Technical Design

Technology choices

Core framework

Giuseppe Lo Presti, German Cancio, Sebastien Ponce
CERN / IT

Castor Readiness Review – June 2006

Core Framework

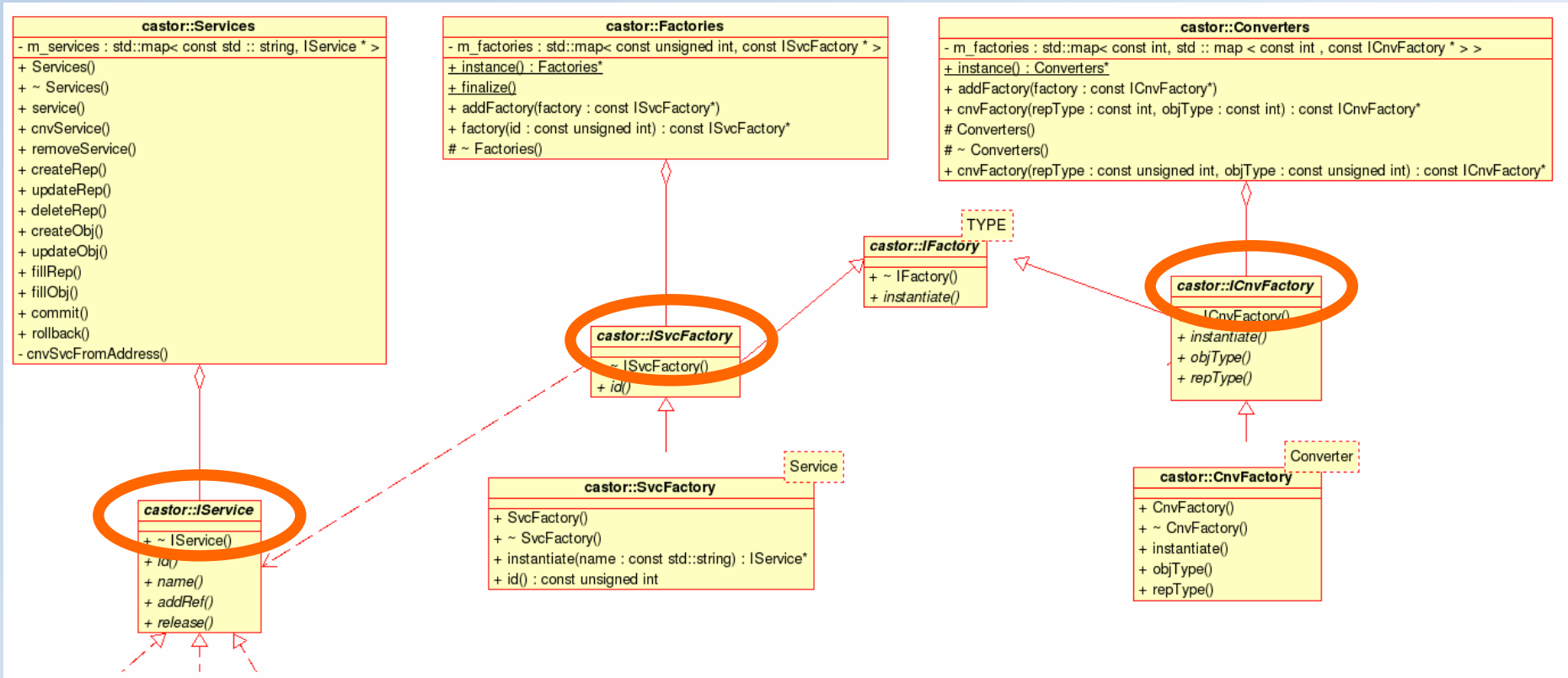
- Goal
  - code maintainability
  - reusability
- Object Orientation
  - Heavy usage of abtract interfaces
  - Java-like inheritance model
    - Multiple inheritance of interfaces only
- Two main parts:
  - Generic mechanism to handle *services*
  - Generic framework for multithreaded daemons

Services handling

- Requirements
  - Several different internal services need to be handled in a homogeneous way
  - Autogenerated persistency layer and streaming layer
  - Reusability
- Chosen implementation
  - ***Gaudi***-like framework with specific enhancements
  - Usage of the *Factory* Design Pattern
  - Central place where services are loaded and instantiated

