servers.

Callback_DB

The core class of the component *server_manager*.

gather_callbacks returns all callback entries interested in named server

gather_all_callbacks returns all callbacks

gather_all_servers returns information about a named server, Name Service is consulted if necessary and permitted (by "`check_NS`" parameter)

gather_servers_in_context returns list of all servers registered in named context, Name Service is consulted if necessary and permitted (by "`check_NS`" parameter)

sweep_db checks all references of currently kept server and callback entries, removes "dead" ones.

garbage_collector periodically calls "`sweep_db`". The period is set by `Callback_DB` constructor. This function never returns and should be called in a separate thread

launch_server_broadcast spawns a new thread. The thread calls all callback entries with an information about a specified server

launch_callback_broadcast spawns a new thread. The thread calls the callback entry with information about specified servers

check_server_entry — a callback method. Checks if a server is alive, returns 0 if no action is required, +1 if server is dead and should be removed

check_callback_entry — a callback method. Checks if a callback is alive, returns 0 if no action is required, -1 if callback is dead and should be removed

send_callback_startup — a callback method. Invokes "`server_startup`" method on callback object with details about the server. Returns 0 if no action is required, -1 if callback does not respond and should be removed

send_callback_shutdown — a callback method. Invokes "`server_shutdown`" method on callback object with details about the server. Returns 0 if no action is required, -1 if callback is not responding and should be removed.

save saves persitent state; all registered callbacks.

2.1.6 Persistency

Persitent state is separated in two areas: information about servers is held by Name Service while callbacks are managed by *server_manager* itself in a file `server_mgr.db` in the current directory.

2.1.7 Configuration and property management

```
Usage: ./server_mgr
                  [-g <garbage_collector_sleep_time>]
                  [-t <orbix_implementation_timeout>]
```

2.1.8 Cocurrency control and locking

The access to the database of servers and callbacks is held with a lock on instance of `Callback_DB` class. All methods are thread–safe, tha results are always value–based, so they can be operated freely, altough one should take care of variables that may change since the operation returns. If such a misinformation may lead to faults of a program, further synchronization mechanisms are required.

One of them is `Notifier_Lock` class that is used by methods {`server,callback`}`_broadcast`. The class synchronizes notifier calls being made about a speicified server. It prevents two threads from sending information about two events in a sequence counterwise to the actual one.

2.1.9 Versioning

Future versions may include an interface for setting information about executable image wise layer resposible for starting servers on demand.

2.1.10 CORBA services usage

The component is based solely on CORBA Name Service. No other services are required.

2.1.11 Life cycle management

The module saves its peristent state, so it is safe to shutdown and startup again. There is a side effect: all registered callbacks will be informed about startups of all entities they are bound to, that is not necessary a real fact. When the `server_manager` starts it is unaware of information it has sent during a previous cycle, so it tries the safest bet; all servers have just started.

2.2 Implementation Specification

2.2.1 Source code organisation

server_manager/server_mgr.cc, server_manager/server_mgr.hh These files contain definitions and declarations of all classes of *server_manager* module

server_manager/server_mgr_cl.cc A test client for the module

Chapter 3

Scheduler

3.1 Functional Specification

3.1.1 Functionality description

ABng scheduler deals with incoming commands issued by users or by other parts of system working on behalf of users. All incoming requests are queued for later execution, the scheduler fetches them when the communication channel they requested is free and ready for requests. A user may specify timeouts for requests he or she issues. Commands are retried when timeout expires, but a user may limit number of retries.

Requests directed to badges are sent through nearest sensor, or a caller may specify set of sensors involved.

It possible to set a time when a request will be executed, otherwise commands are sent immediatetely after acquiring communication channel.

A client of the scheduler after command invocation is given a request identifier, that is used in all subsequent messages from scheduler to client. The client is called back after each unsuccessful command retry, and of course when the request completes execution.

The caller expresses its interest in being called back when some situation occurs; the command is sent to a sensor, reply arrives, or an error is reported.

The scheduler is able to send a variety of pre-built commands, there are provided also generic requests, thus one can sent commands unknown at build time, but you must know the protocol-level details of the underlying hardware.

3.1.2 Interfaces description

```
#ifndef __SCHEDULER_REQUEST_QUEUE_IDL__
#define __SCHEDULER_REQUEST_QUEUE_IDL__ 1

interface Scheduler_Request_Queue {

    struct Scheduler_Request {
        string uid;
        long request_id;
```

```
        Sensor_Ids sensors;
        Badge_Ids badges;
        Octets arguments;
        string command_name;
        ABng_Core::Command_Context command_context;
        Time request_time;
    };

    typedef sequence<Scheduler_Request> Scheduler_Requests;

    Scheduler_Requests list_user_requests(in string user_id)
        raises (Domain_Repository::Destination_Unreachable);
    Scheduler_Requests list_all_requests()
        raises (Domain_Repository::Destination_Unreachable);
    void remove_request(in long request_id)
        raises (ABng_Core::Unknown_Request_Id,
                Domain_Repository::Destination_Unreachable);
};

#endif                                  /* __SCHEDULER_REQUEST_QUEUE_IDL__ */
```

list_user_requests*instringuser_id* Returns description of all currently executing requests issued by user named *user_id*.

list_all_requests Returns description of all currently executing requests.

remove_request*inlongrequest_id* Removes request identified by *request_id* from request queue, raises an exception if *request_id* is invalid.

```
#ifndef __SCHEDULER_CALLER_IDL__
#define __SCHEDULER_CALLER_IDL__ 1

// The interface used for sending command replies. Should be provided
// by a command caller.

interface Scheduler_Caller {

    oneway void challenge_reply(
        in Sensor_Id sensor,
        in Badge_Id badge,
        in Octets random,
        in Time ts,
        in long request_id,
        in long repeat_num);

    oneway void sound_reply(
        in Sensor_Id sensor,
        in Badge_Id badge,
        in Octets sound,
        in Time ts,
        in long request_id,
        in long repeat_num);
```

```
oneway void set_state_reply(
    in Sensor_Id sensor,
    in Badge_Id badge,
    in octet status,
    in Octets sound,
    in Time ts,
    in long request_id,
    in long repeat_num);

oneway void get_state_reply(
    in Sensor_Id sensor,
    in Badge_Id badge,
    in octet major_version,
    in octet minor_version,
    in octet status,
    in Time ts,
    in long request_id,
    in long repeat_num);

oneway void set_home_reply(
    in Sensor_Id sensor,
    in Badge_Id badge,
    in Octets address,
    in Time ts,
    in long request_id,
    in long repeat_num);

oneway void get_home_reply(
    in Sensor_Id sensor,
    in Badge_Id badge,
    in Octets address,
    in Time ts,
    in long request_id,
    in long repeat_num);

oneway void get_station_state_reply(
    in Sensor_Id sensor,
    in octet major_version,
    in octet minor_version,
    in octet status1,
    in octet status2,
    in Time ts,
    in long request_id,
    in long repeat_num);

oneway void sensor_command_reply(
    in Sensor_Id sensor,
    in Octets reply_body,
    in Time ts,
    in long request_id,
    in long repeat_num);
```

```
    oneway void badge_command_reply(
        in Sensor_Id sensor,
        in Badge_Id badge,
        in Octets reply_body,
        in Time ts,
        in long request_id,
        in long repeat_num);

    oneway void null_message(
        in Sensor_Id sensor,
        in Time ts,
        in long request_id,
        in long repeat_num);

    oneway void request_processed(
        in Sensor_Id sensor,
        in Badge_Id badge,
        in Time ts,
        in long request_id,
        in long repeat_num);

    oneway void error_occured(
        in Sensor_Id sensor,
        in Badge_Id badge,
        in ABng_Core::Command_Error_Code error_code,
        in long request_id,
        in long repeat_num,
        in long retry_num);

    oneway void request_completed(
        in long request_id);
};

#endif                                          /* __SCHEDULER_CALLER_IDL__ */
```

Interface *Scheduler_Caller* is not implemented by *Scheduler*, but all its clients must implement this interface, so one should know how to build an implementation of the interface.

All replies related to badge commands (*challenge_reply, sound_reply, set_state_reply, get_state_reply, set_home_reply, get_home_reply, badge_command_reply*) have common arguments:

sensor is an identifier of a sensor the command was sent through

badge is an identifier of a badge that responded

ts is a time-stamp, set when reply arrived

request_id is a request identifier returned by scheduler after issuing command

repeat_num is a repeat number for which the call succeeded

Other parameters are specific to commands and are described below:

challenge_reply(..., in Octets random, ...) The additional parameter is a sequence of octets encrypted by replying badge.

sound_reply(..., in Octets sound, ...) Tones played by badge.

set_state_reply(...,in octet status, in Octets sound, ...) Status of a badge and tones played by it

get_state_reply(..., in octet major_version, in octet minor_version, in octet status, ...) Version of badge equipment and a status.

set_home_reply(..., in Octets address, ...) New address of a badge.

get_home_reply(..., in Octets address, ...) Address of a badge.

badge_command_reply(..., in Octets reply_body, ...) The core of encoded reply body after removing badge and sensor identifiers and reply code. The contents of the parameter is interpreted by client using information about underlying hardware protocol.

The common parts of replies from sensor commands are similar to those of badges with the exception of badge identifier that is not present here.

get_station_state(..., in octet major_version, in octet minor_version, in octet status1, in octet status2) Version of sensor equipment, and its status.

sensor_command_reply(..., in Octets reply_body, ...) The core encoded reply body after removing sensor identifier and reply code.

Operation *request_processed* is called by scheduler after completing command execution (if a client expressed its interest in being informed about such event). Please note that for sensor commands *badge* parameter has undefined value.

error_occured is called when scheduler is unable to deliver a command or get a reply; *error_code* parameter indicates a kind of error that occurred. For sensor commands *badge* parameter has undefined value. If *error_code* equals to *SIGHTING_TIMEOUT* both parameters: *sensor* and *badge* have undefined value and should not be examined as they are meaningless.

Please note that sighting error (*SIGHTING_TIMEOUT*) is not necessary raised because of badge not being seen around lately. It is called also when a request cannot acquire sighting (within specified period of time) because other commands in request queue are holding it.

```
 #ifndef __SCHEDULER_IDL__
#define __SCHEDULER_IDL__ 1

interface Scheduler : Scheduler_Request_Queue {

    // Send a command to specified sensors. If a sensor list is empty send to
    // all sensors.
    long issue_sensor_command(
        in Sensor_Ids sensors,
```

```
    in Octets cmd,
    in boolean has_reply,
    in octet reply_code,
    in ABng_Core::Command_Context ctx,
    in Scheduler_Caller reply_to)
    raises (Unknown_Sensor_Id,
        Invalid_Sensor_Id,
        ABng_Core::Invalid_Request_Time,
        ABng_Core::Invalid_Argument);

// Send a command through the specified sensors to the given badges.
// If there are no sensors, send a command through the nearest sensor.
// If there are no badges, send a command to all badges.
long issue_badge_command(
    in Sensor_Ids sensors,
    in Badge_Ids badges,
    in Octets cmd,
    in boolean has_reply,
    in octet reply_code,
    in ABng_Core::Command_Context ctx,
    in Scheduler_Caller reply_to)
    raises (Unknown_Sensor_Id,
        Invalid_Sensor_Id,
        Unknown_Badge_Id,
        Invalid_Badge_Id,
        ABng_Core::Invalid_Request_Time,
        ABng_Core::Invalid_Argument);

long challenge(
    in Sensor_Ids sensors,
    in Badge_Ids badges,
    in Octets random,
    in ABng_Core::Command_Context ctx,
    in Scheduler_Caller reply_to)
    raises (Unknown_Sensor_Id,
        Invalid_Sensor_Id,
        Unknown_Badge_Id,
        Invalid_Badge_Id,
        ABng_Core::Invalid_Request_Time,
        ABng_Core::Invalid_Argument);

long sound(
    in Sensor_Ids sensors,
    in Badge_Ids badges,
    in Octets sound,
    in ABng_Core::Command_Context ctx,
    in Scheduler_Caller reply_to)
    raises (Unknown_Sensor_Id,
        Invalid_Sensor_Id,
        Unknown_Badge_Id,
        Invalid_Badge_Id,
        ABng_Core::Invalid_Request_Time,
```

```
        ABng_Core::Invalid_Argument);

long set_state(
    in Sensor_Ids sensors,
    in Badge_Ids badges,
    in octet status,
    in Octets sound,
    in ABng_Core::Command_Context ctx,
    in Scheduler_Caller reply_to)
    raises (Unknown_Sensor_Id,
        Invalid_Sensor_Id,
        Unknown_Badge_Id,
        Invalid_Badge_Id,
        ABng_Core::Invalid_Request_Time,
        ABng_Core::Invalid_Argument);

long get_state(
    in Sensor_Ids sensors,
    in Badge_Ids badges,
    in ABng_Core::Command_Context ctx,
    in Scheduler_Caller reply_to)
    raises (Unknown_Sensor_Id,
        Invalid_Sensor_Id,
        Unknown_Badge_Id,
        Invalid_Badge_Id,
        ABng_Core::Invalid_Request_Time,
        ABng_Core::Invalid_Argument);

long set_home(
    in Sensor_Ids sensors,
    in Badge_Ids badges,
    in Octets address,
    in ABng_Core::Command_Context ctx,
    in Scheduler_Caller reply_to)
    raises (Unknown_Sensor_Id,
        Invalid_Sensor_Id,
        Unknown_Badge_Id,
        Invalid_Badge_Id,
        ABng_Core::Invalid_Request_Time,
        ABng_Core::Invalid_Argument);

long get_home(
    in Sensor_Ids sensors,
    in Badge_Ids badges,
    in ABng_Core::Command_Context ctx,
    in Scheduler_Caller reply_to)
    raises (Unknown_Sensor_Id,
        Invalid_Sensor_Id,
        Unknown_Badge_Id,
        Invalid_Badge_Id,
        ABng_Core::Invalid_Request_Time,
        ABng_Core::Invalid_Argument);
```

```
long reset_buffer(
    in Sensor_Ids sensors,
    in ABng_Core::Command_Context ctx,
    in Scheduler_Caller reply_to)
    raises (Unknown_Sensor_Id,
        Invalid_Sensor_Id,
        Unknown_Badge_Id,
        Invalid_Badge_Id,
        ABng_Core::Invalid_Request_Time,
        ABng_Core::Invalid_Argument);

long reset_fifos(
    in Sensor_Ids sensors,
    in ABng_Core::Command_Context ctx,
    in Scheduler_Caller reply_to)
    raises (Unknown_Sensor_Id,
        Invalid_Sensor_Id,
        Unknown_Badge_Id,
        Invalid_Badge_Id,
        ABng_Core::Invalid_Request_Time,
        ABng_Core::Invalid_Argument);

long get_station_state(
    in Sensor_Ids sensors,
    in ABng_Core::Command_Context ctx,
    in Scheduler_Caller reply_to)
    raises (Unknown_Sensor_Id,
        Invalid_Sensor_Id,
        Unknown_Badge_Id,
        Invalid_Badge_Id,
        ABng_Core::Invalid_Request_Time,
        ABng_Core::Invalid_Argument);

long set_output(
    in Sensor_Ids sensors,
    in octet output_value,
    in ABng_Core::Command_Context ctx,
    in Scheduler_Caller reply_to)
    raises (Unknown_Sensor_Id,
        Invalid_Sensor_Id,
        Unknown_Badge_Id,
        Invalid_Badge_Id,
        ABng_Core::Invalid_Request_Time,
        ABng_Core::Invalid_Argument);

long set_persistence(
    in Sensor_Ids sensors,
    in boolean mode,
    in ABng_Core::Command_Context ctx,
    in Scheduler_Caller reply_to)
    raises (Unknown_Sensor_Id,
```

```
                Invalid_Sensor_Id,
                Unknown_Badge_Id,
                Invalid_Badge_Id,
                ABng_Core::Invalid_Request_Time,
                ABng_Core::Invalid_Argument);
};

#endif                                                      /* __SCHEDULER_IDL__ */
```

All badge commands (*issue_badge_command, challenge, sound, set_state, get_state, set_home, get_home*) have a common syntax:

```
long command_name( in Sensor_Ids sensors, in Badge_Ids badges,
                   ..., in ABng_Core::Commant_Context ctx,
                   in Scheduler_Caller reply_to);
```

Common parameters:

sensors is a list of sensor identifiers the command may go through, if the list is empty a nearest sensor is chosen

badges is a list of badge identifiers the command is directed to, if the list is empty the request will be executed for all known badges

ctx is a command context for request

reply_to a call-back object (described above)

Command context is a structure that constrain the way the request is executed (sighting, reply timeout, time when command will be executed, etc.), for details please refer to *ABng New Generation Active Badge System* — section *"Command Execution"*, where you can find detailed description of all fields of the structure.

long issue_badge_command(..., in Octets cmd, in boolean has_reply, in octet reply_code, ...) *cmd* — the first element is a command code, while the others contain command – specific arguments. Note: badge and sensor identifiers are not a part of *cmd* parameter. *has_reply* — "true" if scheduler should wait for a reply, *reply_code* — the code of reply sent by a badge (if command has a reply, otherwise this argument is irrelevant).

long challenge(..., in Octets random, ...); random — a sequence of octets the badge should encrypt with its private key.

long sound(..., in Octets sound, ...); sound — a sequence of beeps the badge should play, for detailed description please refer to *Active Badge System Protocol Specification.*

long set_state(..., in octet status, in Octets sound, ...);

long get_state(...,...);

long set_home(..., in Octets address, ...);

long get_home(...,...); for details about the above commands please refer to Active Badge System Protocol Spec.

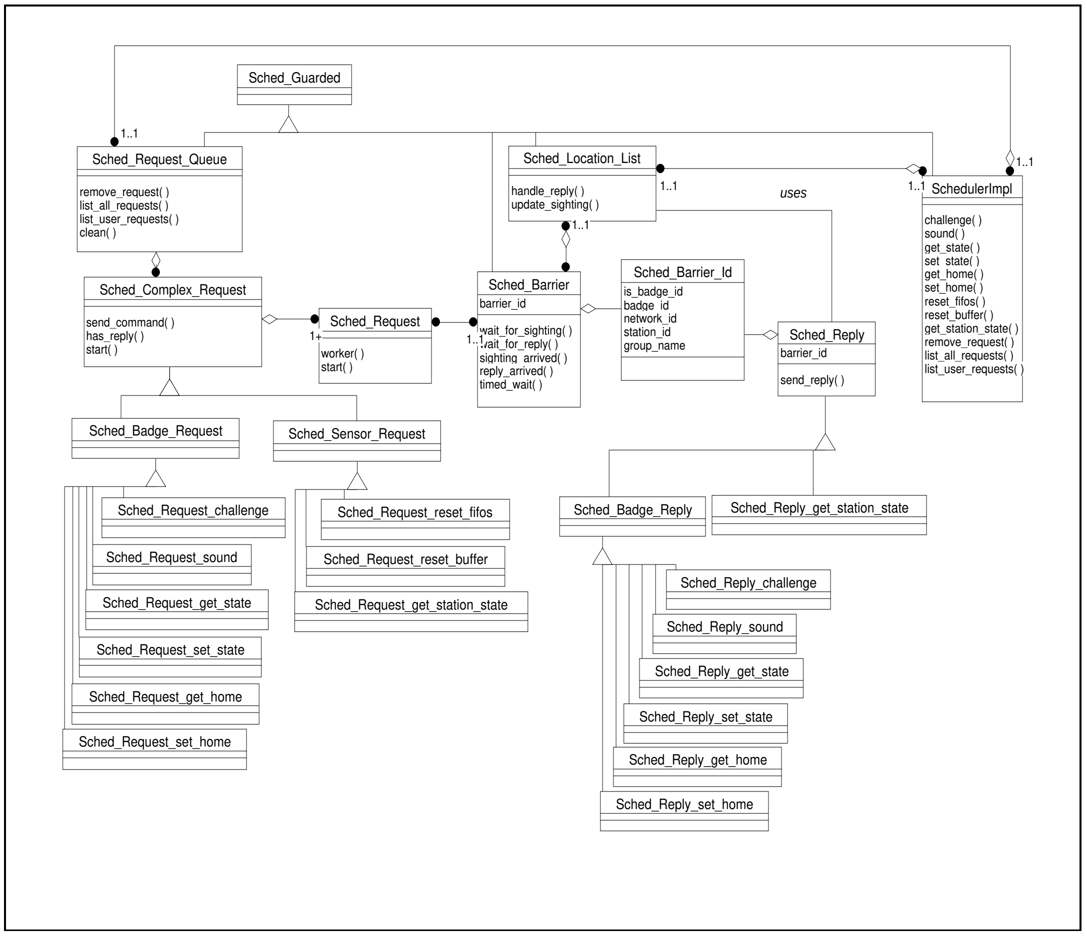
3.1.3 Relationships

Work of *Scheduler* module is based on two interfaces: *AB_Poller* and *abng_db*. They provide functionality required to implement the scheduler.

This module uses *abng_db* to check if sensors and badges have valid identifiers. On the other hand the *Scheduler* module is used by *abng_db* to implement *Sensor* and *Badge* interfaces.

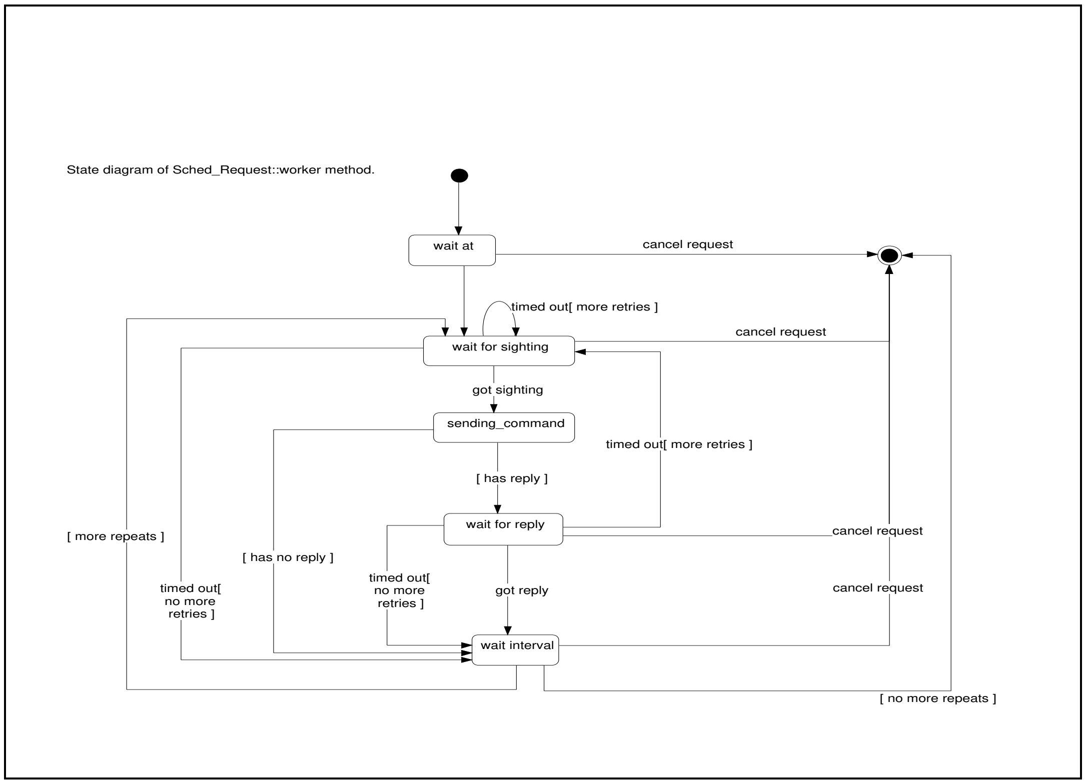
This circular dependency requires further investigation and possible use of start-up service by all modules.

3.1.4 Object model

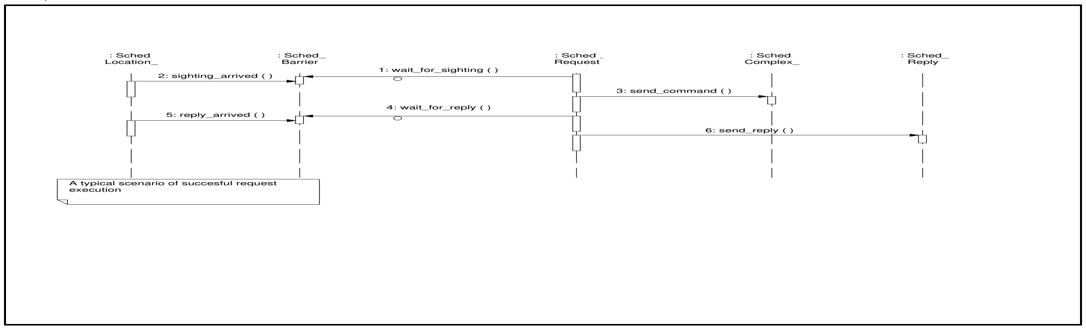


3.1.5 State model

The following algorithm is performed by all requests' threads concurrently:



Below you can see a time diagram of typical request execution. The communication is performed using conditional variables (for in-process events), or *IIOP* calls (for inter process events).



3.1.6 Implementation classes description

Sched_Guard

Class *Sched_Guard* is used for protecting methods of derived classes from being run by more than one thread of control at a time. Methods that require synchronization should declare a variable of type *Sched_Guard::auto_lock* in their body. There is also a method *Sched_Guard::mutex* that returns a reference to a mutex, this reference can be used to initialize conditional variables.

Sched_Location_List

This class acts as a mapper between badge (sensor) identifiers and their respective *barriers*. There is only one object of this class in a server.

All replies and sightings are redirected through an instance of *Sched_Location_List*.

handle_reply Finds a barrier for reply, if there is any request waiting for this reply, it is signaled and the reply object will be redirected to a waiting request, if there is no such request the reply is destroyed.

update_location_list This method is called when sighting arrives for any badge. This information is redirected to a corresponding *barrier*. See there.

get_barrier Returns a reference of a barrier for a given barrier identifier. If there is no barrier for specified identifier, the barrier is created.

Sched_Complex_Request

It is a root class for all types of request, holds information about sub-request (directed to a particular badge or sensor). The *Sched_Complex_Request* class is an abstract class, implementation of specific methods is left for derived classes.

The instances of derived classes are created in class *SchedulerImpl*, after receiving request.

send_command Sends request through *AB_Poller*; this method is called from worker method of Sched_Request after acquiring sighting.

describe_request Fills information about the request required for *list_all_requests* and *list_user_requests* operations.

accepts_reply Checks if reply matches request. Request object may examine various parameters to see if given reply is the one the request is waiting for.

Note: One of the fields the request object may examine is *sequence_number*, but it may happen that underlying hardware does not support sequence numbers, then this field is equal to zero. The request should accept such a field, as if it matched sequence number of request.

poller This method returns a pointer to *AB_Poller* object, currently returns always one object, but future implementation may be connected to several *AB_Poller* objects, thus the method will distinguish required pollers basing on identifiers of sensors.

Sched_Request_Queue

Request queue holds information about all active (uncompleted) requests. It provides implementation for all methods of *IDL* interface *Scheduler_Request_Queue*. There is only one object of this class in a server. All public methods in this class are synchronized with the exception of *clean*, which returns immediately after spawning a separate thread of control.

The only non-trivial method worth further description is *remove_request*. As it was decided that requests (and their threads) cannot be cancelled using some sort of *cancel* method from thread library; the request queue sets a special flag for all involved threads, and signals all conditional variables they could possibly be waiting for. All these threads are waken up (if they were sleeping), thus they can perform necessary cleanup. After all threads all signaled, they are joined to reclaim allocated memory and ensure that they really quit.

Thread cancellation (using thread library) requires special handlers to unlock all mutexes, as there are some other libraries involved in the process, that you can not guarantee they will remain stable.

The additional method *clean* spawns a separate thread and removes given request from queue (using method *remove_request*). This method was implemented primary for the purpose of cleaning up memory of all threads of a given request, as one thread can be joined only be another thread, the request itself cannot perform this task.

Sched_Barrier

This class serves as a synchronization mechanism for worker thread and incoming signals from poller. When worker thread wants to wait for a signal, it enters appropriate method in *Sched_Barrier*, which performs timed wait for a conditional variable.

Signals from poller are matched against barrier identifiers, next. inside a barrier they are matched against waiting request. If a signal is the one, the request is waiting for, the conditional variable is signaled.

Other possible signals may come from request queue, f.e. from *remove_request* method; this situation also requires to signal a conditional variable, but before a cancelling flag is set, thus worker thread after returning from barrier "knows", if it got what it was waiting for, or specified timeout was reached, or the request was cancelled.

Every *Sched_Barrier* class is associated with a badge or a sensor. The instances of *Sched_Barrier* are created dynamically; they are created upon first request or sighting (whatever first occurs) for the specified badge or sensor. The instances are never destroyed (except for a server shutdown).

wait_for_sighting Waits for sighting (with given timeout), upon successful acquiring sighting (operation returns +1), the barrier is hold exclusively by the calling thread till it calls *end_request*. All sightings and replies directed to the barrier are captured by this thread. If timeout expires the operations returns 0, and cancelled request returns -1.

wait_for_reply Waits for reply from command sent. The returns values mean the same as for previous method. If operation returns +1 (reply arrived), the reply object is inserted into *Sched_Request* object, the caller is responsible to delete it.

sighting_arrived Called by *Sched_Location_List* object upon sighting arrival;

reply_arrived Called by *Sched_Location_List* object upon reply arrival;

alloc_seq_number Allocates a sequence number (an identifier of request, required to drop commands with the same sequence number, as well as, to filter replies). Request thread should call it for every command repeat, but not retry.

end_request Operation called by request thread after successful (or unsuccessful) request execution, but before sleeping for a period of time specified in *interval* field of *Command_Context* structure. The operation releases *Sched_Barrier* object for other requests.

insert_request Inserts request in request queue (called at the beginning of request thread).

remove_request Removes request from request queue (called before request thread completion).

Sched_Request

Contains the worker thread of request execution, it synchronizes with associated barrier. A list of *Sched_Request* objects is hold by *Sched_Complex_Request* (and derived). It is responsible for sending request at specified time, waiting for requests and calling back to clients to inform them about the progress in command execution.

The instances of this class are created in constructors of classes derived from *Sched_Complex_Request*. The life time of an instance is terminated with method *cancel* that causes the active thread to terminate, and later with *join* to reclaim the space allocated for thread. Only after these two operations destructor may be called.

Sched_Barrier_Id

A class holding badge and sensor identifiers, it may represent a sensor identifier, as well as a badge identifier (or both, if used as a location representation). It was introduced to create a symmetry between badge and sensor requests. Thus much of the code is independent on whether a request is directed to a badge or a sensor.

Sched_Reply

Objects of this type are created when scheduler gets a response from *abng_poller* server. After creation they are redirected to specific barriers and requests to see if they comply to requirements imposed by request objects (*Sched_Complex_Request* and derived) (see method *accepts_reply* in class *Sched_Complex_Request*).

send_reply Forwards reply to a client.

min_barrier_id Returns a subset of barrier identifier; badge commands returns here only information about badge identifiers, and sensor commands returns sensor identifiers.

3.1.7 Interfaces to implementation classes mapping

Scheduler_Request_Queue is mapped to Sched_Request_Queue via SchedulerImpl class

Scheduler is implemented by SchedulerImpl

Other classes do not expose any interface to the outside world, they are used by the above two.

3.1.8 Persistency

Scheduler component does not maintain any persistent state.

3.1.9 Event notification

All clients of *Scheduler* are being called back through *Scheduler_Caller* interface. The scheduler notifies this way its client about a progress in command execution.

3.1.10 Configuration and property management

There is only one configurable property of the *scheduler* component; it is a time a sighting is considered to be valid. It is set during server start-up as a command line parameter. For details please refer to the server specification.

3.1.11 Concurrency control and locking

The *Scheduler* component is multi-threaded. For every pair of request—badge (for badge commands), and request—sensor (for sensor commands) there is created a thread of control.

Request are synchronized using dedicated objects (*Sched_Barrier*) that are responsible to ensure mutual exclusion for requests that require the same communication channel, as well as for handling replies for individual requests.

Every object of implementation holding information that changes during lifetime of an object is protected by a lock. There are few exceptions; objects accessed only from critical sections of other objects.

Estimated cost

The cost of creating a thread of control in terms of memory usage is about 5 KB. The primer factor of memory usage is a stack for every thread.

It was chosen *Solaris* threads as a thread library implementation mainly because of its light impact on kernel resources and greater flexibility than standard — *POSIX* threads.

3.1.12 Security

The only security mechanism applied in this module is use of Orbix principal to distinguish request from various users.

Although this mechanism does not impose any restrictions, it is used only for identifications purposes.

3.1.13 Versioning

It is possible to build two different versions of *Scheduler* component: a debugging one and a release version. To build a debugging version please include `-DDEBUG` to `CXXDEBUGFLAGS` in Imakefile. For release version please remove the flag.

You can also build a version instrumented with *Purify* tool (useaful in searching for memory leaks) by issuing the following command in abng_sched directory:
`make sched.pure`

3.1.14 CORBA services usage

Current implementation uses no *CORBA* services, but it is planned future use of naming and (or) trading services.

3.2 Implementation Specification

3.2.1 Source code organization

The source tree is put under control of Revision Control System (RCS), the full log of activity, updates, etc. may be found in directory abng_sched/RCS.

abng_sched/sched_barrier.cc, abng_sched/sched_barrier.hh
Sched_Barrier class definition and declaration

abng_sched/sched_barrier_id.cc, abng_sched/sched_barrier_id.hh
Sched_Barrier_Id class definition and declaration

abng_sched/sched_debug.cc, abng_sched/sched_debug.hh
Operators for writing and reading sensor and badge identifiers to and from streams.

abng_sched/sched_main.cc, abng_sched/sched_main.hh
Sched_Guarded, Sched_Location_List, Sched_Request_Queue, SchedulerImpl classes.

abng_sched/sched_reply.cc, abng_sched/sched_reply.hh
Sched_Reply class, and all derived from it

abng_sched/sched_request.cc, abng_sched/sched_request.hh
Sched_Complex_Request and all derived

abng_sched/sched_repos.cc, abng_sched/sched_repos.hh Sched_SB_Repositories

abng_sched/doc Documentation for scheduler component and server

3.2.2 Imake file description

Imakefile is located in abng_sched. To create makefile please issue the following command:
`imake -I/home/iris/grant/abng/config -DXCOMM="#" -DHasPurify`

3.3 Server process structure

Server consists if two entities: the implementation of *Scheduler* interface and a client to *Sensor_Rep* and *Badge_Rep* repositories. The clients waits for updates from repositories to provide up–to date information about badge and sensor identifiers when server wishes to check for their validity, or to retrieve all sensors or badge IDs.

3.4 Sever client description

Client process must implement *Scheduler_Caller* interface to get call backs on events it wishes to know about. For details on this interface please refer to *Scheduler Component Specification.*

3.5 Runtime environment specification

1. *abng_poller* server(s) running somewhere on the network
2. *abng_sensor_repository*
3. *abng_badge_repository*

3.6 Server life cycle

The server is intended to be up and running continuously.

As the current implementation does not maintain a state between server shutdowns and startups, there is no life cycle defined for this server.

Upon (possible) server crash all currently executing requests are lost. Clients should re-issue (after server startup) all requests they wish to be executed again.

3.7 Source code organization

Please refer to "Software Component Specification — Scheduler".

3.8 Installation procedure

There is no special action needed to setup the server, except for those mentioned in "Runtime environment specification".

The general syntax of the server:

```
abngsched <implementation_timeout>
      <-st sighting_timeout>
```

implementation_timeout used only to debug program, specifies the implementation timeout as described in *Orbix Reference Manual*

sighting_timeout is a time information about location of badge is considered to be valid, after this period *scheduler* waits for another sighting. If not specified the default value is 9 seconds.

Example:

```
abngsched -st 15
```

Chapter 4

Repository Template

The key components of the ABng system's core are repositories of items (objects) representing various both abstract and physical elements. Currently, the system consists of 9 different repositories. Additionally, there are view objects, each of which can be considered as a pair of repositories used for storing items representing locations and locatables belonging to the view. Every repository maintains a collection of items of the certain type which is specific for the given repository. This is the main reason for which interfaces implemented by particular repositories differ. In addition, functionalities offered by repositories can be a little different. Nevertheless, a common set of functions performed by every repository can be distinguished, though they may be performed in a slightly different way. These functions are:

- managing a collection of items, i.e. adding, removing and updating items stored in the collection;
- processing other system components' queries about repository contents;
- managing a registry of observers of items stored in the repository and asynchronous notification of observers of changes of the observed items;
- providing a persistent storage for items belonging to the maintained collection.

Each above function has corresponding operations in interfaces provided by particular repositories.

To avoid implementing these functions for every repository it was decided to precisely define the common functions, implement them only once and then take advantage of code-reuse when developing repositories. The common repository functionality has been encapsulated in several C++ base classes, which together constitute a generic Repository Template for development of ABng repositories. Some of these classes are in fact C++ template classes. The Repository Template consists of the following C++ classes:

- *DB_Header* – implements generation of database file headers,
- *DO_ptr* – the template class of the wrapper of observer references;

- *Basic_Observed_I* – the template class implementing basic functions related to maintenance of observer registry,
- *Observed_I* – the template class implementing functions related to maintenance of registry of observers which monitor adding new items to the collection,
- *Basic_Rep_I* – the template class implementing functions related to management of a collection of items,
- *Basic_Server* – the class providing basic building blocks for implementation of the mainline of an ABng server,
- *Rep_Server* – the class providing basic building blocks for implementation of the mainline of an ABng repository server.

The remaining part of this chapter describes the above elements in details.

4.1 DB_Header

In the current ABng release the states of objects stored in a repository are saved in a file, one per each repository. Every file consists of a header and a body. A header contains:

- system name (ABng),
- current release (1.0),
- repository name,
- last update time,
- identifier and address of the process containing the repository object,
- description of the file format in BNF

The body of a database file usually contains records describing states of objects stored in the repository.
For generation of a header of the database file the *DB_Header* class has been defined.

```
#include <iostream.h>
#include <rw/cstring.h>

class DB_Header {
  RWCString rep_name;
  RWCString ver_name;
  RWCString record_description;
  int dont_include_default_description;

public:
  DB_Header() {}
```

```
   DB_Header( const char *_rep_name,
              const char *_ver_name,
              const char *_record_description,
              int _dont_include_default_description= 0);

   friend ostream & operator << ( ostream &ostr, const DB_Header &db);
   friend istream & operator >> ( istream &istr, DB_Header &db);
};

ostream & operator << ( ostream &ostr, const DB_Header &db);
istream & operator >> ( istream &istr, DB_Header &db);

ostream & operator << ( ostream &str, const Domain_Repository::Octets &seq);
istream & operator >> ( istream &str, Domain_Repository::Octets &seq);

ostream & operator << ( ostream &str, const CORBA::TypeCode_ptr tc);
istream & operator >> ( istream &str, CORBA::TypeCode_ptr tc);
```

To construct an object of the *DB_Header* class the following information has to be provided:

- *_rep_name* – repository name,
- *_ver_name* – file format version number,
- *_record_description* – BNF syntax of data records stored in the database file,
- *_dont_include_default_description* – flag suppressing inclusion of the default BNF file syntax description into the header.

This information determines the contents of the header file produced using the stream output parameter. The remaining stream operators allow to retrieve the database header, to read and write both sequences of octets and type codes, which are commonly stored in ABng databases.

Below the format of the header is presented. The header contains variable parameters. Three of them, printed in bold, correspond to the first three arguments of the *DB_Header* constructor. Other parameters (time and date of the last database update, process identifier, process address), which are printed in italic, are computed and inserted when the header is constructed. If the *_dont_include_default_description* argument of the constructor was non-zero the syntax description part will not contain the entry for <*file*> and <*data_records*>. The last two BNF definitions appear in the header only if the definition contained in the *_record_description* parameter refers to the <*sequence_of_octets*> item.

```
# Active Badge next generation (ABng)
# Release:  1.0
#
# _rep_name Repository database
# Last update:  date time
#
# Generated by a process with pid pid on hostname
#
```

```
# Format description (version _ver_name):
#       <file>::=               <next_item_id>
#                               <number_of_items>
#                               <data_records>
#       <data_records>::=       |
#                               <data_record> <data_records>
#       _record_description
#       <sequence_of_octets>::= <hex_numbers>
#       <hex_numbers>::=        |
#                               <hex_number> <hex_numbers>
```

An example header file generated for Sensor Repository is shown below.

```
# Active Badge next generation (ABng)
# Release: 1.0
#
# Sensor Repository database
# Last update: 03/31/98 16:15:06
#
# Generated by a process with pid 1110 on iris
#
# Format description (version 1.0):
#       <file>::=               <next_item_id>
#                               <number_of_items>
#                               <data_records>
#
#       <data_records>::=       |
#                               <data_record> <data_records>
#
#       <data_record>::=        <item_id> <sensor_id>
#
#       <sensor_id>::=          <sensor_number> <networ_number>
#                               <group_name>
#
#       <sensor_number>::=      <decimal_numbers>
#       <network_number>::=     <decimal_numbers>
#       <group_name>::=         <string>
```

4.2 Object Reference Wrapper

A part of every repository is a registry of observers. Applications can manipulate with this registry by adding and removing references of their observer objects. This implies the need of making comparisons of observer references on the repository side. For example, if a new observer of a given repository item is added it must be verified whether that observer has not been already registered with this item, so its reference has to be compared with references stored in the registry. However, in Orbix 2.2 in which repositories written in C++ are implemented, there is a bug which makes comparison of object references difficult. Namely, if an application passes a reference of its object to a server several times, this reference is not always the same on the server side, even though it remains constant within the application. What is worse this is unpredictable behaviour. Consequently, the same object is denoted by several different references and represented by different proxies on the server side. As the result simple comparison of object IIOP references does not allow to determine whether they denote the same object or not. The comparison of proxy pointers (which, on the other hand, is not CORBA-compliant) does not bring the required result either.

To solve this problem it is proposed to use a wrapper of object references and to operate with wrapper objects rather than object references on the repository server side. The wrapper has been developed as a C++ template class defined below.

```
template <class T>
class DO_ptr {

public:

   T my_object;

   DO_ptr(T _my_object = 0);
   DO_ptr(const DO_ptr&);
   ~DO_ptr();

   int operator == (const DO_ptr&) const;

   DO_ptr& operator = (const DO_ptr&);

};

template <class T>
unsigned hash_proxy(const DO_ptr<T>& observer_p);
```

A parameter of the template should be a reference type (*ptr*). The wrapper object is initialised with an object reference or another wrapper object (the copy constructor). The main role of the wrapper is to perform equality testing and this is implemented by the comparison operator. To determine whether two references (one hold within the wrapper denoted by *this* pointer and another one hold in the wrapper object given as the argument) point to the same object, the comparison operator compares elements of the Orbix object references.

The registry of observer objects, described in the next section, does not directly manipulate with observer references, but it uses reference wrappers. These are reference wrappers which are stored in the hash tables maintained by the registry. Therefore, a hash function, for proxy wrappers was needed. The input values of the implemented hashing algorithms are elements of an Orbix reference.

The above problems have been solved in the version 2.3 of Orbix. Hence, the using of reference wrapper will not be needed after the ABng software is ported to new Orbix release.

4.3 Basic_Observed_I

A mechanism of notification of changes of items stored in ABng repositories is based on the well-known Observed/Observer design pattern [2]. An application which wishes to be informed of changes within a repository has to implement an observer interface. The repository maintains a registry of observers. An application can subscribe a reference of its observer objects into the repository's registry using an observed interface implemented by the repository. When an observer is registered a list of identifiers of repository's items which are to be monitored is given. Alternatively, an application can subscribe an observer of an individual item using its IDL interface. An item will locally forwards such a request to the repository registry. Hence, with every observer a list of items monitored by this observer is associated and this mapping is also maintained by the observer registry inside the repository.

An observer which registered its interest in a concrete item will be notified when any of the item attribute changes its value or the item is removed. Additionally, an observer can register its interest in the event of additions of new items. The repository maintains a separate registry to store references of observers monitoring adding new items.

In ABng there are currently four sets of the observed/observer interfaces:

- *Static_Data_Observer*, *Static_Data_Observed_R*, *Static_Data_Observed_E*
- *System_Static_Data_Observer*, *System_Static_Data_Observed_R*, *System_Static_Data_Observed_E*
- *View_Data_Observer*, *View_Data_Observed_R*, *View_Data_Observed_E*
- *Dynamic_Data_Observer*, *Dynamic_Data_Observed_R*, *Dynamic_Data_Observed_E*

Every set of interfaces is used to report changes of items of different types. The first two sets are related to repositories whose interfaces are defined in the *Domain_Rep* and *ABng_Core_System* modules. The third one is used to report changes of location and locatables belonging to a view. The interfaces of the last set serve to report location changes related to locations and locatables. The role of each of three interfaces included in every set is as follows:

- **_Observer* – interfaces implemented by components willing to monitor items,
- **_Observed_R* – interfaces implemented by repositories or views,
- **_Observed_E* – interfaces implemented by items belonging to repositories or views.

Every repository and a view provides an **_Observed_R* interface. Its implementation maintains a registry of observers of items belonging to a repository. Additionally, a repository providing an **_Observed_R* belonging to one of the first three interface sets maintains a registry of observers monitoring adding new items. This functionality is not supported by the *Dynamic_Data_Observed_R* interface.

In the remaining part of this section, the *Basic_Observed_* C++ template class is described. This class implements the common functionality offered by all **_Observed_R* interfaces. The next section presents the *Observed_I* class which implements functions specific for three of four **_Observed_R* interfaces and related to the monitoring of adding new items.

4.3.1 Class Definition

The *Basic_Observed* template is shown below.

```
#include <rw/tvhasht.h>
#include <rw/tvhdict.h>
#include "abng_guarded.hh"

#ifdef ORBIX
#include "DO_ptr.h"
#endif

template <class Observer_Types>
class Basic_Observed_I : public virtual ABng_Guarded {

public:

  Basic_Observed_I();
  virtual ~Basic_Observed_I();

  typedef Observer_Types::Observed Observed;
  typedef Observer_Types::Observer Observer;
  typedef Observer_Types::Observers Observers;
  typedef Observer_Types::Observer_ptr Observer_ptr;
  typedef Observed::Observer_Entries Observer_Entries;
#ifdef ORBIX
  typedef DO_ptr<Observer_Types::Observer_ptr> SDO_ptr;
#endif

  typedef RWTValHashTable<Observer_ptr> Observer_List;
  typedef RWTValHashTableIterator<Observer_ptr> Observer_List_Iterator;

  virtual Observer_Entries *
    observer_list (
    CORBA::Environment &IT_env=CORBA::default_environment)
    throw (CORBA::SystemException);

  virtual void attach_observer (
    Observer_ptr obs,
    const Domain_Repository::Repository_Item_Ids& observed_id_list,
    CORBA::Environment &IT_env=CORBA::default_environment)
    throw (CORBA::SystemException,
           Domain_Repository::Unknown_Repository_Item_Ids);

  virtual void detach_observer (
    Observer_ptr obs,
    const Domain_Repository::Repository_Item_Ids& observed_id_list,
    CORBA::Environment &IT_env=CORBA::default_environment)
    throw (CORBA::SystemException,
           Domain_Repository::Unknown_Repository_Item_Ids,
           Domain_Repository::Unknown_Observer,
           Domain_Repository::Unmatched_Observer_Item_Ids);
```

```
    virtual void private_attach_observer (
      Observer_ptr obs,
      const Domain_Repository::Repository_Item_Ids& observed_id_list,
      CORBA::Environment &IT_env=CORBA::default_environment)
      throw (CORBA::SystemException,
             Domain_Repository::Unknown_Repository_Item_Ids);

    virtual void private_detach_observer (
      Observer_ptr obs,
      const Domain_Repository::Repository_Item_Ids& observed_id_list,
      CORBA::Environment &IT_env=CORBA::default_environment)
      throw (CORBA::SystemException,
             Domain_Repository::Unknown_Repository_Item_Ids,
             Domain_Repository::Unknown_Observer,
             Domain_Repository::Unmatched_Observer_Item_Ids);

    virtual Domain_Repository::Repository_Item_Ids get_item_ids_list(
      Observer_ptr obs);

    virtual Observers* get_item_observers (
      Domain_Repository::Repository_Item_Id item_id_p,
      CORBA::Environment &IT_env=CORBA::default_environment)
      throw (CORBA::SystemException);

    virtual Observer_List *get_item_observer_list (
      Domain_Repository::Repository_Item_Id item_id_p);

    virtual void insert_new_item(
      Domain_Repository::Repository_Item_Id id_p);

    virtual void detach_item_from_observers(
      Domain_Repository::Repository_Item_Id id_p);

  protected:

    // Calculate the hash value of the observer pointer.
    static unsigned
      hash_observer(const Observer_ptr& observer_p);

    // Calculate the hash value of the item id.
    static unsigned hash_item_id(
      const Domain_Repository::Repository_Item_Id& item_id_p);

    void add_proxy(Observer_ptr &obs);
    virtual void remove_proxy(Observer_ptr obs);

    typedef RWTValHashTable<Domain_Repository::Repository_Item_Id>
      Item_Id_List;
    typedef RWTValHashTableIterator<Domain_Repository::Repository_Item_Id>
      Item_Id_List_Iterator;
    typedef RWTValHashDictionary <Observer_ptr, Item_Id_List*>
      Observer_Item_Dictionary;
```

```
  typedef RWTValHashDictionaryIterator <Observer_ptr, Item_Id_List*>
    Observer_Item_Dictionary_Iterator;
  typedef RWTValHashDictionary<Domain_Repository::Repository_Item_Id,
    Observer_List*> Item_Observer_Dictionary;
  typedef RWTValHashDictionaryIterator<Domain_Repository::Repository_Item_Id,
    Observer_List*> Item_Observer_Dictionary_Iterator;

  // Mapping of observers to items.
  Observer_Item_Dictionary observers_to_item_ids_list;
  // Mapping of items to observers
  Item_Observer_Dictionary item_ids_to_observers_list;

#ifdef ORBIX
  typedef RWTValHashTable<SDO_ptr> Proxy_List;
  typedef RWTValHashTableIterator<SDO_ptr> Proxy_List_Iterator;
  Proxy_List proxy_table;
#endif
};
```

4.3.2 Template Parameters

The parameter of the *Basic_Observed* template is a class that should contain typedef definitions of the following types:

- *Observed* – type of implemented **_Observed_R* interface,
- *Observer* – type of the **_Observer* interface,
- *Observers* – **_Observer* sequence type,
- *Observer_ptr* – type of the **_Observer* reference type.