Use of the Factory paradigm

- A **thread safe** singleton
  - **`Services`** – list of service *instances*
- Two pure singletons
  - **`Factories`** – list of service *factories* with “autoregistration”
  - **`Converters`** – list of converter *factories*

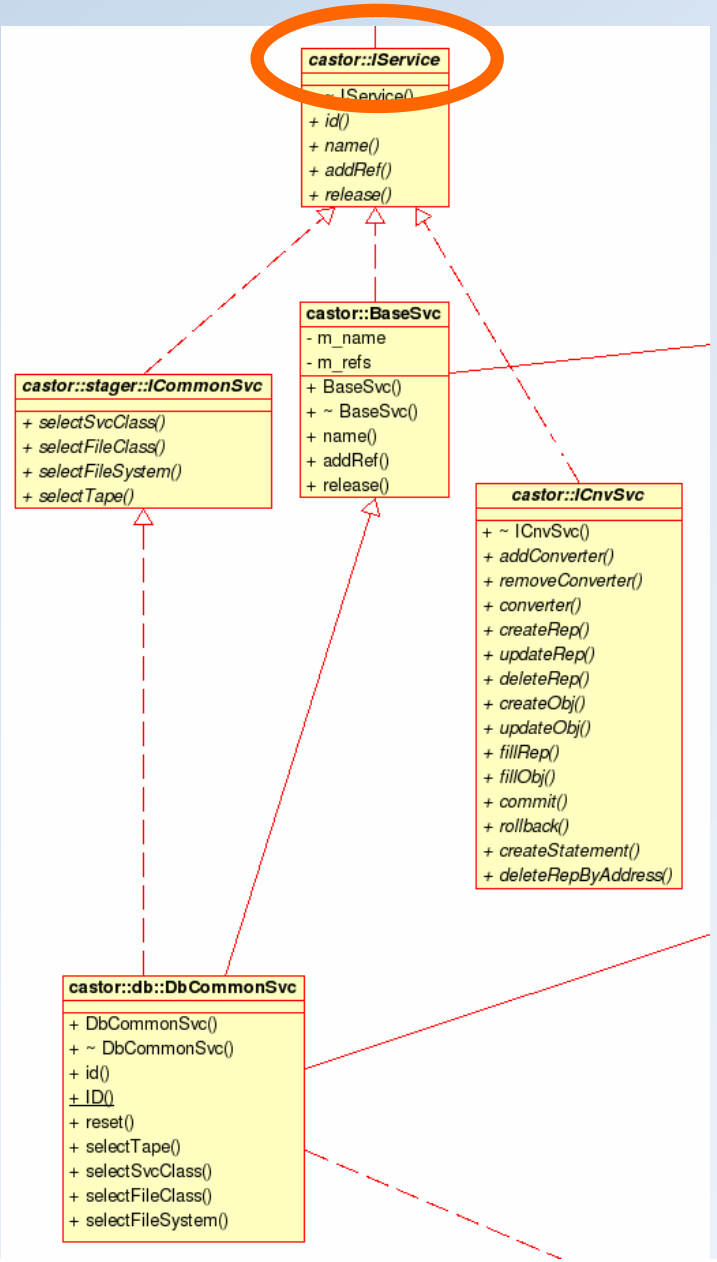


Services loading algorithm

- The **`Services`** "singleton" holds a list of all instantiated services per thread (the **`Services`** class is instantiated once per thread)
- When a new service is requested via **`service(name, id)`**:
  - If a *Factory* has been registered for the given **`id`**, instantiate the service and store it in the internal list with the given **`name`**
  - Otherwise, if an id *alias* has been defined, try to use it
    - Aliases are defined in the main configuration file **`castor.conf`**
  - Otherwise, if a library has to be used, use **`dlopen()`** to dynamically load it and then look again for the factory
    - **Libraries to be used** (e.g. database layer) **are defined in the configuration file** too
    - Can be changed without recompilation
  - Otherwise raise an error
- The implementation is generic enough to be suitable for any project around Castor (for instance SRM 2)