Below a sample definition of the template parameter for the *Static_Data_Observed_R* is given.

```
class SDO_Types {

public:

  typedef Domain_Repository::Static_Data_Observed_R Observed;
  typedef Domain_Repository::Static_Data_Observer Observer;
  typedef Domain_Repository::Static_Data_Observer_ptr Observer_ptr;
  typedef Domain_Repository::Static_Data_Observed_E::Static_Data_Observers
    Observers;
};
```

4.3.3 Attribute Description

The attribute of the *Basic_Observed_I* template class are:

- *observers_to_item_ids_list* – dictionary which maps observer references into items (items) monitored by these observers,

- *item_ids_to_observers_list* – dictionary which maps items to observers which monitor these items,

- *proxy_table* – a table of reference wrappers which hold references of observers.

All the above attributes constitute the registry of observers monitoring updating and removing items.

4.3.4 Method Description

The first three methods correspond to operations of the IDL interfaces:

- *observer_list* – gets a list of all observer references associated with identifiers of the monitored items,

- *attach_observer* – registers the given observer with items with the given identifiers,

- *detach_observer* – deregisters the given observer from items with the given identifiers.

Each of the above methods locks the observer registry at the beginning and unlocks it at the end.

The next two methods: *private_attach_observer* and *private_detach_observer* attach and detach an observer without setting the lock. They are assumed to be called locally, mainly from *attach_observer* and *detach_observer*.

The methods: *get_item_ids_list*, *get_item_observers*, *get_item_observer_list* retrieve information from the repository registry. The first of them returns a sequence of identifiers of items observed by the given observer. The two last return a sequence or a list of references of observers monitoring the given item.

The last two methods: *insert_new_item* and *detach_item_from_observers* are assumed to be invoked when an item is inserted or removed from the repository. The first method inserts an identifier of the new item to the registry and associates it with the empty list of observers. The latter method removes an item identifier from the registry and detaches it from all observers.

4.4 Observed_I

The *Observed_I* C++ template class extends *Basic_Observed_I* and maintains the registry of observers monitoring adding new items.

4.4.1 Class Definition

```
#include "abng_guarded.hh"
#include "basic_observed_i.h"

template <class Observer_Types>
class Observed_I : public virtual Basic_Observed_I<Observer_Types> {
```

```
public:

  Observed_I();
  virtual ~Observed_I();

  virtual Basic_Observed_I<Observer_Types>::Observers* add_observer_list (
    CORBA::Environment &IT_env=CORBA::default_environment)
    throw (CORBA::SystemException);

  virtual void attach_add_observer (
    Observer_ptr obs,
    CORBA::Environment &IT_env=CORBA::default_environment)
    throw (CORBA::SystemException);

  virtual void detach_add_observer (
    Observer_ptr obs,
    CORBA::Environment &IT_env=CORBA::default_environment)
    throw (CORBA::SystemException,
           Domain_Repository::Unknown_Observer);

  virtual void private_attach_add_observer (
    Observer_ptr obs,
    CORBA::Environment &IT_env=CORBA::default_environment)
    throw (CORBA::SystemException);

  virtual void private_detach_add_observer (
    Observer_ptr obs,
    CORBA::Environment &IT_env=CORBA::default_environment)
    throw (CORBA::SystemException,
           Domain_Repository::Unknown_Observer);

protected:

  virtual void remove_proxy(Observer_ptr obs);

  // list of add observers
  Observer_List my_add_observer_list;

};
```

4.4.2 Template Parameters

The parameters of the *Observed_I* template should fulfill the same requirements as for the *Basic_Observer_I* template.

4.4.3 Attribute Description

The attribute of the *Observed_I* template class is:

- *my_add_observer_list* – a list of references of observers who want to be notified when a new item is added to the repository.

4.4.4 Method Description

The first three methods correspond to operations of the **Observed_R* IDL interfaces:

- *add_observer_list* – gets a list of references of observers which monitor adding new items,
- *attach_add_observer* – adds a new observer monitoring additions,
- *detach_add_observer* – removes an observer monitoring additions.

Each of the above methods locks the registry of addition observers at the beginning and unlocks it at the end.

The next two methods: *private_attach_add_observer* and *private_detach_add_observer* attach and detach an addition observer without setting the lock. They are assumed to be called locally, mainly from *attach_add_observer* and *detach_add_observer*.

4.5 Basic_Rep_I

The main component of the Repository Template is the *Basic_Rep_I* template C++ class which provides patterns for implementation of IDL operations of most repository interfaces.

4.5.1 Class Definition

```
#include "observed_i.h"
#include "db_utils.hh"
#include <rw/tphasht.h>

template <class Item_Types, class Observer_Types>
class Basic_Rep_I : public Observed_I<Observer_Types> {

public:

  typedef Item_Types::Item Item;
  typedef Item_Types::Item_ptr Item_ptr;
  typedef Item_Types::Items Items;
  typedef Item_Types::Item_I Item_I;
  typedef Item_Types::Initial_Data Initial_Data;
  typedef Item_Types::Data_Object Data_Object;
  typedef Item_Types::Item::Description Description;
  typedef Item_Types::Item::Pattern Pattern;
  typedef Item_Types::Item::Value_Description Value_Description;
  typedef Item_Types::Item::Value_Descriptions Value_Descriptions;

  typedef Basic_Observed_I<Observer_Types>::Observer_List Observer_List;

  Basic_Rep_I(unsigned (*hash_item_p)(const Item_I&),
#ifdef ObjectStore
    os_database* db_p
#elif FilePersistency
```

```
    const char *repository_name_p
#endif
  );

  virtual ~Basic_Rep_I();

  virtual void init(int cold_start_p = 0);

  virtual Items* item_list (
    CORBA::Environment &IT_env=CORBA::default_environment);

  virtual Items* find(
    const Pattern& data_record_pattern,
    CORBA::Environment &IT_env=CORBA::default_environment);

 virtual Value_Descriptions* find_values (
   const Pattern& data_record_pattern,
   CORBA::Environment &IT_env=CORBA::default_environment);

  virtual Item_ptr add (
    const Initial_Data& data_record,
    CORBA::Environment &IT_env=CORBA::default_environment);

  virtual void update (
    Item_ptr object_to_update,
    const Description& data_record_pattern,
    CORBA::Environment &IT_env=CORBA::default_environment);

  virtual void remove (
    Item_ptr object_to_remove,
    CORBA::Environment &IT_env=CORBA::default_environment);

  virtual Items* private_find(
    const Pattern& data_record_pattern,
    CORBA::Environment &IT_env=CORBA::default_environment) = 0;

 virtual Value_Descriptions* private_find_values (
   const Pattern& data_record_pattern,
   CORBA::Environment &IT_env=CORBA::default_environment);

  virtual Item_ptr private_add (
    const Initial_Data& data_record,
    CORBA::Environment &IT_env=CORBA::default_environment);

  virtual void private_update (
    Item_ptr object_to_update,
    const Description& data_record_pattern,
    CORBA::Environment &IT_env=CORBA::default_environment);

  virtual void private_remove (
    Item_ptr object_to_remove,
    CORBA::Environment &IT_env=CORBA::default_environment);
```

```
protected:

    // Add helper methods
    virtual int validate_add_data(const Initial_Data& data_record) {return 1;}
    virtual Item_I *create_item(const Initial_Data& data_record);
    virtual int add_permitted(Item_I* item_p);
    virtual void init_corba_object(Item_I *item_p) = 0;
    virtual void insert_item(Item_I* item_p);

    // Remove helper methods
    virtual int validate_remove_data(Item_ptr item_p);
    virtual int remove_permitted(Item_I* item_p);
    virtual void uninit_corba_object(Item_I* item_p);
    virtual void remove_item(Item_I* item_p);

    // Update helper methods
    virtual int validate_update_data(Item_ptr item_p,
        const Description& data_record_pattern);
    virtual int update_permitted(Item_I* item_p,
        const Description& data_record_pattern) {return 1;}
    virtual void update_item(Item_I* item_p,
        const Description& data_record_pattern);

    // Find helper methods
    virtual void collect_values(Item_I *item_p, Value_Description& vd) = 0;

    // Common helper methods
    virtual Item_I *deref_item(Item_ptr item_p);
    virtual void construct_data_object(Data_Object& data_object_p,
        Item_ptr item_p) = 0;
    virtual void notify_observers(
        Observer_List& obs_list,
        Item_I *item_p,
        Domain_Repository::Change_Type type_p);
    virtual void save();
    virtual void save(ostream &);
    virtual void restore (istream &);
    virtual Domain_Repository::Repository_Item_Id get_item_counter() {
        return item_counter;
    }
    virtual Domain_Repository::Repository_Item_Id assign_id() {
        return item_counter++;
    }
    virtual DB_Header *create_db_header() = 0;
    static void gather_item(Item_I* current_item, void* apply_data);

    static void write_ior(const CORBA::Object_ptr obj_p,
        const char *file_name);

    virtual void create_corba_objects() = 0;
    virtual void destroy_corba_objects() = 0;
```

```
  // Attributes:

  // internal item id counter of the repository
  Domain_Repository::Repository_Item_Id item_counter;

  typedef RWTPtrHashTable<Item_I> Item_List;
  typedef RWTPtrHashTableIterator<Item_I> Item_List_Iterator;
  typedef RWTValHashDictionary<Domain_Repository::Repository_Item_Id,
    Item_I*> Id_Item_Dictionary;
  typedef RWTValHashDictionaryIterator<Domain_Repository::Repository_Item_Id,
    Item_I*> Id_Item_Dictionary_Iterator;

  // list of items
  Item_List my_item_list;
  Id_Item_Dictionary my_item_deref;

  unsigned (*hash_item)(const Item_I&);

#ifdef ObjectStore
  os_database *my_db;
#elif defined FilePersistency
  char* my_repository_name;
#endif
};
```

4.5.2 Template Parameters

The parameter of the *Basic_Rep_I* template are the *Item_Types* and *Observer_Types* classes. The second one is passed as the parameter for *Observer_I* template which is the base of *Basic_Rep_I*. The first parameter should denote a class containing the following typedef definitions:

- *Item* – the IDL interface provided by items stored in the repository,
- *Item_ptr* – item reference type,
- *Items* – a sequence of item references,
- *Item_I* – item implementation class,
- *Initial_Data* – a data type used for item object initialisation,
- *Data_Object* – a type of data which are send to repository observers after a change.

Below a sample definition of the template parameter for the *Sensor_Rep* repository is given.

```
class Sensor_Rep_Types {

public:
        typedef ABng_Core_System::Sensor Item;
        typedef ABng_Core_System::Sensor_ptr Item_ptr;
```

```
        typedef ABng_Core_Management::Sensor_Rep::Sensors Items;
        typedef Sensor_I Item_I;
        typedef ABng_Core_System::Sensor::Description Initial_Data;
        typedef ABng_Core_System::System_Static_Data_Observer::
                System_Static_Data_Object Data_Object;
};
```

4.5.3 Attribute Description

The attribute of the *Basic_Rep_I* template class are:

- *item_counter* – a counter of items which have been inserted into the repository. It's value is incremented each time a new item is inserted and it becomes an identifier of the item.
- *my_item_list* – a list of item implementation objects.
- *my_item_deref* – a dictionary which maps references of item CORBA objects into their implementations.
- *hash_item* – a function used to hash items.
- my_repository_name – the name of the file in which a repository persistent image is stored.

4.5.4 Method Description

If *cold_start* is non-zero the *init* method creates a file which will be used to store a persistent image of the repository contents. Otherwise the method reads the file and reconstructs the repository objects invoking *restore*.

The next seven methods implement IDL operations of every repository. Each of them locks repository attributes, calls a corresponding method starting with *private_* and unsets the lock. Hence, each *private_* method constitutes a default implementation of a corresponding IDL operation. This default implementation can be overwritten in a derived class, which usually implements a concrete repository. This can be done in two ways: either the whole method implementing an IDL operation can be redefined or some of the methods which are declared in the protected part and which are called by *private_* methods can be overwritten.

- *private_find* – this is a pure virtual function which has to be defined in a derived class. The reason for this is that an algorithm of repository searching is strongly dependent on an item definition.
- *private_find_values* – first it calls *private_find* which results in a sequence of items which satisfy a given pattern. Next this method allocates a sequence of value descriptions, each of which corresponds to a found item. Finally, every value description is initiated by an invocation of *collect_values*, which is repository-specific and has to be defined in a derived class, and the sequence is returned to a caller.
- *private_add* – its code is as follows.

```
template <class Item_Types, class Observer_Types>
Basic_Rep_I<Item_Types, Observer_Types>::Item_ptr
  Basic_Rep_I<Item_Types, Observer_Types>::private_add (
  const Basic_Rep_I<Item_Types, Observer_Types>::Initial_Data& data_record,
  CORBA::Environment &IT_env) {

  validate_add_data(data_record);
  Item_I *new_item = create_item(data_record);
  add_permitted(new_item);
  init_corba_object(new_item);
  // assign a unique item id to it
  new_item->set_item_id(assign_id());
  insert_item(new_item);
  notify_observers(my_add_observer_list, new_item, Domain_Repository::OBJECT_ADDED);
#ifdef FilePersistency
  save();
#endif
  return Item::_duplicate(new_item->get_corba_object());
}
```

The method subsequently calls:

- *validate_add_data* – always returns a non-zero, can be overwritten in a derived class to check whether initial data for a new item are correct.
- *create_item* – creates an instance of the *Item_I* class assuming that it provides a constructor which takes an argument of the *Initial_Data* type.
- *add_ppermitted* – checks if an item has not been already added to a repository.
- *init_corba_object* – initialise a CORBA object associated with a new item.
- *new_item− >set_item_id* – assigns an identifier of a new item.
- *insert_item* – inserts a new item into the item list and the item identifier into the observer registry.
- *notify_observers* – notifies observers of item additions.
- *save* – saves all the repository contents to the file.

The *private_add* method returns a reference of the CORBA object associated with a newly created item.

- *private_update* – its code is as follows.

```
template <class Item_Types, class Observer_Types>
void Basic_Rep_I<Item_Types, Observer_Types>::private_update(
  Basic_Rep_I<Item_Types, Observer_Types>::Item_ptr object_to_update,
  const Basic_Rep_I<Item_Types, Observer_Types>::Description&
    data_record_pattern,
  CORBA::Environment &IT_env) {

  if(!validate_update_data(object_to_update, data_record_pattern)) {
```

```
    return;
  }
  Item_I *item = deref_item(object_to_update);
  if(!update_permitted(item, data_record_pattern)) {
    return;
  }
  my_item_list.remove(item);
  update_item(item, data_record_pattern);
  my_item_list.insert(item);
  Observer_List *observer_list_p =
    get_item_observer_list(item->item_id());
  notify_observers(*observer_list_p, item, Domain_Repository::OBJECT_UPDATED);
#ifdef FilePersistency
  // store updated state of the repository
  save();
#endif
}
```

The method subsequently calls:

- *validate_update_data* – throws an exception if a reference of the CORBA object associated with the item to be updated is nil or remote, otherwise it returns a non-zero.
- *deref_item* – maps a reference of the CORBA object to its implementation.
- *my_item_list.remove* – removes the item form the item list.
- *update_item* – set the new values of item's attributes.
- *my_item_list.insert* – inserts the updated item.
- *get_item_observer_list* – retrieves a list of observer monitoring the item.
- *notify_observers* – notifies item observers about its update.
- *save* – saves all the repository contents to the file.

• *private_remove* – its code is as follows.

```
template <class Item_Types, class Observer_Types>
void Basic_Rep_I<Item_Types, Observer_Types>::private_remove (
  Basic_Rep_I<Item_Types, Observer_Types>::Item_ptr object_to_remove,
  CORBA::Environment &IT_env) {

  if(!validate_remove_data(object_to_remove)) {
    return;
  }
  Item_I *item = deref_item(object_to_remove);
  if(!remove_permitted(item)) {
    return;
  }
  Observer_List *observer_list_p =
    get_item_observer_list(item->item_id());
  notify_observers(*observer_list_p, item, Domain_Repository::OBJECT_REMOVED);
  remove_item(item);
```

```
    uninit_corba_object(item);
    delete item;
#ifdef FilePersistency
    // store updated state of the repository
    save();
#endif
}
```

The method subsequently calls:

- *validate_remove_data* – throws an exception if a reference of the CORBA object associated with the item to be removed is nil or remote, otherwise it returns a non-zero.
- *deref_item* – maps a reference of the CORBA object to its implementation.
- *get_item_observer_list* – retrieves a list of observers monitoring the item.
- *notify_observers* – notifies item observers about its removal.
- *remove_item* – removes the item from the item list and the item identifier from the observer registry.
- *uninit_corba_object* – releases the CORBA object associated with the item.
- *save* – saves all the repository contents to the file.

4.6 Basic_Server

The *Basic_Server* C++ class provides a basis for implementation of ABng CORBA servers. Its role is to provide server's components with references of ORB and BOA. It also implements a method which sets default ABng configuration of CORBA (Orbix) servers.

4.6.1 Class Definition

```
#include <CORBA.h>

class Basic_Server {

public:

  Basic_Server() : my_timeout(CORBA::BOA::INFINITE_TIMEOUT) {}
  virtual ~Basic_Server() {}

  virtual void init_corba(int argc, char **argv);

#ifdef ORBIX
  virtual void set_orbix_conf(
    unsigned long max_retries_p = (CORBA::ULong) 2,              // Orbix default is 10
    unsigned long connection_timeout_p = (CORBA::ULong) 30,
    unsigned long default_tx_timeout_p = CORBA::BOA::INFINITE_TIMEOUT,
    unsigned long object_table_size_p = CORBA::_OBJECT_TABLE_SIZE_DEFAULT,
    int diagnostics_p = 0,
```

```
    int propagate_tie_detele_p = 0,
    int abort_slow_connection_p = 0);
#endif

  virtual CORBA::ORB_ptr get_orb() {return orb_ptr;}
  virtual CORBA::BOA_ptr get_boa() {return boa_ptr;}
  virtual void run() {
#ifdef ORBIX
    boa_ptr->processEvents(my_timeout);
#endif
  }

protected:
    CORBA::ORB_ptr orb_ptr;
    CORBA::BOA_ptr boa_ptr;
#ifdef ORBIX
  long my_timeout;
#endif
};
```

4.6.2 Attribute Description

- *orb_ptr* – an ORB reference,
- *boa_ptr* – a BOA reference,
- *my_timeout* – timeout for Orbix servers.

4.6.3 Method Description

- *init_corba* – initialises ORB and BOA references,
- *set_orbix_conf* – sets the default ABng configuration of CORBA servers,
- *run* – starts CORBA processing loop.

4.7 Rep_Server

The *Rep_Server* class provides a basis for implementation of a server containing a single ABng repository. Its main role is to simplify the construction of a server mainline. Due to this class the main function of a server repository consists of subsequent invocations of the *init*, *run* and *final* methods.

4.7.1 Class Definition

```
#include "basic_server.h"

#ifdef ObjectStore
```

```
#include <oosa.h>
#endif                                                  /* ObjectStore */

class Rep_Server : public Basic_Server {

public:

  Rep_Server(
#ifdef ObjectStore
    const char *db_name_p,                              // default db name
#elif FilePersistency
    const char *file_name_p,                            // default db name
#endif
    const char *current_directory_p = ".");

  virtual ~Rep_Server();

  virtual void init(int argc, char **argv);
  virtual void final();
  virtual void get_options(int argc, char **argv);
  virtual void check_directory(const char *dir_name_p);
  virtual char *concat_path_name(const char *dir_name_p,
    const char *file_name_p);
  virtual void print_info();

#ifdef ObjectStore
  virtual os_database * init_os();
#endif                                                  /* ObjectStore */
  virtual void create_objects() = 0;
  virtual void delete_objects() = 0;

protected:

#ifdef ObjectStore
  char *my_db_name;
#elif FilePersistency
  char *file_name;
#endif                                                  /* ObjectStore */
  char *my_current_directory;
  int cold_start;
  char *prog_name;
};
```

4.7.2 Method Description

Two methods invoked in *main* of a repository server are:

- *init* – first it initialises ORB and BOA references and reads command line options passed in arguments. The options are:
  - *-d directory* – specify a directory containing a repository database file,

– *-s* – creates a new database file, used when the repository is started for the first time,

– *-t timeout* – specify timeout between Orbix events.

Afterwards, *init* sets Orbix configuration using *set_orbix_conf* and instantiates a repository using *create_objects* which has to be defined in a derived class for a concrete repository server.

- *final* – delete a repository by calling *delete_objects* which has to be defined in a derived class.

4.8 Source Code Organisation

The Repository Template sources are placed in the following files:

- *common/DO_ptr.h*, *common/DO_ptr.cc* – the *DO_ptr* wrapper template class,
- *common/db_header.h*, *common/db_header.cc* – the *DB_Header* class,
- *common/basic_observed_i.h*, *common/basic_observed_i.cc* – the *Basic_Observed_I* template class,
- *common/observed_i.h*, *common/observed_i.cc* – the *Observed_I* template class,
- *common/basic_rep_i.h*, *common/basic_rep_i.cc* – the *Basic_Rep_I* template class,
- *common/basic_server.h*, *common/basic_server.cc* – the *Basic_Server* class,
- *common/rep_server.h*, *common/rep_server.cc* – the *Rep_Server* class,

To use template C++ classes in an application, template header files must be included into the application's code. The C++ SPARC Compiler takes care about the instantiation and the linking of template methods definitions. The non-template classes are linked into the *libncommon.a* library, which should be linked to the application.

utils octets ecords ctets tatic ystem tem ds ist ew etach rom dd ypes bject ounter eref unsets ind alues cc orba pdate deref emove uninit onf bjects imeout LocalWords

Chapter 5

Badge Repository

5.1 Functional Specification

5.1.1 Functionality description

Badge is an entity that defines type of a badge e.g. its identifier. *Badge Repository* constitutes a database that stores *Badge* entities. It supports operations that allow searching and modifying its contents.

5.1.2 Entity

5.1.2.1 Interface description

```
#ifndef __BADGE_IDL__
#define __BADGE_IDL__ 1

interface Badge: System_Static_Data_Observed_E, Badge_Commander {

    struct Description {
        Domain_Repository::Repository_Item_Id item_id;
        Badge_Id id;
        Badge_Holder holder;
    };

    typedef sequence<Description> Descriptions;

    typedef Description Value_Description;
    typedef sequence<Value_Description> Value_Descriptions;

    struct Pattern {
        boolean is_any;
        Domain_Repository::Repository_Item_Id_Pattern item_id_pattern;
        Badge_Id_Pattern id_pattern;
    };

    Description descr()
```

```
        raises (Domain_Repository::Destination_Unreachable);
    Value_Description value_descr()
        raises (Domain_Repository::Destination_Unreachable);
    readonly attribute Domain_Repository::Repository_Item_Id item_id;
    readonly attribute Badge_Id id;
    Badge_Holder holder()
        raises (Domain_Repository::Destination_Unreachable);
};

#endif                                                  /* __BADGE_IDL__ */
```

The *Description* structure of the *Badge* entity contains:

- *item_id* – Repository Item Id of the entity,
- *id* – identifier of a badge,
- *badge_holder* – a reference to corresponding *Badge Holder* object.

Value_Description structure is equivalent to *Description*.
A *Badge* entity supports the following attributes:

- *descr* – get description of a given entity,
- *value_descr* – get value description of a given entity; in *Badge* it is equivalent to *descr*,
- *item_id* – get Repository Item Id of the entity,
- *id* – get an identifier of a badge,
- *badge_holder* – get a reference to corresponding *Badge Holder* object.

5.1.2.2 Equality of entities

Two entties are considered equal if their identifiers are equal. The repository may not store any two entities that are equal to one another.

5.1.3 Repository

5.1.3.1 Interface description

```
#ifndef __BADGE_REP_IDL__
#define __BADGE_REP_IDL__ 1

interface Badge_Rep : ABng_Core_System::System_Static_Data_Observed_R {

    readonly attribute ABng_Core_System::Badges badge_list;

    ABng_Core_System::Badge add(
        in ABng_Core_System::Badge::Description data_record)
        raises(Domain_Repository::Duplicate_Data_Record,
```

```
                 Domain_Repository::Destination_Unreachable);
      void remove(in ABng_Core_System::Badge object_to_remove)
            raises(Domain_Repository::Unknown_Object_Ref);
      ABng_Core_System::Badges find(
            in ABng_Core_System::Badge::Pattern data_record_pattern)
            raises(Domain_Repository::Search_Not_Implemented);
      ABng_Core_System::Badge::Value_Descriptions find_values(
            in ABng_Core_System::Badge::Pattern data_record_pattern)
            raises(Domain_Repository::Search_Not_Implemented);
      void update(in ABng_Core_System::Badge object_to_update,
            in ABng_Core_System::Badge::Description data_record_pattern)
            raises(Domain_Repository::Unknown_Object_Ref,
                  Domain_Repository::Duplicate_Data_Record,
                  Domain_Repository::Destination_Unreachable);
};

#endif                                                  /* __BADGE_REP_IDL__ */
```

5.1.3.2 Relationships

The *Badge Repository* is related to *Badge Holder Repository*. A badge does not have to be associated to a badge holder, but badge holders must be linked to badges.

The *Badge Repository* register itself as an observer in *Scheduler*, as it contains *Badge_Commander* interface that forwards requests to *Scheduler*. It uses also *Sensor Repository* while submitting commands to *Scheduler*, but it is not registered as an observer.

5.1.4 Implementation classes

5.1.4.1 Badge_I

The class implements interfaces of *Badge*, *Badge_Commander* and *System_Static_Data_Observed_E*.

5.1.4.2 Badge_Rep_I

The class uses the building blocks of the repository template to implement functionality of *Badge_Rep* and *System_Static_Data_Observed_R* interfaces. The class inherits from *Basic_Rep_I<Badge_Types, SDO_Types>*, where the parameter classes are defined as follows:

```
class SDO_Types {

public:

   typedef Domain_Repository::Static_Data_Observed_R Observed;
   typedef Domain_Repository::Static_Data_Observer Observer;
   typedef Domain_Repository::Static_Data_Observer_ptr Observer_ptr;
   typedef Domain_Repository::Static_Data_Observed_E::Static_Data_Observers
      Observers;
};
```

```
class LT_Types {

public:

   typedef Domain_Repository::Location_Type Item;
   typedef Domain_Repository::Location_Type_ptr Item_ptr;
   typedef Domain_Repository::Location_Type_Rep::Location_Types Items;
   typedef Location_Type_I Item_I;
   typedef Domain_Repository::Location_Type::Description Initial_Data;
   typedef Domain_Repository::Static_Data_Observer::Static_Data_Object
      Data_Object;
};
```

The class overwrites the following template methods:

- *private_find* – invokes the function matching entities with the specified pattern.

5.1.4.3 Badge_Rep_Server

This is the repository server implementation class based on the template of the repository server.

- *create_objects* – creates and initializes an instance of the *Badge_Rep_I* class.
- *delete_objects* – deletes the instance of the *Badge_Rep_I* class.

5.1.5 Persistency

Currently, the persistent state of the repository is stored in files. Since the *Location Type Repository* does not observe other repositories, the start-up and revoking the saved state are stright-forward.

5.1.6 CORBA services usage

Current implementation uses only naming service.

5.1.7 Life cycle management

The repository stores its reference in the naming service using ABng Server Manager. Currently it must be launched by hand.

5.2 Implementation Specification

5.2.1 Source code organization

- *badge_rep/badge_i.h, badge_rep/badge_i.cc* – definition of the *Badge_I* class,

- *badge_rep/badge_rep_i.h, badge_rep/badge_rep_i.cc* – definition of the *Badge_Rep_I* class,
- *badge_rep/badge_rep_server.h, badge_rep/badge_rep_server.cc* – definition of the *Badge_Rep_Server* class,
- *badge_rep/abng_db_srv.cc* – definition of the main function.

Chapter 6

Badge Holder Repository

6.1 Functional Specification

6.1.1 Functionality description

Badge Holder is an entity that binds a user or an equipment with a badge. The *Badge Holder Repository* constitutes a database that stores *Badge Holder* entities.

6.1.2 Entity

6.1.2.1 Interface description

```
#ifndef __BADGE_HOLDER_IDL__
#define __BADGE_HOLDER_IDL__ 1

interface Badge_Holder : System_Static_Data_Observed_E {

    struct Description {
        Domain_Repository::Repository_Item_Id item_id;
        Badge my_badge;
        Badge current_badge;
        Sensor home_sensor;
        ABng_Core::Locatable_Type_Specific_Object type_specific_object;
    };

    typedef sequence<Description> Descriptions;

    struct Value_Description {
        Domain_Repository::Repository_Item_Id item_id;
        Badge::Value_Description my_badge;
        Badge::Value_Description current_badge;
        Sensor::Value_Description home_sensor;
        ABng_Core::Locatable_Type_Specific_Object_Value type_specific_object;
    };

    typedef sequence<Value_Description> Value_Descriptions;
```

```
    struct Pattern {
        boolean is_any;
        Domain_Repository::Repository_Item_Id_Pattern item_id_pattern;
        Badge::Pattern my_badge_pattern;
        Badge::Pattern current_badge_pattern;
        Sensor::Pattern home_sensor_pattern;
        ABng_Core::Locatable_Type_Specific_Object_Pattern
            type_specific_object_pattern;
    };

    Description descr()
        raises (Domain_Repository::Destination_Unreachable);
    Value_Description value_descr()
        raises (Domain_Repository::Destination_Unreachable);
    readonly attribute Domain_Repository::Repository_Item_Id item_id;
    Badge my_badge()
        raises (Domain_Repository::Destination_Unreachable);
    Badge current_badge()
        raises (Domain_Repository::Destination_Unreachable);
    Sensor home_sensor()
        raises (Domain_Repository::Destination_Unreachable);
    ABng_Core::Locatable_Type_Specific_Object type_specific_object()
        raises (Domain_Repository::Destination_Unreachable);

};

#endif                                          /* __BADGE_HOLDER_IDL__ */
```

The *Description* structure of the *Badge Holder* entity contains:

- *item_id* – Repository Item Id of the entity,
- *my_badge* – a reference to the badge which is owned by this user or equipment,
- *current_badge* – a reference of the badge currently hold by this user or equipment,
- *home_sensor* – a reference of the sensor, which is nearest to the user usual work place or which is nearest to the usual installation of the equipment,
- *type_specific_object* – a union which store either reference to a user entity or to an equipment entity.

Value_Description structure is composed of:

- *item_id* – Repository Item Id of the entity,
- *my_badge* – a value description of the badge which is owned by this user or equipment,
- *current_badge* – a value description of the badge currently hold by this user or equipment,
- *home_sensor* – a value description of the home sensor,

- *type_specific_object* – a union storing a value description of either a user or an equipment.

A *Badge Holder* entity supports the following attributes and operations:

- *descr* – get description of a given entity,
- *value_descr* – get value description of a given entity; this method necessitates obtaining value descriptions of the *Location Description* object and all *Sensor* objects corresponding to the given entity from the appropriate repositories; if any of these repositories cannot be contacted the exception is thrown,
- *item_id* – get Repository Item Id of the entity,
- *my_badge* – get a reference of the badge which is owned by this user or equipment,
- *current_badge* – get a reference of the badge currently hold by this user or equipment,
- *home_sensor* – get a reference of the home sensor,
- *type_specific_object* – get a union storing either reference to a user entity or to an equipment entity.

6.1.2.2 Equality of entities

Two entities are considered equal if they have to the same *current_badge*, i.e. repository item ids of their corresponding *Badge* objects are equal.

6.1.3 Repository

6.1.3.1 Interface description

```
#ifndef __BADGE_HOLDER_REP_IDL__
#define __BADGE_HOLDER_REP_IDL__ 1

interface Badge_Holder_Rep : ABng_Core_System::System_Static_Data_Observed_R {

    typedef sequence<ABng_Core_System::Badge_Holder> Badge_Holders;

    readonly attribute Badge_Holders badge_holder_list;

    ABng_Core_System:: Badge_Holder add(
        in ABng_Core_System::Badge_Holder::Description data_record)
        raises(Domain_Repository::Duplicate_Data_Record,
               Domain_Repository::Destination_Unreachable);
    void remove(in ABng_Core_System::Badge_Holder object_to_remove)
        raises(Domain_Repository::Unknown_Object_Ref);
    Badge_Holders find(
        in ABng_Core_System::Badge_Holder::Pattern data_record_pattern)
        raises(Domain_Repository::Search_Not_Implemented,
               Domain_Repository::Destination_Unreachable);
```

```
    ABng_Core_System::Badge_Holder::Value_Descriptions find_values(
        in ABng_Core_System::Badge_Holder::Pattern data_record_pattern)
        raises(Domain_Repository::Destination_Unreachable,
               Domain_Repository::Search_Not_Implemented);
    void update(in ABng_Core_System::Badge_Holder object_to_update,
        in ABng_Core_System::Badge_Holder::Description data_record_pattern)
        raises(Domain_Repository::Unknown_Object_Ref,
               Domain_Repository::Duplicate_Data_Record,
               Domain_Repository::Destination_Unreachable);
};

#endif                                        /* __BADGE_HOLDER_REP_IDL__ */
```

6.1.3.2 Relationships

The *Badge Holder Repository* observes the *Badge Repository*, *Sensor Repository*, *User Repository* and *Equipment Repository*. It attaches itself as an observer to:

- all badges which are either current or owned by any badge holder,
- all sensors which are home sensor for any badge holder,
- all users which describe any badge holder,
- all equipments which describe any badge holder.

6.1.4 Implementation classes

6.1.4.1 Badge_Holder_I

The class implements interfaces of *Badge_Holder* and *System_Static_Data_Observed_E*.

6.1.4.2 Badge_Holder_Rep_I

The class uses the building blocks of the repository template to implement functionality of *Badge_Holder_Rep* and *System_Static_Data_Observed_R* interfaces. It inherits from the *Basic_Rep_I<ALD_Types, SSDO_Types>* class, where the template parameter classes are defined as follows:

```
class SDO_Types {

public:
        typedef ABng_Core_System::System_Static_Data_Observed_R Observed;
        typedef ABng_Core_System::System_Static_Data_Observer Observer;
        typedef ABng_Core_System::System_Static_Data_Observer_ptr Observer_ptr;
        typedef ABng_Core_System::System_Static_Data_Observed_E::
               System_Static_Data_Observers Observers;
};

class LT_Types {

public:
```

```
        typedef ABng_Core_System::Badge_Holder Item;
        typedef ABng_Core_System::Badge_Holder_ptr Item_ptr;
        typedef ABng_Core_Management::Badge_Holder_Rep::Badge_Holders Items;
        typedef Badge_Holder_I Item_I;
        typedef ABng_Core_System::Badge_Holder::Description Initial_Data;
        typedef ABng_Core_System::Badge_Holder::Description Description;
        typedef ABng_Core_System::System_Static_Data_Observer::
                System_Static_Data_Object Data_Object;
};
```

The class overwrites the following template methods:

- *private_find* – invokes the function matching entities with the specified pattern. Depending on a pattern this method may be forced to cooperate with the *Badge Repository*, *Sensor Repository*, *User Repository* and *Equipment Repository*. However, as much is checked locally as possible. Moreover, a cache local to the matching method has been applied in order to remove recurrence of remote finds with the same pattern.

6.1.4.3 Badge_Rep_Observer_I

This class implements the functionality of the *Static_Data_Observer* for a *Badge Repository* observer. It reacts to the change of the observed repository in the following way:

1. if a *Badge* object has been removed then:
    - if this is a current badge of some badge holder then it removes this badge holder entity from the *Badge Holder Repository*,
    - if this is a my badge of some badge holder then it invalidates appropriate fields in this badge holder.
2. if a *Badge* object has been updated it does not react.

6.1.4.4 Sensor_Rep_Observer_I

This class implements the functionality of the *System_Static_Data_Observer* for a *Sensor Repository* observer. It reacts to the change of the observed repository in the following way:

1. if a *Sensor* object has been removed it removes its reference from the appropriate badge holder,
2. if a *Sensor* object has been updated it does not react.

6.1.4.5 User_Rep_Observer_I

This class implements the functionality of the *System_Static_Data_Observer* for a *User Repository* observer. It reacts to the change of the observed repository in the following way:

1. if some *User* object has been removed it badge holder entity from the *Badge Holder Repository*, which was described by the removed user,
2. if a *User* object has been updated it does not react.

6.1.4.6 Equipment_Rep_Observer_I

This class implements the functionality of the *System_Static_Data_Observer* for a *Equipment Repository* observer. It reacts to the change of the observed repository in the following way:

1. if some *Equipment* object has been removed it badge holder entity from the *Badge Holder Repository*, which was described by the removed equipment,

2. if a *Equipment* object has been updated it does not react.

6.1.4.7 Badge_Holder_Server

This is the repository server implementation class based on the template of the repository server.

- *create_objects* – creates instances of the *Badge_Holder_Rep_I*, *Badge_Rep_Observer_I*, *Sensor_Rep_Observer_I*, *User_Rep_Observer_I* and *Equipment_Rep_Observer_I* classes and calls *init* on the *Badge_Holder_Rep_I* instance.

- *delete_objects* – deletes the instances of the created objects.

6.1.5 Persistency

Currently, the persistent state of the repository is stored in files. Objects included in the entity description are stored in a form of their repository item ids. When a persistent state is revoked item ids are read from the file and cached. When a callback from the server manager comes saying that either *Badge Repository*, *Sensor Repository*, *User Repository* or *Equipment Repository* is active the *resolve_references* method restores the consistent state of the data base and makes observer objects register themselves. If it happens that during the inactivity of the repository the state of *Badge Repository*, *Sensor Repository*, *User Repository* or *Equipment Repository* has changed the same rules are applied as those used by observers in case of *add-update-remove* notification.

6.1.6 CORBA services usage

Current implementation uses only naming service.

6.1.7 Life cycle management

The repository stores its reference in the naming service using ABng Server Manager. Currently it must be launched by hand.

6.2 Implementation Specification

6.2.1 Source code organization

- *badge_holder_rep/badge_holder_i.h*
  badge_holder_rep/badge_holder_i.cc –
  implementation of the *Badge_Holder_I* class,

- *badge_holder_rep/badge_holder_rep_i.h*
  badge_holder_rep/badge_holder_rep_i.h –
  implementation of the *Badge_Holder_Rep_I* class,

- *badge_holder_rep/badge_rep_observer.h*
  badge_holder_rep/badge_rep_observer.cc –
  implementation of the *Badge_Rep_Observer_I* class,

- *badge_holder_rep/sensor_rep_observer.h*
  badge_holder_rep/sensor_rep_observer.cc –
  implementation of the *Sensor_Rep_Observer_I* class,

- *badge_holder_rep/user_rep_observer.h*
  badge_holder_rep/user_rep_observer.cc –
  implementation of the *User_Rep_Observer_I* class,

- *badge_holder_rep/equipment_rep_observer.h*
  badge_holder_rep/equipment_rep_observer.cc –
  implementation of the *Equipment_Rep_Observer_I* class,

- *badge_holder_rep/badge_holder_rep_svr.h*
  badge_holder_rep/badge_holder_svr.cc –
  implementation of the *Badge_Holder_Rep_Server* class,

- *badge_holder_rep/badge_holder_rep_main.cc* –
  implementation of the main method.

Chapter 7

User Repository

7.1 Functional Specification

7.1.1 Functionality description

User is an entity that contains user properties such as its id, forename, surname, position within a institution, email and phone. This list of properties could be extended in the future. The *User Repository* constitutes a database that stores *ABng Location Description* entities.

7.1.2 Entity

7.1.2.1 Interface description

```
#ifndef __USER_IDL__
#define __USER_IDL__ 1

interface User : Static_Data_Observed_E {

    struct Description {
        Repository_Item_Id item_id;
        string uid;
        string forename;
        string surname;
        string position;
        string email;
        string phone;
    };

    typedef sequence<Description> Descriptions;

    typedef Description Value_Description;
    typedef sequence<Value_Description> Value_Descriptions;

    struct Pattern {
        boolean is_any;
        Repository_Item_Id_Pattern item_id_pattern;
```

```
        String_Pattern uid_pattern;
        String_Pattern forename_pattern;
        String_Pattern surname_pattern;
        String_Pattern position_pattern;
        String_Pattern email_pattern;
        String_Pattern phone_pattern;
    };

    readonly attribute Description descr;
    readonly attribute Value_Description value_descr;
    readonly attribute Repository_Item_Id item_id;
    readonly attribute string uid;
    readonly attribute string forename;
    readonly attribute string surname;
    readonly attribute string position;
    readonly attribute string email;
    readonly attribute string phone;
};

#endif                                                    /* __USER_IDL__ */
```

The *Description* structure of the *User Description* entity contains:

- *item_id* – Repository Item Id of the entity,
- *uid*,
- *forename*,
- *surname*,
- *position*,
- *email*,
- *phone*.

Value_Description structure is equal to the *Description*.
A *User* entity supports the following attributes and operations:

- *descr* – get description of a given entity,
- *value_descr* – get value description of a given entity,
- *item_id* – get Repository Item Id of the entity,
- *forename* – get user forname,
- *surname* – get user surname,
- *position* – get user position,
- *email* – get user email,
- *phone* – get user phone.

7.1.2.2 Equality of entities

Two entities are considered equal if their Repository Item Ids are equal.

7.1.3 Repository

7.1.3.1 Interface description

```
#ifndef __USER_REP_IDL__
#define __USER_REP_IDL__ 1

interface User_Rep : Static_Data_Observed_R {

    typedef sequence<User> Users;

    Users find(in User::Pattern data_record_pattern)
        raises(Search_Not_Implemented);
    User::Value_Descriptions find_values(
        in User::Pattern data_record_pattern)
        raises(Search_Not_Implemented);
};

#endif                                          /* __USER_REP_IDL__ */

#ifndef __USER_REP_MANAGER_IDL__
#define __USER_REP_MANAGER_IDL__ 1

interface User_Rep_Manager {

    readonly attribute Domain_Repository::User_Rep rep;
    readonly attribute Domain_Repository::User_Rep::Users user_list;

    Domain_Repository::User add(
        in Domain_Repository::User::Description data_record)
        raises(Domain_Repository::Duplicate_Data_Record);

    void remove(
        in Domain_Repository::User object_to_remove)
        raises(Domain_Repository::Unknown_Object_Ref);

    void update(
        in Domain_Repository::User object_to_update,
        in Domain_Repository::User::Description data_record_pattern)
        raises(Domain_Repository::Unknown_Object_Ref,
                Domain_Repository::Duplicate_Data_Record);
};

#endif                                  /* __USER_REP_MANAGER_IDL__ */
```

7.1.3.2 Relationships

The *User Repository* is not dependent on any other repository or other ABng server.

7.1.4 Implementation classes

7.1.4.1 User_I

The class implements interfaces of *User* and *System_Static_Data_Observed_E*.

7.1.4.2 User_Rep_I

The class uses the building blocks of the repository template to implement functionality of *User_Rep* and *System_Static_Data_Observed_R* interfaces. It inherits from the *Basic_Rep_I<ALD_Types, SSDO_Types>* class, where the template parameter classes are defined as follows:

```
class SDO_Types {

public:
        typedef Domain_Repository::Static_Data_Observed_R Observed;
        typedef Domain_Repository::Static_Data_Observer Observer;
        typedef Domain_Repository::Static_Data_Observer_ptr Observer_ptr;
        typedef Domain_Repository::Static_Data_Observed_E::Static_Data_Observers
        Observers;
};

class LT_Types {

public:
        typedef Domain_Repository::User Item;
        typedef Domain_Repository::User_ptr Item_ptr;
        typedef Domain_Repository::User_Rep::Users Items;
        typedef User_I Item_I;
        typedef Domain_Repository::User::Description Initial_Data;
        typedef Domain_Repository::User::Description Description;
        typedef Domain_Repository::Static_Data_Observer::
                Static_Data_Object Data_Object;
};
```

The class overwrites the following template methods:

- *private_find* – invokes the function matching entities with the specified pattern. Its implementation is not complex as user description is simple and there is no object references to other entities within it. Thus find could be performed locally.

7.1.4.3 ABng_LD_Rep_Server

This is the repository server implementation class based on the template of the repository server.

- *create_objects* – creates instances of the *User_Rep_I* and calls *init* on the *User_Rep_I* instance.

- *delete_objects* – deletes the instances of the created objects.

7.1.5 Persistency

Currently, the persistent state of the repository is stored in files. Since the *User Repository* does not observe other repositories, the start-up and revoking the saved state are strightforward.

7.1.6 CORBA services usage

Current implementation uses only naming service.

7.1.7 Life cycle management

The repository stores its reference in the naming service using ABng Server Manager. Currently it must be launched by hand.

7.2 Implementation Specification

7.2.1 Source code organization

- *user_rep/user_i.h*
  user_rep/user_i.cc –
  implementation of the *User_I* class,

- *user_rep/user_rep_i.h*
  user_rep/user_rep_i.h –
  implementation of the *User_Rep_I* class,

- *user_rep/user_rep_svr.h*
  user_rep/ensor_svr.cc –
  implementation of the *User_Rep_Server* class,

- *user_rep/user_rep_main.cc* –
  implementation of the main method.

Chapter 8

Meta Equipment Repository

8.1 Functional Specification

8.1.1 Functionality description

Meta_Equipment is an entity that defines type of a equipment e.g. tv-set, air-conditioner etc. The meta equipment type definition includes: a name of the type, name of the vendor (manufacturer) of this type of equipment, its model, some set of fixed properties and a sequence of definitions of properties of arbitrary types. The *Meta_Equipment Repository* constitutes a database that stores *Meta_Equipment* entities.

8.1.2 Entity

8.1.2.1 Interface description

```
#ifndef __META_EQUIPMENT_IDL__
#define __META_EQUIPMENT_IDL__ 1

#include "IRtypes.idl"

interface Meta_Equipment : Static_Data_Observed_E {
    typedef Octets Image;

    typedef Octets_Pattern Image_Pattern;

    struct Dimensions {
        unsigned long height;
        unsigned long width;
        unsigned long depth;
    };

    struct Dimensions_Pattern {
        boolean is_any;
        Unsigned_Long_Pattern height_pattern;
        Unsigned_Long_Pattern width_pattern;
        Unsigned_Long_Pattern depth_pattern;
```

```
};

enum Control_Kind {INFRA_RED, ANYTHING_ELSE};

union Control_Info switch(Control_Kind) {
    case INFRA_RED: IR_Control_Data my_IR_control_data;
    default: unsigned long default_field;                    /* no meaning */
};

struct Control_Info_Pattern {
    boolean is_any;
    union IRCI_Pattern switch(Control_Kind) {
        case INFRA_RED: IR_Control_Data_Pattern my_IRCD_pattern;
        default: Unsigned_Long_Pattern default_field_pattern;
    } my_control_data_pattern;
};

struct Description {
    Repository_Item_Id item_id;

    string type;

    string vendor;
    string model;

    Dimensions my_dimensions;

    Control_Info my_control_info;

    Property_Descriptions
        specific_property_description_list;

    Image my_icon;
};

typedef sequence<Description> Descriptions;

typedef Description Value_Description;
typedef sequence<Value_Description> Value_Descriptions;

struct Pattern {
    Repository_Item_Id_Pattern item_id_pattern;

    boolean is_any;

    String_Pattern type_pattern;
    String_Pattern vendor_pattern;
    String_Pattern model_pattern;

    Dimensions_Pattern my_dimensions_pattern;

    Control_Info_Pattern my_control_info_pattern;
```

```
        Property_Descriptions_Pattern
            specific_property_description_list_pattern;

        Image_Pattern my_icon_pattern;
    };

    readonly attribute Description descr;
    readonly attribute Value_Description value_descr;
    readonly attribute Repository_Item_Id item_id;
    readonly attribute string type;
    readonly attribute string vendor;
    readonly attribute string model;
    readonly attribute Dimensions my_dimensions;
    readonly attribute Control_Info my_control_info;
    readonly attribute Property_Descriptions
                          specific_property_description_list;
    readonly attribute Image my_icon;
};

#endif                                    /* __META_EQUIPMENT_IDL__ */
```

The *Description* structure of the *Meta_Equipment* entity contains:

- *item_id* – Repository Item Id of the entity,
- *type* – type name,
- *vendor* – the name of a vendor or manufacturer,
- *model* – id of the equipment model,
- *my_dimensions* – equipment dimensions: height, width and depth,
- *my_control_info* – a union storing information whether the given equipment is controlled and if yes then the data enabling a performance of controlling operations is stored as well,
- *my_icon* – an icon in the GIF format representing the equipment.

Value_Description structure is equal to the *Description*.
A *Meta_Equipment* entity supports the following attributes and operations:

- *descr* – get description of a given entity,
- *value_descr* – get value description of a given entity,
- *item_id* – get Repository Item Id of the entity,
- *type* – get type name,
- *venodr* – get the name of a vendor or manufacturer,

- *model* – get id of the equipment model,
- *my_dimensions* – get a structure with equipment dimensions,
- *my_control_info* – get a union containing information enabling controlling of the equipment
- *my_icon* – get an icon representing the equipment.

8.1.2.2 Equality of entities

Two entities are considered equal if they have the same system type, vendor and model names.

8.1.3 Repository

8.1.3.1 Interface description

```
#ifndef __META_EQUIPMENT_REP_IDL__
#define __META_EQUIPMENT_REP_IDL__ 1

interface Meta_Equipment_Rep : Static_Data_Observed_R {

    typedef sequence<Meta_Equipment> Meta_Equipments;

    Meta_Equipments find(
        in Meta_Equipment::Pattern data_record_pattern)
        raises(Search_Not_Implemented);
    Meta_Equipment::Value_Descriptions find_values(
        in Meta_Equipment::Pattern data_record_pattern)
        raises(Search_Not_Implemented);
};

#endif                                          /* __META_EQUIPMENT_REP_IDL__ */

#ifndef __META_EQUIPMENT_REP_MANAGER_IDL__
#define __META_EQUIPMENT_REP_MANAGER_IDL__ 1

interface Meta_Equipment_Rep_Manager {

    readonly attribute Domain_Repository::Meta_Equipment_Rep rep;
    readonly attribute Domain_Repository::Meta_Equipment_Rep::Meta_Equipments
        meta_equipment_list;

    Domain_Repository::Meta_Equipment add(
        in Domain_Repository::Meta_Equipment::Description data_record)
        raises(Domain_Repository::Duplicate_Data_Record,
               Domain_Repository::Invalid_Description);

    void remove(in Domain_Repository::Meta_Equipment object_to_remove)
        raises(Domain_Repository::Unknown_Object_Ref,
               Domain_Repository::Operation_Not_Allowed);

    void update(
```

```
        in Domain_Repository::Meta_Equipment object_to_update,
        in Domain_Repository::Meta_Equipment::Description data_record_pattern
    ) raises(Domain_Repository::Unknown_Object_Ref,
                Domain_Repository::Duplicate_Data_Record,
                Domain_Repository::Invalid_Description);
};

#endif                                  /* _META_EQUIPMENT_REP_MANAGER_IDL__ */
```

8.1.3.2 Relationships

The *Meta_Equipment Repository* is related to *Equipment Repository*. Removing a *Meta_Equipment* entity for which a corresponding *Equipment* entities exist is illegal, therefore before each remove it is checked in the *Equipment Repository* whether it contains an entity of the type being removed. The *Meta_Equipment Repository* does not register itself as an observer in any other repository.

8.1.4 Implementation classes

8.1.4.1 Meta_Equipment_I

The class implements interfaces of *Meta_Equipment* and *System_Static_Data_Observed_E*.

8.1.4.2 Meta_Equipment_Rep_I

The class uses the building blocks of the repository template to implement functionality of *Meta_Equipment_Rep* and *System_Static_Data_Observed_R* interfaces. It inherits from the *Basic_Rep_I<ALD_Types, SSDO_Types>* class, where the template parameter classes are defined as follows:

```
class SDO_Types {

public:
        typedef Domain_Repository::Static_Data_Observed_R Observed;
        typedef Domain_Repository::Static_Data_Observer Observer;
        typedef Domain_Repository::Static_Data_Observer_ptr Observer_ptr;
        typedef Domain_Repository::Static_Data_Observed_E::Static_Data_Observers
                Observers;
};

class LT_Types {

public:
        typedef Domain_Repository::Meta_Equipment Item;
        typedef Domain_Repository::Meta_Equipment_ptr Item_ptr;
        typedef Domain_Repository::Meta_Equipment_Rep::Meta_Equipments Items;
        typedef Meta_Equipment_I Item_I;
        typedef Domain_Repository::Meta_Equipment::Description Initial_Data;
        typedef Domain_Repository::Meta_Equipment::Description Description;
        typedef Domain_Repository::Static_Data_Observer::
                Static_Data_Object Data_Object;
```

};

The class overwrites the following template methods:

- *remove_permitted* – the new implementation of the methods checks whether the *Equipment Repository* contains entities with type, vendor and model names of a removed entity. If so, the removing is not allowed.
- *validate_update_data* – ensures that the update does not change the entity's type, vendor and model names.
- *private_find* – invokes the function matching entities with the specified pattern. Its implementation is not simple as there are a lot of complex properties to match. However, there is no object references to other entities within it, thus find could be performed locally.

8.1.4.3 Meta_Equipment_Rep_Server

This is the repository server implementation class based on the template of the repository server.

- *create_objects* – creates instances of the *Meta_Equipment_Rep_I* and calls *init* on the *Meta_Equipment_Rep_I* instance.
- *delete_objects* – deletes the instances of the created objects.

8.1.5 Persistency

Currently, the persistent state of the repository is stored in files. Since the *Meta_Equipment Repository* does not observe other repositories, the start-up and revoking the saved state are straightforward.

8.1.6 CORBA services usage

Current implementation uses only naming service.

8.1.7 Life cycle management

The repository stores its reference in the naming service using ABng Server Manager. Currently it must be launched by hand.

8.2 Implementation Specification

8.2.1 Source code organization

- *meta_equipment_rep/meta_equipment_i.h*
  meta_equipment_rep/meta_equipment_i.cc –
  implementation of the *Meta_Equipment_I* class,

- *meta_equipment_rep/meta_equipment_rep_i.h*
  meta_equipment_rep/meta_equipment_rep_i.h –
  implementation of the *Meta_Equipment_Rep_I* class,
- *meta_equipment_rep/meta_equipment_rep_svr.h*
  meta_equipment_rep/meta_equipment_svr.cc –
  implementation of the *Meta_Equipment_Rep_Server* class,
- *meta_equipment_rep/meta_equipment_rep_main.cc* –
  implementation of the main method.

Chapter 9

Equipment Repository

9.1 Functional Specification

9.1.1 Functionality description

Equipment is an entity that describes particular instances of equipment available in the ABng environment. The *Equipment Repository* constitutes a database that stores *Equipment* entities.

9.1.2 Entity

9.1.2.1 Interface description

```
#ifndef __EQUIPMENT_IDL__
#define __EQUIPMENT_IDL__ 1

interface Equipment : Static_Data_Observed_E {

    struct Description {
        Repository_Item_Id item_id;
        Meta_Equipment meta_data;
        string name;
        string last_command;
        Properties property_list;
    };

    typedef sequence<Description> Descriptions;

    struct Value_Description {
        Repository_Item_Id item_id;
        Meta_Equipment::Description meta_data;
        string name;
        string last_command;
        Properties property_list;
    };
    typedef sequence<Value_Description> Value_Descriptions;
```

```
    struct Pattern {
        boolean is_any;
        Repository_Item_Id_Pattern item_id_pattern;
        Meta_Equipment::Pattern meta_data_pattern;
        String_Pattern name_pattern;
        String_Pattern last_command_pattern;
        Properties_Pattern property_list_pattern;
    };

    Description descr()
        raises(Destination_Unreachable);

    Value_Description value_descr()
        raises(Destination_Unreachable);

    readonly attribute Repository_Item_Id item_id;

    Meta_Equipment meta_data()
        raises(Destination_Unreachable);

    readonly attribute string name;
    readonly attribute string last_command;
    readonly attribute Properties property_list;

};

#endif                                          /* __EQUIPMENT_IDL__ */
```

The *Description* structure of the *Equipment* entity contains:

- *item_id* – Repository Item Id of the entity,
- *meta_data* – a reference of the corresponding *Meta_Equipment* object,
- *name* – the name of a given piece of equipment.
- *last_command* – the name of a last command.
- *property_list* – a list of properties.

Value_Description structure is composed of:

- *item_id* – Repository Item Id of the entity,
- *meta_data* – a value description of the corresponding *Meta_Equipment* object,
- *name* – the name of a given piece of equipment.
- *last_command* – the name of a last command.
- *property_list* – a list of properties.

A *Equipment* entity supports the following attributes and operations:

- *descr* – get description of a given entity,
- *value_descr* – get value description of a given entity; this method necessitates obtaining value descriptions of the *Meta Equipment* object corresponding to the given entity from the appropriate repository; if the repository cannot be contacted the exception is thrown,
- *item_id* – get Repository Item Id of the entity,
- *meta_data* – get a reference of the corresponding *Meta Equipment* object; the exception is thrown when the reference is nil,
- *property_list* – get the sequence of properties,
- *name* – get the name of the entity,
- *last_command* – get the last command of the entity.

9.1.2.2 Equality of entities

Two entities are considered equal if they possess the same name.

9.1.3 Repository

9.1.3.1 Interface description

```
#ifndef __EQUIPMENT_REP_IDL__
#define __EQUIPMENT_REP_IDL__ 1

interface Equipment_Rep : Static_Data_Observed_R {

    typedef sequence<Equipment> Equipments;

    Equipments find(in Equipment::Pattern data_record_pattern)
        raises(Search_Not_Implemented,
               Destination_Unreachable);
    Equipment::Value_Descriptions find_values(
        in Equipment::Pattern data_record_pattern)
        raises(Search_Not_Implemented,
               Destination_Unreachable);
};

#endif                                   /* __EQUIPMENT_REP_IDL__ */

#ifndef __EQUIPMENT_REP_MANAGER_IDL
#define __EQUIPMENT_REP_MANAGER_IDL 1

interface Equipment_Rep_Manager {

    readonly attribute Domain_Repository::Equipment_Rep rep;
    readonly attribute Domain_Repository::Equipment_Rep::Equipments
            equipment_list;
```

```
    Domain_Repository::Equipment add(in
        Domain_Repository::Equipment::Description data_record)
        raises(Domain_Repository::Duplicate_Data_Record,
               Domain_Repository::Invalid_Description,
               Domain_Repository::Destination_Unreachable);
    void remove(in Domain_Repository::Equipment object_to_remove)
        raises(Domain_Repository::Unknown_Object_Ref);
    void update(in Domain_Repository::Equipment object_to_update,
        in Domain_Repository::Equipment::Description data_record_pattern)
        raises(Domain_Repository::Unknown_Object_Ref,
               Domain_Repository::Duplicate_Data_Record,
               Domain_Repository::Invalid_Description,
               Domain_Repository::Destination_Unreachable);
};

#endif                                  /* __EQUIPMENT_REP_MANAGER_IDL */
```

9.1.3.2 Relationships

The *Equipment Repository* observes the *Meta Equipment Repository*. It attaches itself as an observer to all entities stored there. Whenever it receives a notification that a *Meta Equipment* has been removed it also removes all *Equipment* entities that correspond to the removed *Meta Equipment* object from its database. The *Equipment Repository* does not react to *Meta Equipment* updates.

9.1.4 Implementation classes

9.1.4.1 Equip

The class implements interfaces of *Equipment* and *Static_Data_Observed_E*. It encapsulates the following member fields:

- *orb* – the ORB reference,
- *meta_equipment_item_id* – the item id of the associated *Meta Equipment* object,
- *corba_object* – the reference of the associated IDL generated object,
- *container* – the *Equipment Repository* containing this object,
- *description* – the description of the entity.

The class implements the following IDL operations:

- *descr* – get description of the entity,
- *value_descr* – get value description of the entity,
- *item_id* – get item id of the entity,

- *meta_data* – get reference of the *Meta Equipment* associated with the given entity,
- *name* – get the name of the entity,
- *last_command* – get the last command of the entity,
- *property_list* – get the list of properties,
- *static_data_observer_list* – get the list of registered observers,
- *attach_static_data_observer* – register a new observer,
- *detach_static_data_observer* – detach an observer.

The remaining operations are:

- *Equip* – constructors,
- *get_corba_object* – get the value of *corba_object*,
- *get_meta_equipment_item_id* – get the value of the *meta_equipment_item_id*,
- *set_item_id* – set *item_id*,
- *set_descr* – set *description*,
- *set_meta_data* – set *meta_data* field of the *description*,
- *save* – save persistent state,
- *match_property* – return *true* if a specified property matches the specified property pattern – invoked by *match_equip* operation,
- *match_equip* – return *true* if the entity matches the specified equipment pattern, invoked by the *find* and *find_values* methods of the *Equip_Rep* class,
- *change_type* – check if the property list matches the property description list of the corresponding *Meta Equipment* object; remove all properties that do not exist in the property description list. For all unmatched property descriptions create properties with a default value. The method is invoked by the *update_type* operation of the *Equip_Rep* class,
- *check_record* – check if the properties of the new description match corresponding property descriptions of the associated *Meta Equipment* object, return item id of this object,
- *equals* – return *true* if the entity is equal to the specified entity.

9.1.4.2 Equip_Rep

The class *Equip_Rep* implements interfaces of the *Equipment Repository*, *Equipment Repository Manager* and *Static_Data_Observed_R*. The member fields of the class are:

- *Repository_DB_Name* – the name of the file storing the database,
- *manager_object* – the reference of the IDL generated manager object,
- *repository_object* – the reference of the IDL generated repository object,
- *observer_object* – the reference of the IDL generated observer object,
- *equip_list* – the hash table of all *Equipment* objects,
- *equip_deref* – the hash table that associates all *Equip* objects with their Orbix references,
- *item_ids_to_observers_list* – the hash table that binds *Equipment* item ids with the lists of the corresponding observers,
- *observers_to_item_ids_list* – the hash table that binds observers with the lists of the corresponding *Equipment* item ids,
- *add_static_data_observer_list* – the hash table that stores all observers interested with the notification of the addition of a new entity,
- *meta_equipment_rep* – the reference of the *Meta Equipment Repository*,
- *bootstrap_listener* – the *Bootstrap_listener_n* object, on which a callback from the server manager is to be invoked,
- *item_id* – the current item id.

The class implements the following IDL operations:

- *_notify* – handle the notified change of the *Meta Equipment Repository*,
- *equipment_list* – get the list of entities,
- *add* – add the entity and notify the observers. The operation throws the *Duplicate_Data_Record* when an attempt is made to add the entity which is equal (in terms of *Equipment* equality) to some other object in the repository,
- *remove* – remove the specified entity,
- *update* – update the specified entity. It is not allowed to change the *Meta Equipment* object associated with the given entity. Moreover the change may not cause that two equal (in terms of *Equipment* equality) entities exist in the repository. Also, when it is not possible to contact the *Meta Equipment Repository* the *Destination_Unreachable* is thrown,

- *find* – find all entities that match the specified pattern,
- *find_values* – find value descriptions of all entities that match the specified pattern,
- *static_data_observer_list* – get all observers,
- *attach_static_data_observer* – attach an observer the specified entities,
- *detach_static_data_observer* – detach an observer form the specified entities,
- *add_static_data_observer_list* – get all observers interested in the *add* operation,
- *attach_add_static_data_observer* – attach the observer interested in the *add* operation,
- *detach_add_static_data_observer* – detach the observer interested in the *add* operation,
- *reference_resolved* – the *Meta Equipment Repository* is up. Review the database and make it consistent with the *Meta Equipment Repository*

The remaining methods are:

- *get_manager_object* – get the value of the *manager_object*,
- *get_repository_object* – get the value of the *repository_object*,
- *get_observer_object* – get the value of the *observer_object*,
- *static_data_observer_list* – get the list of observers of the specified item,
- *update_type* – calls the *change_type* operation of the *Equip* object to adjust its description to the possibly new *Meta Equipment* description. If the operation changes the entity (returns *true*) the observers of the given entity are notified and the persistent database is updated,
- *check_equipments* – attach the repository as an observer to all entities of the *Meta Equipment Repository*. Moreover, adjust the entities to the new state of the *Meta Equipment Repository* – call *update_type* to update properties, *set_meta_data* to update the *meta_data* description field and remove all entities for which the corresponding *Meta Equipment* object no longer exists,
- *meta_equipment_change* – invoked by the *notify* method when a notification of the new object removal or update is handled. It removes the entities whose the corresponding *Meta Equipment* object has been removed or updates all entities for which the corresponding *Meta Equipment* object has been updated,
- *rep* – get the value of the *repository_object*,
- *detach_dead_observers* – detach those observers that have been recognized as no longer active,

- *deref_object* – find the implementation object for the given CORBA object – invoked by *update* and *remove* methods,
- *get_item_ids_list* – get item ids of all entities observed by the specified observer,
- *inform_observers* – inform all observers of the given entity about the specified change,
- *check_unknown_item_ids* – check if all entities corresponding to the specified items are observed by some observer,
- *save* – save the persistent state,
- *restore* – restore the persistent state,

9.1.4.3 Bootstrap_base

The class is a wrapper for Naming Service.

- *Bootstrap_base* – a constructor, resolves reference of Naming Service,
- *incremental_bind* – binds an object to a name in Naming Service, any contexts required to bind the object are created if necessary,
- *resolve* – resolves a denoted name to a CORBA object,
- *file_to_object* – reads an *IOR* from a file and returns a CORBA object.

9.1.4.4 Bootstrap_listener_interface_n

The interface one should implement to be notified about server startups and shutdowns. It is a wrapper interface for *ABng_Server_Manager::Callback* interface introduced for backward compatibility to the previous solution.

- *reference_resolved* – called when a server starts up, corresponds to *ABng_Server_Manager::Callback::server_startup* (see chapter 2)
- *server_shutdown* – called when a server shuts down, corresponds to *ABng_Server_Manager::Callback::server_shutdown* (see chapter 2)

9.1.4.5 Bootstrap_listener_n

The class implements *IDL* interface *ABng_Server_Manager::Callback* and forwards all call-backs to an associated object that implements *Bootstrap_listener_interface_n* interface.

The remaining methods are:

- *Bootstrap_listener_n* – a constructor
- *register_callback* – registers a callback for a server (see description of *ABng_Server_Manager::Register::register_callback* method)

- *register_context_callback* – see *ABng_Server_Manager::Register::register_context_callback*
- *unregister_callback* – see *ABng_Server_Manager::Register::unregister_callback*

9.1.4.6 Bootstrap_n

The wrapper class for *ABng_Server_Manager* interfaces, contains some helper utilities, all methods in this class are static.

- *init* – initializes the class, it should be called before any other methods are used,
- *get_register_object* – returns a reference to *ABng_Server_Manager::Register* object in *server_manager* server,
- *register_server_in_NS* – registers a server in *server_manager* (and Naming Service)
- *unregister_server_from_NS* – unregisters a server from *server_manager* (and Naming Service),
- *concat_names* – concatenates two complex names (sequneces of strings),
- *equal_names* – checks if two complex names are equal,
- *name_in_context* – checks if a name is registered in a context,
- *CTX...* – methods returning names of context as arrays of strings,
- *SRV...* – methods returning names of server in Naming Context as arrays of strings.

9.1.4.7 bugfixes

There is a directory *bugfixes* that contains classes responsible for translating Orbix *TypeCodes* to and from OMG *TypeCodes*. As all other servers are running *Orbix 2.2*, which uses Orbix *TypeCodes*, while the *OrbixWeb 3.0* uses OMG *TypeCodes*, it was necessary translate *TypeCodes*.

The directory contains also some patches for generated stubs and skeletons of the *Core* that calls translating routines in appropriate places.

9.1.4.8 Equip_Main

The main class of the repository, implements static method *main* necessary for a *Java* application.

9.1.5 Persistency

Currently, the persistent state of the repository is stored in files. Values of *CORBA::Any* type are stored as sequence of hexadecimal octets that represent a *CDR* (Common Data Representation) of the value. Saving *CORBA::Any* was a primary factor of moving the server to Java, as for CORBA–Java mapping there is guarantee of portable serialization/deserialization of any value (including *CORBA::Any*), while for C++ mapping there is no such functionality.

9.1.6 Persistency

Currently, the persistent state of the repository is stored in files. Objects included in the entity description are stored in a form of their repository item ids. When a persistent state is revoked item ids are read from the file and cached. When a callback from the server manager comes saying that *Meta Equipment Repository* is active the *resolve_references* method restores the consistent state of the data base and registers itself as an observer. If it happens that during the inactivity of the repository the state of *Meta Equipment Repository* has changed the same rules are applied as those used in case of *add-update-remove* notification.

9.1.7 CORBA services usage

Current implementation uses only naming service.

9.1.8 Life cycle management

The repository stores its reference in the naming service using ABng Server Manager. Currently it must be launched by hand.

9.2 Implementation Specification

9.2.1 Source code organization

- *equipment_rep/EquipmentRep/Equip.java* – implementation of the *Equip* class,
- *equipment_rep/EquipmentRep/Equip_Rep.java* – implementation of the *Equip_Rep* class,
- *equipment_rep/EquipmentRep/Equip_Main.java* – implementation of the *Equip_Main* class.

  equipment_rep/EquipmentRep/Bootstrap_listner_interface_n.java –
  interface *Bootstrap_listener_interface_n*.
- *equipment_rep/EquipmentRep/Bootstrap_base.java* – implementation of the *Bootstrap_base* class.
- *equipment_rep/EquipmentRep/Bootstrap_n.java* – implementation of the *Bootstrap_n* class.
- *equipment_rep/EquipmentRep/Bootstrap_listner_n.java* – interface *Bootstrap_listener_n*.

Chapter 10

Sensor Repository

10.1 Functional Specification

10.1.1 Functionality description

Sensor is an entity that represents a sensor device within the ABng software. The *Sensor Repository* constitutes a database that stores *Sensor* entities.

10.1.2 Entity

10.1.2.1 Interface description

```
#ifndef __SENSOR_IDL__
#define __SENSOR_IDL__ 1

interface Sensor: System_Static_Data_Observed_E, Sensor_Commander {

    struct Description {
        Domain_Repository::Repository_Item_Id item_id;
        Sensor_Id id;
    };

    typedef sequence<Description> Descriptions;

    typedef Description Value_Description;
    typedef sequence<Value_Description> Value_Descriptions;

    struct Pattern {
        boolean is_any;
        Domain_Repository::Repository_Item_Id_Pattern item_id_pattern;
        Sensor_Id_Pattern id_pattern;
    };

    // Yes, yes.... I know you don't want it here....
    // But try to compile my rep. without this definition :(
    // G.
```

```
    struct Sensors_Pattern {
        boolean is_any;
        Domain_Repository::Sequence_Pattern_Constraints constraints;
        sequence<Pattern> sensors_pattern;
    };

    readonly attribute Description descr;
    readonly attribute Value_Description value_descr;
    readonly attribute Domain_Repository::Repository_Item_Id item_id;
    readonly attribute Sensor_Id id;
};

#endif                                                  /* __SENSOR_IDL__ */
```

The *Description* structure of the *Sensor* entity contains:

- *item_id* – Repository Item Id of the entity,
- *id* – a structure containing sensor system id: its number, the number of the sensor network and the name of the group of networks to which the network of this sensor belongs.

Value_Description structure is equal to the *Description*.
A *Sensor* entity supports the following attributes and operations:

- *descr* – get description of a given entity,
- *value_descr* – get value description of a given entity,
- *item_id* – get Repository Item Id of the entity,
- *id* – get a structure containing sensor system id.

10.1.2.2 Equality of entities

Two entities are considered equal if they have the same system id.

10.1.3 Repository

10.1.3.1 Interface description

```
#ifndef __SENSOR_REP_IDL__
#define __SENSOR_REP_IDL__ 1

interface Sensor_Rep : ABng_Core_System::System_Static_Data_Observed_R {

    typedef ABng_Core_System::Sensor::Sensors_Pattern Sensors_Pattern;

    readonly attribute ABng_Core_System::Sensors sensor_list;

    ABng_Core_System::Sensor add(
```

```
        in ABng_Core_System::Sensor::Description data_record)
        raises(Domain_Repository::Duplicate_Data_Record);
    void remove(in ABng_Core_System::Sensor object_to_remove)
        raises(Domain_Repository::Unknown_Object_Ref);
    ABng_Core_System::Sensors find(
        in ABng_Core_System::Sensor::Pattern data_record_pattern)
        raises(Domain_Repository::Search_Not_Implemented);
    ABng_Core_System::Sensor::Value_Descriptions find_values(
        in ABng_Core_System::Sensor::Pattern data_record_pattern)
        raises(Domain_Repository::Search_Not_Implemented);
    void update(in ABng_Core_System::Sensor object_to_update,
        in ABng_Core_System::Sensor::Description data_record_pattern)
        raises(Domain_Repository::Unknown_Object_Ref,
            Domain_Repository::Duplicate_Data_Record);
};

#endif                                          /* __SENSOR_REP_IDL__ */
```

10.1.3.2 Relationships

The *Sensor Repository* is not dependent on any other repository. however, it needs a Scheduler to enable invocations of sensor commands.

10.1.4 Implementation classes

10.1.4.1 Sensor_I

The class implements interfaces of *Sensor* and *System_Static_Data_Observed_E*. Additionally, it implements the *Sensor_Commander* interface. This interface possesses methods enabling access to the full functionality of a sensor device. Further, it make possible performing operations on the Scheduler request queue through the *Scheduler_Request_Queue* interface.

10.1.4.2 Sensor_Rep_I

The class uses the building blocks of the repository template to implement functionality of *Sensor_Rep* and *System_Static_Data_Observed_R* interfaces. It inherits from the *Basic_Rep_I<ALD_Types, SSDO_Types>* class, where the template parameter classes are defined as follows:

```
class SDO_Types {

public:
        typedef ABng_Core_System::System_Static_Data_Observed_R Observed;
        typedef ABng_Core_System::System_Static_Data_Observer Observer;
        typedef ABng_Core_System::System_Static_Data_Observer_ptr Observer_ptr;
        typedef ABng_Core_System::System_Static_Data_Observed_E::
                System_Static_Data_Observers Observers;
};

class LT_Types {
```

```
public:
        typedef ABng_Core_System::Sensor Item;
        typedef ABng_Core_System::Sensor_ptr Item_ptr;
        typedef ABng_Core_System::Sensors Items;
        typedef Sensor_I Item_I;
        typedef ABng_Core_System::Sensor::Description Initial_Data;
        typedef ABng_Core_System::Sensor::Description Description;
        typedef ABng_Core_System::System_Static_Data_Observer::
                System_Static_Data_Object Data_Object;
};
```

The class overwrites the following template methods:

- *private_find* – invokes the function matching entities with the specified pattern. Its implementation is not complex as sensor description is simple and there is no object references to other entities within it. Thus find could be performed locally.

10.1.4.3 Sensor_Rep_Server

This is the repository server implementation class based on the template of the repository server.

- *create_objects* – creates instances of the *Sensor_Rep_I* and calls *init* on the *Sensor_Rep_I* instance.
- *delete_objects* – deletes the instances of the created objects.

10.1.5 Persistency

Currently, the persistent state of the repository is stored in files. Since the *Sensor Repository* does not observe other repositories, the start-up and revoking the saved state are strightforward.

10.1.6 CORBA services usage

Current implementation uses only naming service.

10.1.7 Life cycle management

The repository stores its reference in the naming service using ABng Server Manager. Currently it must be launched by hand.

10.2 Implementation Specification

10.2.1 Source code organization

- *sensor_rep/sensor_i.h*
  sensor_rep/sensor_i.cc –
  implementation of the *Sensor_I* class,

- *sensor_rep/sensor_rep_i.h*
  sensor_rep/sensor_rep_i.h –
  implementation of the *Sensor_Rep_I* class,
- *sensor_rep/sensor_rep_svr.h*
  sensor_rep/sensor_svr.cc –
  implementation of the *Sensor_Rep_Server* class,
- *sensor_rep/sensor_rep_main.cc* –
  implementation of the main method.

Chapter 11

Location Type Repository

11.1 Functional Specification

11.1.1 Functionality description

Location Type is an entity that defines type of a location e.g. room, hall etc. The location type definition includes: a name of the type and a sequence of properties of arbitrary types. *Location Type Repository* constitutes a database that stores *Location Type* entities. It supports operations that allow searching and modifying its contents.

11.1.2 Entity

11.1.2.1 Interface description

```
#ifndef __LOCATION_TYPE_IDL__
#define __LOCATION_TYPE_IDL__ 1

interface Location_Type : Static_Data_Observed_E {

    typedef string Location_Type_Name;
    typedef String_Pattern Location_Type_Name_Pattern;

    struct Description {
        Repository_Item_Id item_id;
        Location_Type_Name name;
        Property_Descriptions property_description_list;
    };

    typedef sequence<Description> Descriptions;

    typedef Description Value_Description;
    typedef sequence<Value_Description> Value_Descriptions;

    struct Pattern {
        boolean is_any;
        Repository_Item_Id_Pattern item_id_pattern;
```

```
        Location_Type_Name_Pattern name_pattern;
        Property_Descriptions_Pattern
            property_description_list_pattern;
    };

    readonly attribute Description descr;
    readonly attribute Value_Description value_descr;
    readonly attribute Repository_Item_Id item_id;
    readonly attribute Location_Type_Name name;
    readonly attribute Property_Descriptions property_description_list;
};

#endif                                          /* __LOCATION_TYPE_IDL__ */
```

The *Description* structure of the *Location Type* entity contains:

- *item_id* – Repository Item Id of the entity,
- *name* – type name,
- *property_description_list* – sequence of property descriptions.

Value_Description structure is equivalent to *Description*.
A *Location Type* entity supports the following attributes:

- *descr* – get description of a given entity,
- *value_descr* – get value description of a given entity; in *Location Type* it is equivalent to *descr*,
- *item_id* – get Repository Item Id of the entity,
- *name* – get name of the *Location Type* entity,
- *property_description_list* – get the sequence of property descriptions.

11.1.2.2 Equality of entities

Two entties are considered equal if their type names (*name* fields of their descriptions) are equal. The repository may not store any two entities that are equal to one another. Also, it is not possible to change type name during update.

11.1.3 Repository

11.1.3.1 Interface description

```
#ifndef __LOCATION_TYPE_REP_IDL__
#define __LOCATION_TYPE_REP_IDL__ 1
```

```
interface Location_Type_Rep : Static_Data_Observed_R {

    typedef sequence<Location_Type> Location_Types;

    Location_Types find(
        in Location_Type::Pattern data_record_pattern)
        raises(Search_Not_Implemented);
    Location_Type::Value_Descriptions find_values(
        in Location_Type::Pattern data_record_pattern)
        raises(Search_Not_Implemented);
};

#endif                                  /* __LOCATION_TYPE_REP_IDL__ */
```

```
#ifndef __LOCATION_TYPE_REP_MANAGER_IDL__
#define __LOCATION_TYPE_REP_MANAGER_IDL__ 1

interface Location_Type_Rep_Manager {

    readonly attribute Domain_Repository::Location_Type_Rep rep;
    readonly attribute Domain_Repository::Location_Type_Rep::Location_Types
        location_type_list;

    Domain_Repository::Location_Type add(
        in Domain_Repository::Location_Type::Description data_record)
        raises(Domain_Repository::Duplicate_Data_Record,
               Domain_Repository::Invalid_Description);
    void remove(in Domain_Repository::Location_Type object_to_remove)
        raises(Domain_Repository::Unknown_Object_Ref,
               Domain_Repository::Operation_Not_Allowed);
    void update(in Domain_Repository::Location_Type object_to_update,
        in Domain_Repository::Location_Type::Description data_record_pattern)
        raises(Domain_Repository::Unknown_Object_Ref,
               Domain_Repository::Duplicate_Data_Record,
               Domain_Repository::Invalid_Description);
};

#endif                          /* __LOCATION_TYPE_REP_MANAGER_IDL__ */
```

11.1.3.2 Relationships

The *Location Type Repository* is related to *Location Description Repository*. Removing a *Location Type* entity for which a corresponding *Location Description* entity exists is illegal, therefore before each remove it is checked in the *Location Description Repository* whether it contains an entity of the type being removed.

The *Location Type Repository* does not register itself as an observer in any other repository.

11.1.4 Implementation classes

11.1.4.1 Location_Type_I

The class implements interfaces of *Location_Type* and *Static_Data_Observed_E*.

11.1.4.2 Location_Type_Rep_I

The class uses the building blocks of the repository template to implement functionality of *Location_Type_Rep*, *Location_Type_Rep_Manager* and *Static_Data_Observed_R* interfaces. The class inherits from *Basic_Rep_I<LT_Types, SDO_Types>*, where the parameter classes are defined as follows:

```
class SDO_Types {

public:

  typedef Domain_Repository::Static_Data_Observed_R Observed;
  typedef Domain_Repository::Static_Data_Observer Observer;
  typedef Domain_Repository::Static_Data_Observer_ptr Observer_ptr;
  typedef Domain_Repository::Static_Data_Observed_E::Static_Data_Observers
    Observers;
};

class LT_Types {

public:

  typedef Domain_Repository::Location_Type Item;
  typedef Domain_Repository::Location_Type_ptr Item_ptr;
  typedef Domain_Repository::Location_Type_Rep::Location_Types Items;
  typedef Location_Type_I Item_I;
  typedef Domain_Repository::Location_Type::Description Initial_Data;
  typedef Domain_Repository::Static_Data_Observer::Static_Data_Object
    Data_Object;
};
```

The class overwrites the following template methods:

- *remove_permitted* – the new implementation of the methods checks whether the *Location Description Repository* contains entities with type name of a removed entity. If so, the removing is not allowed.
- *validate_update_data* – ensures that the update does not change the entity's name.
- *private_find* – invokes the function matching entities with the specified pattern.

11.1.4.3 LT_Rep_Server

This is the repository server implementation class based on the template of the repository server.

- *create_objects* – creates and initializes an instance of the *Location_Type_Rep_I* class.
- *delete_objects* – deletes the instance of the *Location_Type_Rep_I* class.

11.1.5 Persistency

Currently, the persistent state of the repository is stored in files. Since the *Location Type Repository* does not observe other repositories, the start-up and revoking the saved state are stright-forward.

11.1.6 CORBA services usage

Current implementation uses only naming service.

11.1.7 Life cycle management

The repository stores its reference in the naming service using ABng Server Manager. Currently it must be launched by hand.

11.2 Implementation Specification

11.2.1 Source code organization

- *location_type_rep/location_type_i.h, location_type_rep/location_type_i.cc* – definition of the *Location_Type_I* class,
- *location_type_rep/location_type_rep_i.h, location_type_rep/location_type_rep_i.cc* – definition of the *Location_Type_Rep_I* class,
- *location_type_rep/lt_rep_server.h, location_type_rep/lt_rep_server.cc* – definition of the *LT_Rep_Server* class,
- *location_type_rep/domain_srv.cc* – definition of the main function.

Chapter 12

Location Description Repository

12.1 Functional Specification

12.1.1 Functionality description

Equipment is an entity that describes particular instances of location description available in the ABng environment. The *Location Description Repository* constitutes a database that stores *Location Description* entities.

12.1.2 Entity

12.1.2.1 Interface description

```
#ifndef __LOCATION_DESCRIPTION_IDL__
#define __LOCATION_DESCRIPTION_IDL__ 1

interface Location_Description : Static_Data_Observed_E {

    struct Description {
        Repository_Item_Id item_id;
        Location_Type::Location_Type_Name type_name;
        Properties property_list;
    };

    typedef sequence<Description> Descriptions;

    typedef Description Value_Description;
    typedef sequence<Value_Description> Value_Descriptions;

    struct Pattern {
        boolean is_any;
        Repository_Item_Id_Pattern item_id_pattern;
        Location_Type::Location_Type_Name_Pattern type_name_pattern;
        Properties_Pattern property_list_pattern;
    };
```

```
    readonly attribute Description descr;
    readonly attribute Value_Description value_descr;
    readonly attribute Repository_Item_Id item_id;

// Original:    readonly attribute Location_Type::Location_Type_Name type_name;
// typedef unwinded due to a bug in OrbixWeb 3.0
    readonly attribute string type_name;

    readonly attribute Properties property_list;
};

#endif                                    /* __LOCATION_DESCRIPTION_IDL__ */
```

The *Description* structure of the *Equipment* entity contains:

- *item_id* – Repository Item Id of the entity,
- *type_name* – a name of a type (in *Location Type Repository*)
- *property_list* – a list of properties.

Value_Description structure is equivalent to *Description*.
A *Location Description* entity supports the following attributes and operations:

- *descr* – get description of a given entity,
- *value_descr* – get description of a given entity,
- *item_id* – get Repository Item Id of the entity,
- *type_name* – get type name of the location description
- *property_list* – get the sequence of properties,

12.1.2.2 Equality of entities

Two entities are always considered to be different.

12.1.3 Repository

12.1.3.1 Interface description

```
#ifndef __LOCATION_DESCRIPTION_REP_IDL__
#define __LOCATION_DESCRIPTION_REP_IDL__ 1

interface Location_Description_Rep : Static_Data_Observed_R {

    typedef sequence<Location_Description> Location_Descriptions;

    Location_Descriptions find(
        in Location_Description::Pattern data_record_pattern)
```

```
            raises(Search_Not_Implemented);
        Location_Description::Value_Descriptions find_values(
            in Location_Description::Pattern data_record_pattern)
            raises(Search_Not_Implemented);
};

#endif                                  /* __LOCATION_DESCRIPTION_REP_IDL__ */
```

12.1.3.2 Relationships

The *Location Description Repository* observes the *Location Type Repository*. It attaches itself as an observer to all entities stored there. Whenever it receives a notification that a *Location Type* has been removed it also removes all *Location Description* entities that correspond to the removed *Location Description* object from its database. Whenever there is a change in corresponding *Location Type* object, the repository tries to adapt to the changes; if a property type is removed all property description is removed, and if it is added a new property description is created with a null value. The entries that has null properties should be updated as soon as possible with correct values.

12.1.4 Implementation classes

12.1.4.1 LDescr

The class implements interfaces of *Location_Description* and *Static_Data_Observed_E*. It encapsulates the following member fields:

- *orb* – the ORB reference,
- *ldescr_item_id* – the item id of the associated *Location Description* object,
- *corba_object* – the reference of the associated IDL generated object,
- *container* – the *Location Description Repository* containing this object,
- *description* – the description of the entity.

The class implements the following IDL operations:

- *descr* – get description of the entity,
- *value_descr* – get value description of the entity,
- *item_id* – get item id of the entity,
- *type_name* – get the type name of the entity,
- *property_list* – get the list of properties,
- *static_data_observer_list* – get the list of registered observers,

- *attach_static_data_observer* – register a new observer,
- *detach_static_data_observer* – detach an observer.

The remaining operations are:

- *LDescr* – constructors,
- *get_corba_object* – get the value of *corba_object*,
- *get_ldescr_item_id* – get the value of the *ldescr_item_id*,
- *set_item_id* – set *item_id*,
- *set_descr* – set *description*,
- *save* – save persistent state,
- *match_property* – return *true* if a specified property matches the specified property pattern – invoked by *match_ldescr* operation,
- *match_ldescr* – return *true* if the entity matches the specified location description pattern, invoked by the *find* and *find_values* methods of the *LDescr_Rep* class,
- *change_type* – check if the property list matches the property description list of the corresponding *Location Type* object; remove all properties that do not exist in the property description list. For all unmatched property descriptions create properties with a default null value. The method is invoked by the *update_type* operation of the *Equip_Rep* class,
- *check_record* – check if the properties of the new description match corresponding property descriptions of the associated *Location Type* object, return item id of this object,
- *equals* – return *true* if the entity is equal to the specified entity.

12.1.4.2 LDescr_Rep

The class *LDescr_Rep* implements interfaces of the *Location_Description_Rep*, *Location_Description_Rep_Manager* and *Static_Data_Observed_R*. The member fields of the class are:

- *Repository_DB_Name* – the name of the file storing the database,
- *manager_object* – the reference of the IDL generated manager object,
- *repository_object* – the reference of the IDL generated repository object,
- *observer_object* – the reference of the IDL generated observer object,
- *ldescr_list* – the hash table of all *LDescr* objects,
- *ldescr_deref* – the hash table that associates all item identfiers *item_id* with *LDescr* objects

- *item_ids_to_observers_list* – the hash table that binds *Location Description* item ids with the lists of the corresponding observers,
- *observers_to_item_ids_list* – the hash table that binds observers with the lists of the corresponding *Location Description* item ids,
- *add_static_data_observer_list* – the hash table that stores all observers interested with the notification of the addition of a new entity,
- *location_type_rep* – the reference of the *Location Type Repository*,
- *bootstrap_listener* – the *Bootstrap_listener_n* object, on which a callback from the server manager is to be invoked,
- *item_id* – the current item id, the one to be assigned for the next *Location Description* object.

The class implements the following IDL operations:

- *_notify* – handle the notified change of the *Location Type Repository*,
- *location_description_list* – get the list of entities,
- *add* – add the entity and notify the observers. The operation throws the *Invalid_Description* when an attempt is made to add the entity which description does not match type stored in *Location Type Repository*,
- *remove* – remove the specified entity,
- *update* – update the specified entity,
- *find* – find all entities that match the specified pattern,
- *find_values* – find value descriptions of all entities that match the specified pattern,
- *static_data_observer_list* – get all observers,
- *attach_static_data_observer* – attach an observer the specified entities,
- *detach_static_data_observer* – detach an observer form the specified entities,
- *add_static_data_observer_list* – get all observers interested in the *add* operation,
- *attach_add_static_data_observer* – attach the observer interested in the *add* operation,
- *detach_add_static_data_observer* – detach the observer interested in the *add* operation,
- *reference_resolved* – the *Location Type Repository* is up. Review the database and make it consistent with the *Location Type Repository*

The remaining methods are:

- *get_manager_object* – get the value of the *manager_object*,
- *get_repository_object* – get the value of the *repository_object*,
- *get_observer_object* – get the value of the *observer_object*,
- *static_data_observer_list* – get the list of observers of the specified item,
- *update_type* – calls the *change_type* operation of the *LDescr* object to adjust its description to the possibly new *Location Type* description. If the operation changes the entity (returns *true*) the observers of the given entity are notified and the persistent database is updated,
- *check_ldescr* – attach the repository as an observer to all entities of the *Location Type Repository*. Moreover, adjust the entities to the new state of the *Location Type Repository* – call *update_type* to update properties and remove all entities for which the corresponding *Location Type* object no longer exists,
- *location_type_change* – invoked by the *_notify* method when a notification of the new object removal or update is handled. It removes the entities whose the corresponding *Location Type* object has been removed or updates all entities for which the corresponding *Location Type* object has been updated,
- *rep* – get the value of the *repository_object*,
- *detach_dead_observers* – detach those observers that have been recognised as no longer active,
- *deref_object* – find the implementation object for the given CORBA object – invoked by *update* and *remove* methods,
- *get_item_ids_list* – get item ids of all entities observed by the specified observer,
- *inform_observers* – inform all observers of the given entity about the specified change,
- *check_unknown_item_ids* – check if all entities corresponding to the specified items are observed by some observer,
- *save* – save the persistent state,
- *restore* – restore the persistent state,

12.1.4.3 Bootstrap_base

The class is a wrapper for Naming Service.

- *Bootstrap_base* – a constructor, resolves reference of Naming Service,
- *incremental_bind* – binds an object to a name in Naming Service, any contexts required to bind the object are created if necessary,

- *resolve* – resolves a denoted name to a CORBA object,
- *file_to_object* – reads an *IOR* from a file and returns a CORBA object.

12.1.4.4 Bootstrap_listener_interface_n

The interface one should implement to be notified about server startups and shutdowns. It is a wrapper interface for *ABng_Server_Manager::Callback* interface intruduced for backward compability to the previous solution.

- *reference_resolved* – called when a server starts up, corresponds to *ABng_Server_Manager::Callback::server_startup* (see chapter 2)
- *server_shutdown* – called when a server shutts down, corresponds to *ABng_Server_Manager::Callback::server_shutdown* (see chapter 2)

12.1.4.5 Bootstrap_listener_n

The class implements *IDL* interface *ABng_Server_Manager::Callback* and forwards all callbacks to an associated object that implements *Bootstrap_listener_interface_n* interface.

The remaining methods are:

- *Bootstrap_listener_n* – a constructor
- *register_callback* – registers a callback for a server (see description of *ABng_Server_Manager::Register::register_callback* method)
- *register_context_callback* – see *ABng_Server_Manager::Register::register_context_callback*
- *unregister_callback* – see *ABng_Server_Manager::Register::unregister_callback*

12.1.4.6 Bootstrap_n

The wrapper class for *ABng_Server_Manager* interfaces, contains some helper utilities, all methods in this class are static.

- *init* – initializes the class, it should be called before any other mathods are used,
- *get_register_object* – returns a reference to *ABng_Server_Manager::Register* object in *server_manager* server,
- *register_server_in_NS* – registers a server in *server_manager* (and Naming Service)
- *unregister_server_from_NS* – unregisters a server from *server_manager* (and Naming Service),
- *concat_names* – concatenates two complex names (sequneces of strings),
- *equal_names* – checks if two complex names are equal,

- *name_in_context* – checks if a name is registered in a context,
- *CTX...* – methods returning names of context as arrays of strings,
- *SRV...* – methods returning names of server in Naming Context as arrays of strings.

12.1.4.7 bugfixes

There is a directory *LDescr/bugfixes* that contains classes responsible for translating Orbix *TypeCodes* to and from OMG *TypeCodes*. As all other servers are running *Orbix 2.2*, which uses Orbix *TypeCodes*, while the *OrbixWeb 3.0* uses OMG *TypeCodes*, it was necessary translate *TypeCodes*.

The directory contains also some patches for generated stubs and skeletons of the *Core* that calls translating routines in appropriate places.

12.1.4.8 LDescr_Main

The main class of the repository, implements static method *main* necessary for a *Java* application.

12.1.5 Persistency

Currently, the persistent state of the repository is stored in files. Values of *CORBA::Any* type are stored as sequence of hexadecimal octets that represent a *CDR* (Common Data Representation) of the value. Saving *CORBA::Any* was a primary factor of moving the server to Java, as for CORBA–Java mapping there is gurantee of portable serialization/deserialization of any value (including *CORBA::Any*), while for C++ mapping there is no such functionality.

12.1.6 CORBA services usage

Current implementation uses only Naming Service.

12.1.7 Life cycle management

The repository stores its reference in the naming service using ABng Server Manager. Currently it must be launched by hand.

12.2 Implementation Specification

12.2.1 Source code organization

- *location_description_rep/LDescr/LDescr.java* – implementation of the *LDescr* class,
- *location_description_rep/LDescr/LDescr_Rep.java* – implementation of the *LDescr_Rep* class,
- *location_description_rep/LDescr/LDescr_Main.java* – implementation of the *LDescr_Main* class.

- *location_description_rep/LDescr/Bootstrap_listner_interface_n.java* – interface *Bootstrap_listener_interface_n*.
- *location_description_rep/LDescr/Bootstrap_base.java* – implementation of the *Bootstrap_base* class.
- *location_description_rep/LDescr/Bootstrap_n.java* – implementation of the *Bootstrap_n* class.
- *location_description_rep/LDescr/Bootstrap_listner_n.java* – interface *Bootstrap_listener_n*.

Chapter 13

ABng Location Description Repository

13.1 Functional Specification

13.1.1 Functionality description

ABng Location Description is an entity that defines a location as a component of the ABng environment. It binds a *Location Description* with a sequence of sensors. The *ABng Location Description Repository* constitutes a database that stores *ABng Location Description* entities.

13.1.2 Entity

13.1.2.1 Interface description

```
#ifndef __ABNG_LOCATION_DESCRIPTION_IDL__
#define __ABNG_LOCATION_DESCRIPTION_IDL__ 1

interface ABng_Location_Description : System_Static_Data_Observed_E {

    struct Description {
        Domain_Repository::Repository_Item_Id item_id;
        Domain_Repository::Location_Description loc_descr;
        Sensors sensor_list;
    };

    typedef sequence<Description> Descriptions;

    struct Value_Description {
        Domain_Repository::Repository_Item_Id item_id;
        Domain_Repository::Location_Description::Value_Description
            loc_descr;
        Sensor::Value_Descriptions sensor_list;
    };

    typedef sequence<Value_Description> Value_Descriptions;

    struct Pattern {
```

```
        boolean is_any;
        Domain_Repository::Repository_Item_Id_Pattern item_id_pattern;
        Domain_Repository::Location_Description::Pattern
            loc_descr_pattern;
        Sensor::Sensors_Pattern sensor_list_pattern;
    };

    Description descr() raises(Domain_Repository::Destination_Unreachable);
    Value_Description value_descr()
        raises(Domain_Repository::Destination_Unreachable);

    readonly attribute Domain_Repository::Repository_Item_Id item_id;

    Domain_Repository::Location_Description loc_descr()
        raises(Domain_Repository::Destination_Unreachable);
    Sensors sensor_list() raises(Domain_Repository::Destination_Unreachable);

    void add(in Sensor new_sensor)
        raises(Domain_Repository::Duplicate_Data_Record,
            Domain_Repository::Unknown_Object_Ref);
    void remove(in Sensor sensor_to_remove)
        raises(Domain_Repository::Unknown_Object_Ref);
};

#endif                                  /* __ABNG_LOCATION_DESCRIPTION_IDL__ */
```

The *Description* structure of the *ABng Location Description* entity contains:

- *item_id* – Repository Item Id of the entity,
- *loc_descr* – a reference of the corresponding *Location Description* object,
- *sensor_list* – a sequence of references of sensors installed at the represented physical location.

Value_Description structure is composed of:

- *item_id* – Repository Item Id of the entity,
- *loc_descr* – a value description of the corresponding *Location Description* object,
- *sensor_list* – a sequence of value descriptions of sensors installed at the represented physical location.

A *ABng Location Description* entity supports the following attributes and operations:

- *descr* – get description of a given entity,
- *value_descr* – get value description of a given entity; this method necessitates obtaining value descriptions of the *Location Description* object and all *Sensor* objects corresponding to the given entity from the appropriate repositories; if any of these repositories cannot be contacted the exception is thrown,

- *item_id* – get Repository Item Id of the entity,
- *loc_descr* – get a reference of the corresponding *Location Description* object; the exception is thrown when the reference is nil,
- *sensor_list* – get the sequence of sensor references; if any of them is nil the exception is thrown,
- *add* – add the new sensor to the sensor list,
- *remove* – remove the sensor from the sensor list.

13.1.2.2 Equality of entities

Two entities are considered equal if they correspond to the same *Location Description* object, i.e. repository item ids of their corresponding *Location Description* objects are equal.

13.1.3 Repository

13.1.3.1 Interface description

```
#ifndef __ABNG_LOCATION_DESCRIPTION_REP_IDL__
#define __ABNG_LOCATION_DESCRIPTION_REP_IDL__ 1

interface ABng_Location_Description_Rep :
    ABng_Core_System::System_Static_Data_Observed_R {

    readonly attribute ABng_Core_System::ABng_Location_Descriptions
        abng_location_descriptions_list;

    ABng_Core_System::ABng_Location_Description add(
        in ABng_Core_System::ABng_Location_Description::Description
            data_record)
        raises(Domain_Repository::Duplicate_Data_Record,
               Domain_Repository::Invalid_Description);
    void remove(in ABng_Core_System::ABng_Location_Description
            object_to_remove)
        raises(Domain_Repository::Unknown_Object_Ref);
    ABng_Core_System::ABng_Location_Descriptions find(
        in ABng_Core_System::ABng_Location_Description::Pattern
            data_record_pattern)
        raises(Domain_Repository::Search_Not_Implemented,
               Domain_Repository::Destination_Unreachable);
    ABng_Core_System::ABng_Location_Description::Value_Descriptions
        find_values(
        in ABng_Core_System::ABng_Location_Description::Pattern
            data_record_pattern)
        raises(Domain_Repository::Search_Not_Implemented,
               Domain_Repository::Destination_Unreachable);
    void update(in ABng_Core_System::ABng_Location_Description
            object_to_update,
```

```
        in ABng_Core_System::ABng_Location_Description::Description
            data_record_pattern)
        raises(Domain_Repository::Unknown_Object_Ref,
               Domain_Repository::Duplicate_Data_Record,
               Domain_Repository::Invalid_Description);
};

#endif                          /* __ABNG_LOCATION_DESCRIPTION_REP_IDL__ */
```

13.1.3.2 Relationships

The *ABng Location Description Repository* observes the *Location Description Repository* and *Sensor Repository*. It attaches itself as an observer to all entities stores in these repositories. Whenever it receives a notification that a *Location Description* has been removed it also removes an *ABng Location Description* entity that corresponds to the removed *Location Description* object from its database. When a *Sensor* object is removed from the *Sensor Repository* it is also removed from all sensor lists of all *ABng Location Description*s associated with this sensor. The *ABng Location Description Repository* does not react to *Location Description* or *Sensor* updates.

13.1.4 Implementation classes

13.1.4.1 ABng_Loc_Descr_I

The class implements interfaces of *ABng_Location_Description* and *System_Static_Data_Observed_E*.

13.1.4.2 ABng_Loc_Descr_Rep_I

The class uses the building blocks of the repository template to implement functionality of *ABng_Location_Description_Rep* and *System_Static_Data_Observed_R* interfaces. It inherits from the *Basic_Rep_I<ALD_Types, SSDO_Types>* class, where the template parameter classes are defined as follows:

```
class SSDO_Types {

public:

  typedef ABng_Core_System::System_Static_Data_Observed_R Observed;
  typedef ABng_Core_System::System_Static_Data_Observer Observer;
  typedef ABng_Core_System::System_Static_Data_Observer_ptr Observer_ptr;
  typedef ABng_Core_System::System_Static_Data_Observed_E
          ::System_Static_Data_Observers Observers;
};

class ALD_Types {

public:

  typedef ABng_Core_System::ABng_Location_Description Item;
```

```
    typedef ABng_Core_System::ABng_Location_Description_ptr Item_ptr;
    typedef ABng_Core_System::ABng_Location_Descriptions Items;
    typedef ABng_Loc_Descr_I Item_I;
    typedef ABng_Core_System::ABng_Location_Description::Description
                Initial_Data;
    typedef ABng_Core_System::System_Static_Data_Observer
            ::System_Static_Data_Object Data_Object;
};
```

The method *validate_update_add_data* implements the requirement that no two *ABng Location Descriptions* corresponding to the same *Location Type* (indirectly, via *Location Descriptions*) possess the same sensor in their sensor lists. This method is invoked by the *add* method of *ABng_Location_Description* interface and by modified functions:

- *validate_add_data* – apart from the default requirement that no two equal objects may by present in the database it introduces also the rule implemented by the *validate_update_add-_data* method.
- *validate_update_data* – it implements the same requirements as the *validate_add_data* method for the *update* operation.
- *private_find* – invokes the function matching entities with the specified pattern. Depending on a pattern this method may be forced to cooperate with the *Location Description Repository* and *Sensor Repository*. However, as much is checked locally as possible. Moreover, a cache local to the matching method has been applied in order to remove recurrence of remote finds with the same pattern.

13.1.4.3 Location_Descr_Observer_I

This class implements the functionality of the *Static_Data_Observer* for a *Location Description Repository* observer. It reacts to the change of the observed repository in the following way:

1. if a *Location_Description* object has been added it attaches itself as its observer,
2. if a *Location_Description* object has been removed it removes all entities from the *ABng Location Description Repository* that correspond to the removed entity,
3. if a *Location_Description* object has been updated it does not react.

13.1.4.4 Sensor_Observer_I

This class implements the functionality of the *System_Static_Data_Observer* for a *Sensor Repository* observer. It reacts to the change of the observed repository in the following way:

1. if a *Sensor* object has been added it attaches itself as its observer,
2. if a *Sensor* object has been removed it removes its reference from all entities of the *ABng Location Description Repository* that hold it in their sensor lists,
3. if a *Sensor* object has been updated it does not react.

13.1.4.5 ABng_LD_Rep_Server

This is the repository server implementation class based on the template of the repository server.

- *create_objects* – creates instances of the *ABng_Loc_Descr_Rep_I*, *Sensor_Observer_I* and *Location_Descr_Observer_I* classes and calls *init* on the *ABng_Loc_Descr_Rep_I* instance.
- *delete_objects* – deletes the instances of the created objects.

13.1.5 Persistency

Currently, the persistent state of the repository is stored in files. Objects included in the entity description are stored in a form of their repository item ids. When a persistent state is revoked item ids are read from the file and cached. When a callback from the server manager comes saying that either *Location Description Repository* or *Sensor Repository* is active the *resolve_references* method restores the consistent state of the data base and makes observer objects register themselves. If it happens that during the inactivity of the repository the state of *Location Description Repository* or *Sensor Repository* has changed the same rules are applied as those used by observers in case of *add-update-remove* notification.

13.1.6 CORBA services usage

Current implementation uses only naming service.

13.1.7 Life cycle management

The repository stores its reference in the naming service using ABng Server Manager. Currently it must be launched by hand.

13.2 Implementation Specification

13.2.1 Source code organization

- *abng_location_description_rep/abng_location_descr_i.h*
  abng_location_description_rep/abng_location_descr_i.cc –
  implementation of the *ABng_Loc_Descr_I* class,
- *abng_location_description_rep/abng_location_descr_rep_i.h*
  abng_location_description_rep/abng_location_descr_rep_i.h –
  implementation of the *ABng_Loc_Descr_Rep_I* class,
- *abng_location_description_rep/location_descr_observer.h*
  abng_location_description_rep/location_descr_observer.cc –
  implementation of the *Location_Descr_Observer_I* class,

- *abng_location_description_rep/sensor_observer.h*
  abng_location_description_rep/sensor_observer.cc –
  implementation of the *Sensor_Observer_I* class,

- *abng_location_description_rep/abng_ld_rep_server.h*
  abng_location_description_rep/abng_ld_rep_server.cc –
  implementation of the *ABng_LD_Rep_Server* class,

- *abng_location_description_rep/abng_location_descr_srv.cc* –
  implementation of the main method.

Chapter 14

Main Sighting Processor

14.1 Functional Specification

14.1.1 Functionality description

Main Sighting Processor (MSP) is responsible for gathering messages from all sensors installed in an office and from all badges associated with locatable objects, and dispatching these messages to View Sighting Processors (VSPs). The MSP is an observer of all pollers active in the system. For every message received from a poller the MSP selects VSPs which might be interested in that message. This is performed using a list of badges and a list of sensors which are associated with a view corresponding to the VSP. If a source sensor (in a case of sensor messages) or both a source sensor and a source badge (in a case of badge messages) belong to a view, that message is forwarded to the VSP corresponding to that view.

Lists of badges and sensors are maintained by the MSP for every view. When a view is instantiated it is responsible for registering itself within the MSP. A view passes a list of badges associated with its locatable objects and a list of sensors associated with its location objects to the MSP. Additionally, a view passes its name, a reference of its VSP and a flag which informs the MSP whether the view is interested in duplicated badge messages or not. A badge message received by a sensor and forwarded to a poller and then to the MSP is considered duplicated if it was earlier captured by another sensor belonging to the view.

When a view is updated, i.e. a new badge or a new sensor is added to the view, the view notifies the MSP about a change so that the MSP would maintain the current mapping of sensors and badges to views.

14.1.2 Interfaces description

The MSP implements two IDL interfaces: *Main_Sighting_Processor* and *AB_Observer*. The first one allows views to register themselves within the MSP and to update badge-view and sensor-view mapping dictionaries which are maintained by the MSP. The latter one is used for passing sighting messages from pollers to the MSP.

The *Main_Sighting_Processor* interface is as follows.

```
#ifndef __MAIN_SIGHTING_PROCESSOR_IDL__
#define __MAIN_SIGHTING_PROCESSOR_IDL__ 1

interface Main_Sighting_Processor {

    typedef sequence<View_Sighting_Processor> View_Sighting_Processors;

    readonly attribute ABng_Domain_Internals my_domain;
    readonly attribute View_Sighting_Processors view_sp_list;

    void add_vsp(in string view_name,
        in ABng_Core_Internals::View_Sighting_Processor vsp,
        in ABng_Core_System::Badge_Ids badge_id_list,
        in ABng_Core_System::Sensor_Ids sensor_id_list,
        in boolean wants_duplicates)
        raises(Domain_Repository::Unknown_Object_Ref,
            Domain_Repository::Duplicate_Data_Record);
    void find_vsp(in string view_name,
        out ABng_Core_Internals::View_Sighting_Processor vsp,
        out ABng_Core_System::Badge_Ids badge_id_list,
        out ABng_Core_System::Sensor_Ids sensor_id_list,
        out boolean wants_duplicates)
        raises(ABng_Core::Unknown_View_Name);
    void remove_vsp(in string view_name)
        raises(ABng_Core::Unknown_View_Name);
    void add_badges(in string view_name,
        in ABng_Core_System::Badge_Ids badge_id_list)
        raises(ABng_Core::Unknown_View_Name);
    void remove_badges(in string view_name ,
        in ABng_Core_System::Badge_Ids badge_id_list)
        raises(ABng_Core::Unknown_View_Name,
            ABng_Core_System::Unknown_Badge_Id);
    void add_sensors(in string view_name,
        in ABng_Core_System::Sensor_Ids sensor_id_list)
        raises(ABng_Core::Unknown_View_Name);
    void remove_sensors(in string view_name,
        in ABng_Core_System::Sensor_Ids sensor_id_list)
        raises(ABng_Core::Unknown_View_Name,
            ABng_Core_System::Unknown_Sensor_Id);
};

#endif                                   /* __MAIN_SIGHTING_PROCESSOR_IDL__ */
```

- *view_sp_list* – a sequence containing references of all currently active VSP objects.
- *add_vsp* – registers a new VSP. Allows a VSP to pass a name of its view, its reference, a list of badges and sensors associated with the view, and information whether the view wants to receive duplicated badge messages or not.
- *find_vsp* – looks for a VSP corresponding to the view with the given name. If a VSP is found the data passed during its registration are returned in output parameters.

- *remove_vsp* – removes a VSP corresponding to the view with the given name. Can be invoked by a VSP just before it is going down.

- *add_badges* – enables a VSP to add new badges to the list of badges associated with a VSP and maintained by the MSP. Note that no exception is raised when some badges in a parameter already exist in the a list of badges. Such an event is just ignored. Only not duplicated badges are inserted into the list of badges.

- *remove_badges* – enables a VSP to remove some badges from a list of badges associated with a VSP and maintained by the MSP. Note that no exception is raised when some badges in a parameter do not exist in the a list of badges associated with the VSP. Such an event is just ignored.

- *add_sensors* – enables a VSP to add new sensors to the list of sensors associated with a VSP and maintained by the MSP. Note that no exception is raised when some sensors in a parameter already exist in the a list of sensors. Such an event is just ignored. Only not duplicated sensors are inserted into the list of sensors.

- *remove_sensors* – enables a VSP to remove some sensors from a list of sensors associated with the VSP and maintained by the MSP. Note that no exception is raised when some sensors in a parameter do not exist in the a list of sensors associated with the VSP. Such an event is just ignored.

The *AB_Observer* interface is used by pollers to transmit sighting messages to the MSP. When the MSP starts it registers a reference of *AB_Observer* within all pollers active in a system. Their references are retrieved from the Naming Service. Since the MSP is interested only in sighting messages (i.e. badge messages which are periodically emitted by a badge every 10 seconds and messages triggered by a badge button pressing) other badge and sensor messages are ignored. Therefore all operations but *sighting* of the *AB_Observer* interface have empty implementations and they are normally not invoked as the MSP registers its interest in sightings only. The *sighting* operation is described below.

- *sighting* – it is invoked by a poller to pass a sighting message to the MSP. Since in the system there may be several active pollers and tens or hundreds of badges, this operation will be typically called tens times per second. For every received sighting the *ABng_Core_Internals::Internal_Sighting_Description* structure is constructed and filled with information from the sighting message. Afterwards, a list of registered VSPs is iterated. For every registered VSP, the MSP checks whether both a badge and a sensor involved in a sighting belong to the view associated with the VSP. If not, the VSP will not be notified about a new sighting. Otherwise, the MSP checks whether a VSP wants to receive duplicated sightings. If so, the structure containing sighting information is immediately transmitted to the VSP via invocation of its *process_sighting* operation. Otherwise, the MSP checks whether the sighting is duplicated or not. If so, it is ignored for that VSP, otherwise its is transmitted to the VSP. To determine whether the sighting is duplicated or not, the MSP uses a sequence number carried in every sighting message. For every badge the MSP remembers a sequence number of the last sighting emitted by that badge. When a sequence number of a new message is equal to the number of the previous message, the new one is considered as being duplicated. Duplicated sightings appear if an infra-red sighting message emitted by a badge is received by a couple of sensors. Since different sets of badges and sensors can belong to particular views, the MSP has to determine whether a sighting is duplicated or not for every view individually.

14.1.3 Relationships

The MSP depends on two types of ABng components: pollers and VSPs. The MSP registers itself in the Server Manager as an observer of all pollers. When a new poller is launched, the MSP is notified about it and registers a reference of the *AB_Observer* interface within a new poller. Afterwards, the MSP will receive sightings captured by the poller.

Every view registers itself in the Server Manager as an observer of the MSP. When the MSP is launched the view is informed about it and registers its VSP within the MSP passing its attributes (name, badges, sensors, preferences related to duplicated messages). Afterwards, sighting messages are forwarded to the VSP.

14.1.4 Implementation classes description

The implementation of the MSP component consists of three C++ classes:

- *Comparable_Badge_Id* – extends the *ABng_Core_System::Badge_Id* type by adding an equality operator which enables to compare two badge identifiers and a hash function.

- *Comparable_Sensor_Id* – extends the *ABng_Core_System::Sensor_Id* type by adding an equality operator which enables to compare two sensor identifiers and a hash function.

- *VSP_Entry* – objects of this class are entries in a registry of VSPs. A *VSP_Entry* object is created when a new VSP registers itself within the MSP. It holds the VSP attributes (VSP reference, view name, badge and sensor list, *wants_duplicates* flag). Badge and sensor lists are implemented as hash sets containing objects of *Comparable_Badge_Id* and *Comparable_Sensor_Id* classes. A *VSP_Entry* also maintains a hash dictionary which maps badge identifiers into sequence numbers of last messages emitted by badges belonging to the view. This dictionary is used to discover duplicated sightings.

- *MSP_I* – implements the IDL interfaces provided by the MSP. It maintains a dictionary which maps view names into *VSP_Entry* objects. This dictionary constitutes the registry of VSPs.

- *MSP_Server* – provides building blocks for a server containining the MSP component. This class is derived from *common/Basic_Server*.

14.1.5 Interfaces to implementation classes mapping

Both *Main_Sighting_Processor* and *AB_Observer* interfaces are implemented by the *MSP_I* C++ class.

14.1.6 Persistency

The MSP component does not maintain any persistent state.

14.1.7 Event notification

A VSP registered within the MSP is triggered by the MSP after the MSP receives a sighting in which a given VSP is interested.

14.1.8 Configuration and property management

The MSP does not require any special configuration and property management.

14.1.9 Concurrency control and locking

Currently, the MSP is a single threaded component.

14.1.10 Security

No security mechanisms are currently supported by the MSP. Every IIOP client which obtains a reference of one of two MSP's interfaces can communicate with it. Additionally, someone can register a false poller in the Naming Service and to supply the MSP with fake sightings.

14.1.11 Versioning

It is possible to build two different versions of the MSP component: a debugging one and a release version. To build a debugging version please include `-DDEBUG` to `CXXDEBUGFLAGS` in Imakefile. For release version please remove the flag.

14.1.12 CORBA services usage

The MSP uses CORBA Naming Service via Server Manager.

14.1.13 Life cycle management

The MSP is instantiated when the MSP server, called *abngmsp* is launched. It is destroyed when *abngmsp* terminates.

14.2 Implementation Specification

14.2.1 Source code organisation

The source files of the MSP components are as follows:

- *sighting_processing/comparable_badge_id.[h|cc]* – the *Comparable_Badge_Id* class,
- *sighting_processing/comparable_sensor_id.[h|cc]* – the *Comparable_Sensor_Id* class,
- *sighting_processing/msp_i.[h|cc]* – the *MSP_I* class,
- *sighting_processing/msp_server.[h|cc]* – the *MSP_Server* class,
- *sighting_processing/msp_srv.[h|cc]* – the mainline of a server containing the MSP component.

14.2.2 Imakefile description

The *Imakefile* for compilation and linkage of the MSP and view components and their servers is located in the *servers/sighting_processors* directory.

14.3 Server process structure

The *abngmsp* server contains contains the single MSP component. In the system there should be only one instance of this server active.

14.4 Runtime environment specification

No ABng components have to be active when *abngmsp* is launched. The required environment setting is the same as for any other ABng servers.

14.5 Server life cycle

The server is intended to be up and running continuously. After the server start-up, MSP dynamically registers itself in currently active pollers. Due to the Server Manager new pollers can be dynamically discovered when the MSP is working. When a poller crashes and recovers after that the MSP is informed about it and it can rebind to the new instance of a poller.

The MSP's start-ups are monitored by view components which registers themselves within the MSP after it is launched. The MSP does not save its VSPs registry in any persistent storage. After a crash it is gradually recovered via subsequent registration of view components which discover a new instance of the MSP.

14.6 Server installation and starting

The *abngmsp* server requires no arguments and is started as follows:

```
abngmsp
```

Chapter 15

View

15.1 Functional Specification

15.1.1 Functionality description

A view is the most sophisticated component of the ABng Core and it implements the crucial concepts of the ABng system. It is a complex module which provides several IDL interfaces.

From the logical point of view, a view component represents a subset of objects which can be localised in an office environment and a subset of locations which have been differentiated in this environment. In a system, many views can be instantiated. Each of them can represent different subsets of locations and locatables providing users with different abstract views at localisation information. A location is represented by a *Location* object within a view. A *Location* object is associated with an *ABng_Location_Description* object describing sensors installed in a *Location* and with a *Location_Description* object describing properties of the location, such as a room number or a room name. A locatable is represented by a *Locatable* object within a view. A *Locatable* object is associated with a *User* object or an *Equipment* object which describe a user or a piece of equipment. Additionally a *Locatable* is associated with a *Badge_Holder* object which keeps mapping between a user or a piece of equipment and a badge.

A view enables to maintain subsets of *Locations* and *Locatables*, hence, it acts as a repository of *Locations* and as a repository of *Locatables*. Users' applications can browse lists of *Locations* and *Locatables* and management applications can both add and remove *Locations* and *Locatables*. Applications can also register themselves within the view to monitor adding and removing *Locations* or *Locatables*.

On the other hand, a view is responsible for processing of localisation data, i.e. sighting messages which are provided by the Main Sighting Processor (MSP). A result of the processing of a sighting message is the updating of the maintained relationships between *Locations* and *Locatables* and notification of applications about these changes. A *Location* and a *Locatable* are related to each other if a *Locatable* is currently seen in the *Location* or the given *Location* is the last one in which the *Locatable* was seen. Attributes of this relationships are values carried in the last sighting message which was emitted by a badge associated with the *Locatable* and received in the *Location* (i.e. by one of its sensors).

Sightings are processed by a two-level hierarchy of sighting processors which are contained within a view. The root of this hierarchy is the View Sighting Processor (VSP). The leafs are Location Sighting Processors (LSPs) and LocatablE Sighting Processors (LESPs). Every LSP is associated with a *Location* object whereas every LESP corresponds to one *Locatable*. An LSP holds a list of LESPs corresponding to *Locatables* which are related to the *Location*. A LESP holds attributes of the last sighting emitted by its badge and a reference of an LSP associated with the last *Location* of the *Locatable* represented by the LESP.

After activation of the view, its VSP registers itself within the MSP passing a VSP reference, view name, a list of badges associated with its *Locatables*, a list of sensors installed in its *Locations*, and information whether a view wants to be notified about duplicated messages or not (see Section 14.1 for more details).

For every received sighting, the VSP determines LSPs and LESPs associated with *Locations* and *Locatables* involved in a sighting and a kind of a change. This is performed using sensor and badge identifiers carried in the sighting message and dictionaries maintained by the VSP. These dictionaries map badge identifiers into *Locatables* and sensor identifiers into *Locations*.

The VSP distinguishes the following cases:

1. The *Locatable* changed a *Location* within the view and both the new and the old *Location* belong to the view.

2. The *Locatable* has never been seen in the view and just entered one of its *Locations*.

3. The *Locatable* did not change the *Location*.

In case 1 the LESP corresponding to the *Locatable* involved in the change is removed from the LESP list maintained by the LSP associated with the old *Location*.

In cases 1 and 2 the LESP corresponding to the *Locatable* involved in the change is inserted into the LESP list maintained by the LSP associated with the current *Location*. A reference of the related LSP is updated in the LESP.

In any case current sighting attributes are set in the LESP corresponding to the *Locatable* involved in the change.

The last step of sighting processing is notification of applications which observe changes of dynamic locatlisation information. An application can register its interest in localisation data related to all or selected individual *Locatables* or *Locations* belonging to the view. This can be performed using the interface of the view or interfaces of individual *Locations* or *Locatables*. Any way of registration results in updating a registry of dynamic data observers maintained by the view for all *Locations* and *Locatables*.

A view also deals with commands which are send by applications to *Locations* and *Locatables*, and, as a consequence, to sensors and badges associated with them. When an application issues a command to a *Location* or a *Locatable*, the view maps the target object into a set of sensors or badges. Afterwards, it forwards the command to the scheduler passing a corresponding list of sensors or badges and other attributes received from the application. The scheduler gives back an identifier of the request representing the command. If the application wishes to be notified about a command execution result it submits a reference of a call-back object. The view remembers mapping between request identifier and the application's call-back object.

When a reply arrives from the scheduler it carries the request identifier. It is mapped to a application's call-back object (if it was submitted). The application is notified about a result of a command execution.

15.1.2 Interfaces description

The view implements the following IDL interfaces:

- *View* – this interface allows users' applications to:
  - browse lists of *Locatables* and *Locations* associated with the view,
  - retrieve sighting data related to all *Locatables*,
  - broadcast commands to all sensors and badges associated with *Locatables* and *Locatables* of the view,
  - register their interest in additions and removals of selected *Locations* and *Locatables*,
  - register their interest in localisation information related to selected *Locations* and *Locatables*.
- *View_Manager* – this interface allows management applications to add and remove new *Locations* and *Locatables*, i.e. to maintain repositories of *Locations* and *Locatables*.
- *View_Sighting_Processor* – is used by the MSP to transmit sighting data to the view and its VSP.
- *Location* – this is the interface of a *Location* object. It allows an application to:
  - retrieve information about association of the *Location* object and the *Location_Description* object,
  - retrieve sighting data related to *Locatables* seen in the *Location*,
  - register itself in the event of *Location* removal,
  - register its interest in sighting data related to the *Location*,
  - broadcast commands to all sensors installed in the *Location* and all badges seen in the *Location*.
- *Locatable* – this is the interface of a *Locatable* object. It allows an application to:
  - retrieve information about association of the *Locatable* object and a *User* or *Equipment* object,
  - retrieve sighting data related to the *Locatable*,
  - register itself in the event of *Location* removal,
  - register its interest in sighting data related to the *Locatable*,
  - send commands to the badge associated with the *Locatable*.

- *Scheduler_Caller* – it is used by the scheduler to transmit replies to commands issued by the view on other applications' demand.

The complete specification of IDL interfaces implemented by a view can be found in [3]

15.1.3 Relationships

The components which a view depends on are:

- *ABng_Location_Description_Rep* – a view observes removals and changes of *ABng_Location_-Description* objects which are related to view's *Locations*. If an *ABng_Location_Description* is removed a corresponding *Location* is also removed from the view. If a sensor list maintained by an *ABng_Location_Description* changes this change is reflected by a view's dictionary which maps sensors to *Locations*. If a reference of a *Location_Description* associated with an *ABng_Location_Description* changes this change is propagated to the related *Location* and reported to observers of that *Location*.

- *Sensor_Rep* – a view observes updates of identifiers of sensors associated with *Locations* and updates a dictionary mapping sensors into *Locations*.

- *Badge_Holder_Rep* – a view observes removals and changes of *Badge_Holder* objects which are related to view's *Locatables*. If a *Badge_Holder* is removed a corresponding *Locatable* is also removed from the view. If a badge associated with the *Badge_Holder* changes this change is reflected by a view's dictionary which maps badges to *Locations*. If a reference of a *User* or *Equipment* object associated with a *Badge_Holder* changes this change is propagated to the related *Locatable* and reported to observers of that *Locatable*.

- *Badge_Rep* – a view observes updates of identifiers of badges associated with *Locatables* and updates a dictionary mapping badges into *Locatables*.

- *MSP* – each time a view discovers the MSP it registers itself with the MSP. The MSP is discovered after view starts or after the MSP recovers from the crash.

- *Scheduler* – a view forwards application's commands to the scheduler and receives replies from the scheduler.

15.1.4 Implementation classes description

The implementation of the view component consists of the following C++ classes:

- *Comparable_Badge_Id* – extends the *ABng_Core_System::Badge_Id* type by adding a hash function and an equality operator which enables to compare two badge identifiers.

- *Comparable_Badge_Descr* – extends the *ABng_Core_System::Badge::Description* type by adding a hash function and an equality operator which enables to compare two badge descriptions.

- *Comparable_Sensor_Id* – extends the *ABng_Core_System::Sensor_Id* type by adding a hash function and an equality operator which enables to compare two sensor identifiers.
- *Comparable_Sensor_Descr* – extends the *ABng_Core_System::Sensor::Description* type by adding a hash function and an equality operator which enables to compare two sensor descriptions.
- *Location_I* – implements the *Location* interface and the *Location* component.
- *Location_Rep_I* – implements a location repository part of the view. This class derived from the *Basic_Rep* template described in chapter 4.
- *Locatable_I* – implements the *Locatable* interface and the *Locatable* component.
- *Locatable_Rep_I* – implements a locatable repository part of the view. This class derived from the *Basic_Rep* template described in chapter 4.
- *View_I* – implements the *View*, *View_Manager* and *View_Sighting_Processor* interfaces.
- *Request_Mapper* – maps request identifiers assigned by the scheduler into call-back objects provided by applications sending commands to *Locations* or *Locatables*..
- *View_Server* – provides the basic building blocks for an implementation of a server containing the view component.

15.1.5 Persistency

View data which are stored in three files:

- *Location_Rep.db* – in this file persistent states of *Location* objects are stored. For every *Location*, its item identifier and an item identifier of the corresponding *ABng_Location_Description* is stored. A persistent image of all *Locations* is saved in the file after removing, adding or updating any *Location*.
- *Locatable_Rep.db* – in this file persistent states of *Locatable* objects are stored. For every *Locatable*, its item identifier, an item identifier of the corresponding *Badge_Holder* is stored and the attributes of the last sighting message issued by the *Locatable* (its badge) are stored. A persistent image of all *Locatables* is saved in the file after removing, adding or updating any *Locatable*. Additionally, to ensure persistency of sighting data, *Locatables* are periodically saved every 5 minutes. This time interval can be change using the *-p* option of the view server command.
- *View.db* – this file stores a view name and a value of the flag which informs whether the view is interested in receiving duplicated messages or not. This file is created and saved when the view is launched for the first time and its contents is never changed later on.

Since there might be several views active in the system, for each of them a separate subdirectory is created in the ABng database directory when the view with the given name (which has to be unique) is started for the first time (cold start). All three database files are saved there.

15.1.6 Event notification

The view plays a key role in notification of ABng applications about changes of location of monitored *Locatable* objects. Additionally, it may notify interested applications about adding and removing *Locations* and *Locatables*.

15.1.7 Configuration and property management

The MSP does not require any special configuration and property management.

15.1.8 Concurrency control and locking

Currently, the MSP is a single-threaded component.

15.1.9 Security

No security mechanisms are currently supported by the view.

15.1.10 Versions

It is possible to build two different versions of the view component: a debugging one and a release version. To build a debugging version please include `-DDEBUG` to `CXXDEBUGFLAGS` in Imakefile. For release version please remove the flag.

15.1.11 CORBA services usage

The view uses CORBA Naming Service via Server Manager.

15.1.12 Life cycle management

The view is instantiated when the view server, called *abngview* is launched. It is destroyed when *abngview* terminates.

15.2 Implementation Specification

labelsec:viewimpl

15.2.1 Source code organisation

The source files of the MSP components are as follows:

- *sighting_processing/comparable_badge_id.[h|cc]* – the *Comparable_Badge_Id* class,
- *sighting_processing/comparable_badge_descr.[h|cc]* – the *Comparable_Badge_Descr* class,

- *sighting_processing/comparable_sensor_id.[h|cc]* – the *Comparable_Sensor_Id* class,
- *sighting_processing/comparable_sensor_descr.[h|cc]* – the *Comparable_Sensor_Descr* class,
- *sighting_processing/request_mapper.[h|cc]* – the *Request_Mapper* class,
- *sighting_processing/location_i.[h|cc]* – the *Location_I* class,
- *sighting_processing/locatable_i.[h|cc]* – the *Locatable_I* class,
- *sighting_processing/view_i.[h|cc]* – the *View_I* class,
- *sighting_processing/view_server.[h|cc]* – the *view_Server* class,
- *sighting_processing/msp_srv.[h|cc]* – the mainline of a server containing the view component.

15.2.2 Imakefile description

The *Imakefile* for compilation and linkage of both MSP and view components and their servers is located in the *servers/sighting_processors* directory.

15.3 Server process structure

The *abngview* server contains the single view component. In the system there may be many instances of this server active, however each of them should have a different view name passed as a server argument.

15.4 Runtime environment specification

No ABng components have to be active when *abngview* is launched. The required environment setting is the same as for any other ABng servers.

15.5 Server life cycle

The server is intended to be up and running continuously.

15.6 Server installation and starting

The syntax of the *abngview* command follows.

```
Usage:   abngview [options] view_name
         abngview -s [options] view_name type_name
The following options may be used:
-d directory      Specify the database directory.
-p period         Specify a number of minutes to wait for sightings dump
```

	(default: 5 minutes)
-s	Used when the server is launched for the first time (cold start). When this option is used the type name of locations belonging to the view must be given as the type_name parameter.
-t timeout	Specify a number of milliseconds to wait between Orbix events.
-w	Require to pass duplicated locatable messages to the view. Can be only used in conjunction with -s.

If a view with the given name is started for the first time or view's database files have been removed or destroyed for any reason, the *abngview* server should launched with *-s* option. This causes the view to create and initialise its database files.

Chapter 16

Client Template

Although the elements stored by 9 ABng System repositories differ in type and sometimes functionality, the operations which may be invoked on repositories are the same: find, find_values, add, update and remove and except for differing parameter types their signatures are the same. Therefore, a command line client interfaces to these repositories follow the same pattern of implementation and execution. This lead to the idea of building a client template above which client interfaces for different repositories may be built. The template must be however extendible, to allow creating clients supporting not only the five aforementioned functions but also the commander interfaces of repository entities if such interfaces exist.

The Repository Client Template consists of the following C++ classes:

- *Basic_Client* – the class providing basic building blocks and default implementation for an ABng repository handle with a basic functionality,
- *Rep_Client* – the class providing basic building blocks for implementation of the mainline of an ABng repository client.

The remaining part of this chapter describes the above elements in details.

16.1 Basic_Client

The *Basic_Client* C++ template class provides a basis for implementation of ABng CORBA clients. Its role is to provide client components with references of ORB and BOA. It also implements a method which sets the default ABng configuration of CORBA (Orbix) clients. Moreover, it includes methods that allow obtaining an entity reference based on its repository item identifier.

16.1.1 Class Definition

```
typedef ABng_Server_Manager::Name (*resolve_fun)();

template<class Repository_Types>
class Basic_Client {
```

```
public:

  typedef Repository_Types::Item Item;
  typedef Repository_Types::Item_ptr Item_ptr;
  typedef Repository_Types::Items Items;
  typedef Repository_Types::Initial_Data Initial_Data;
  typedef Repository_Types::Item::Description Description;
  typedef Repository_Types::Item::Pattern Pattern;
  typedef Repository_Types::Item::Value_Description Value_Description;
  typedef Repository_Types::Item::Value_Descriptions Value_Descriptions;
  typedef Repository_Types::Repository Repository;
  typedef Repository_Types::Repository_ptr Repository_ptr;


  Basic_Client(resolve_fun _srv_rep) :
    srv_rep(_srv_rep) {}

  virtual ~Basic_Client() {}

  virtual void init(int argc, char **argv);
  virtual void init_rep_ptr();

  virtual Item_ptr init_item_ptr(char* data1);
  virtual Pattern init_pattern();

  virtual Pattern
    init_pattern_id(Domain_Repository::Repository_Item_Id id);

  virtual CORBA::ORB_ptr get_orb() {return orb_ptr;}
  virtual CORBA::BOA_ptr get_boa() {return boa_ptr;}

protected:

  CORBA::ORB_ptr orb_ptr;
  CORBA::BOA_ptr boa_ptr;

  Repository_ptr my_rep;

  resolve_fun srv_rep;

};
```

16.1.2 Template parameters

The *Basic_Client* template is parameterized by a single parameter – *Repository_Types* class. Within this class the following typedef definitions should be specified:

- *Item* – the IDL interface implemented by items stored in the repository,
- *Item_ptr* – an item reference type,

- *Items* – a sequence of item references,
- *Initial_Data* – a data type used for item object initialization,
- *Repository* – the IDL interface provided by the repository,
- *Repository_ptr* – a repository reference type.

Moreover, the class should contain a static method *init_pattern* which creates and initializes a *Pattern* object appropriate for a given *Item*. The pattern should have all *is_any* fields set to *true*.

The definition of *Repository_Types* classes for all repositories of the *Domain_Repository* module follows the same scheme. This applies also to repositories of the *ABng_Core_Management* module. Therefore, frames for definition of these classes for both *Domain_Repository* and *ABng-_Core_Management* modules has been prepared in a form of macro-definitions. The macro for the *Domain_Repository* module, *DEF_TYPE*, defines only the types specified above. On the contrary, the macro for the *ABng_Core_Management* module, *DEF_TYPE_ABNG* contains also two additional definitions: *Repository_Manager* and *Repository_Manager_ptr*, which denote the IDL interface of the repository manager (an object which supports *add*, *remove* and *update* operations) and its reference type correspondingly. Obviously, in case of the *ABng_Core_Management* module these definitions are equivalent to *Repository* and *Repository_ptr*. It will be explained in the next section why they have been introduced to the *DEF_TYPE_ABNG* macro.

16.1.3 Attribute Description

- *orb_ptr* – an ORB reference,
- *boa_ptr* – a BOA reference,
- *my_rep* – repository reference,
- *srv_rep* – pointer to an appropriate *ABng_Server_Manager* method that builds a compound name bound with *my_rep* in Naming Service,

16.1.4 Method Description

- *init_rep_ptr* – initializes *my_rep* field,
- *init* – initializes ORB and BOA references as well as *my_rep* field,
- *init_pattern* – initializes pattern; all *is_any*s are set to *true*,
- *init_pattern_id* – initializes pattern; all *is_any*s except that of *item_id* field are set to *true*; the value of *item_id* field is set to the specified number,
- *init_item_ptr* – based on an item identifier or an item reference finds its reference in the repository.

16.2 Rep_Client

The *Rep_Client* class provides a basis for implementation of a client cooperating with a single ABng repository. Its main role is to simplify the construction of a client. The template provides methods that facilitate interpreting command line arguments and contacting the repository. By default the template supports all basic repository operations (*find*, *find_values*, *add*, *update* and *remove*). Other methods may be added by properly defining a static member, *operations*, and overloading *other_operations* method.

16.2.1 Class Definition

```
#include "basic_client.h"

template<class Repository_Types>
class Rep_Client : public Basic_Client<Repository_Types> {

public:

  typedef Repository_Types::Repository_Manager Repository_Manager;
  typedef Repository_Types::Repository_Manager_ptr Repository_Manager_ptr;

  Rep_Client(resolve_fun f, resolve_fun f_mgr)
    : Basic_Client<Repository_Types>(f), srv_rep_mgr(f_mgr) {}

  virtual ~Rep_Client() {
      if(operation) delete operation;
      if(prog_name) delete prog_name;
  }

  typedef char* Parameters[20];

  struct _Operation {
          char* name;
          Parameters parameters;
  };

  typedef _Operation Operations[20];

  static Operations operations;
  static char* description;

  virtual void init(int argc, char **argv);
  virtual void init_fields(int argc, char **argv);
  virtual void init_rep_ptr();

  virtual void run();
  virtual void get_options(int argc, char **argv);
  virtual void find();
```

```
    virtual void find_values();
    virtual void add(char* data1);
    virtual void remove(char* data1);
    virtual void update(char* data2, char* data1);

    virtual void other_operations() {}

    virtual Initial_Data init_data(char* data) = 0;

    virtual void print_info();

    virtual char* read_id(char*& data);
    virtual CORBA::TypeCode_ptr read_TC(char*& data);
    virtual void skip_digits(char*& data);
    virtual short read_short(char*& data);
    virtual long read_long(char*& data);
    virtual unsigned short read_ushort(char*& data);
    virtual unsigned long read_ulong(char*& data);
    virtual float read_float(char*& data);
    virtual double read_double(char*& data);
    virtual char read_char(char*& data);
    virtual unsigned char read_octet(char*& data);
    virtual char* read_string(char*& data);
    virtual void read_spaces(char*& data);
    virtual unsigned char read_boolean(char*& data);
    virtual void* read_void(CORBA::TypeCode_ptr tc, char*& data);

  protected:

    char *prog_name;
    int ior_flag;
    char* operation;
    char** my_argv;
    int my_argc;

    Repository_Manager_ptr my_rep_mgr;

    resolve_fun srv_rep_mgr;


  };
```

The *Rep_Client* template is parametrized by a single parameter – *Repository_Types* class, which extends the parameter class of the *Basic_Client* template with two typedef definitions:

- *Repository_Manager* – the IDL interface provided by the repository manager supporting *add*, *update* and *remove* operations,

- *Repository_Manager_ptr* – a repository manager reference type.

This explains why the *DEF_TYPE_ABNG* macro described in the previous section possesses these definitions. Although they are equivalent to the *Repository* and *Repository_ptr* types respectively, their inclusion was necessary to keep the macro for the *ABng_Core_Management* module consistent with that of the *Domain_Repository_Management* module. Due to relationships between the *ABng_Core_Management* and *Domain_Repository* modules, which impose some constraints on compilation process, the frame for the *Rep_Client* parameter class had to be defined separately in a form of *DEF_TYPE_MGR* macro.

16.2.2 Attribute Description

- *operations* – the static field that lists names and number of parameters of all non-default operations supported by a given client; if no non-default operations are supported the field should be set to `{ { 0, { 0 } } }`,
- *description* – the static field that defines the syntax of the command line argument which depicts description of an entity of a given repository; the value of this field is printed as a help to a user of a given client,
- *ior_flag* – specifies whether the client should print IORs or item identifiers of repository entities,
- *my_rep_mgr* – the reference to the repository manager object; this reference is the same as *my_rep* for all *ABng_Core_Management* repositories,
- *srv_rep_mgr* – pointer to an appropriate *ABng_Server_Manager* method that builds a compound name bound with *my_rep_mgr* in Naming Service,

16.2.3 Method Description

Two methods invoked in *main* of a repository client are:

- *init* – first it initializes ORB and BOA references and reads command line options passed in arguments. Afterwards, *init* instantiates a repository reference using *init_rep_ptr*.
- *run* – based on the first command line argument which denotes a repository operation to perform the function invokes an appropriate operation *find*, *find_values*, *add*, *update* or *remove*. In case of *find* and *find_values* it is assumed that a user wants to list all repository entities, so pattern may be set using *init_pattern* method. Operation *add* interprete the second command line argument as an item description using *init_data* method. This method must be defined specifically for each repository. Various methods are provided within template to facilitate its implementation. The operation *update* interprets the second argument as a reference of an object to update using *init_item_ptr* method. The last argument is interpreted as a new description. The operation *remove* uses only the second argument and interprets is as a reference of an item to remove. If the first command line argument does not represent any of these five methods, the *operations* field is analyzed to check if it is a declared non-default operation. If so, the *other_operations* method is invoked.

16.2.4 Implementing a client

In order to implement the client the following steps must be taken:

1. Ensure the domain_types.cc or abng_core_types.cc contain definition of the appropriate **_Types* class.
2. Create a client implementation class that inherits from the appropriately parametrized template.
3. If item description or a parameter of any non-default operation contains a reference of an entity from a repository other than represented by the client, create an instance of *Basic_Client<A>* as a member of the implementation class, where *A* is a name of the **_Types* class corresponding to this repository. Entity references will be obtained using *init_item_ptr* call on the instance. Invocation of this method must be proceeded by *init_rep_ptr* call.
4. Define *operations* and *description* fields, and if necessary redefine *other_operations* method.
5. Implement *init_data* method.
6. Implement the client main function.

16.3 Implementation Specification

16.3.1 Source Code Organization

- *common/basic_client.h, common/basic_client.cc* – definition of the *Basic_Client* template,
- *common/rep_client.h, common/rep_client.cc* – definition of the *Rp_Client* template,
- *common/domain_rep_types.h, common_domain_rep_types.cc* – definition of the *DEF_TYPE* macro and *Basic_Client* parameter classes corresponding to the *Domain_Repository* module,
- *common/domain_rep_types_mgr.h* – definition of the *DEF_TYPE_MGR* macro and *Rep_Client* parameter classes corresponding to the *Domain_Repository_Manager* module,
- *common/abng_core_types.h, common/abng_core_types.cc* – definition of the *DEF_TYPE_ABNG* macro and *Basic_Client* and *Rep_Client* parameter classes corresponding to the *ABng_Core_Management* module.

References

[1] Frazer Bennett and Andy Harter. Low bandwidth infra-red networks and protocols for mobile communicating devices. Technical Report 93.5, Olivetti Research Laboratory, Cambridge, UK, 1993.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[3] Andrzej Uszok Jakub Szymaszek and Kszysztof Zieliński. Abng — new generation active badge systems. Technical report, UMM, Kraków, Poland, 1997.

[4] Andrzej Uszok Jakub Szymaszek and Kszysztof Zieliński. Building a scalable and efficeint component oriented system using corba — active badge system case study. In *Proceedings of The Fourth Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 165–177, Santa Fe, New Mexico, April 1998.

[5] Capasso Raffaele. Low bandwidth infra-red networks and protocols for mobile communicating devices. Technical Report 93.5, Olivetti Research Laboratory, Cambridge, UK, 1993.

[6] Jakub Szymaszek, Aleksander Laurentowski, and Krzysztof Zieliński. Instrumentation of CORBA-compliant applications for monitoring purposes. In *Proceedings of European Research Seminar on Advances in Distributed Systems (ERSADS '97)*, Zinal, Switzerland, March 1997.