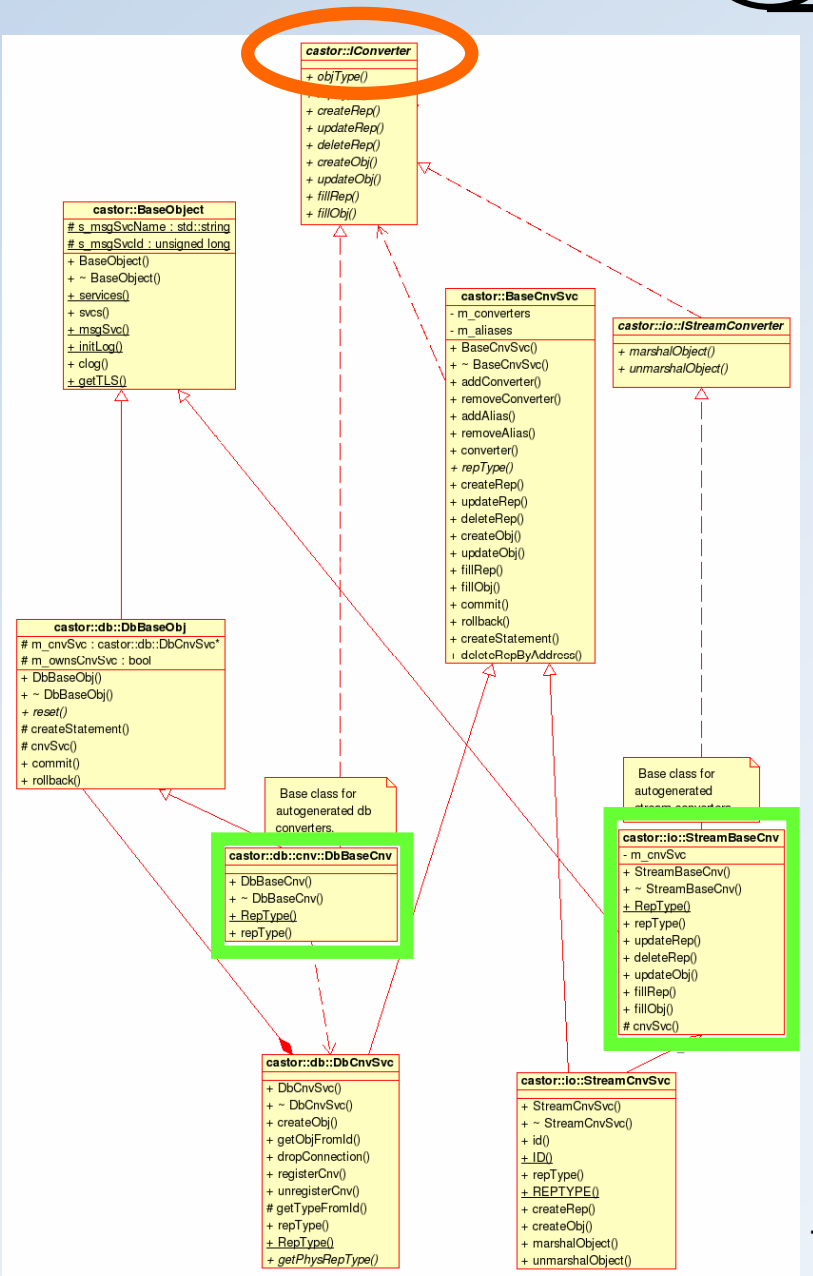
Services

- Inherit from **`IService`**
  - have a name and an id
  - common ancestor for autogenerated as well as “handwritten” services
- Special services
  - Conversion services
    - inherit from **`ICnvSvc`**
    - provide *create/fillObj/Rep* type of interface
  - DB services
    - implement specific queries to the DB
    - DB-vendor independent when possible
    - multiple implementation for the some of them (e.g. the GC service)
      - *DB* and *Remote*
    - example: **`DbCommonSvc`**



Converters

- Inherit from **`IConverter`**
  - have an object type and a representation type
  - implement **generic** *create/fillObj/Rep* type of interface
- Two main converter types
  - **`DbBaseCnv`**
    - Base class for autogenerated db converters
    - Uses the **`DbCnvSvc`** service
  - **`StreamBaseCnv`**
    - Base class for autogenerated streaming converters
    - *Uses* the **`StreamCnvSvc`** service
- The whole class diagram is available on the web



Example: load a service

- Request Handler:
  instantiation of the stream and db conversion service

```
castor::IObject* obj = sock->readObject();
…
castor::BaseAddress ad;
ad.setCnvSvcName("DbCnvSvc");
…
svcs()->createRep(&ad, obj);
```

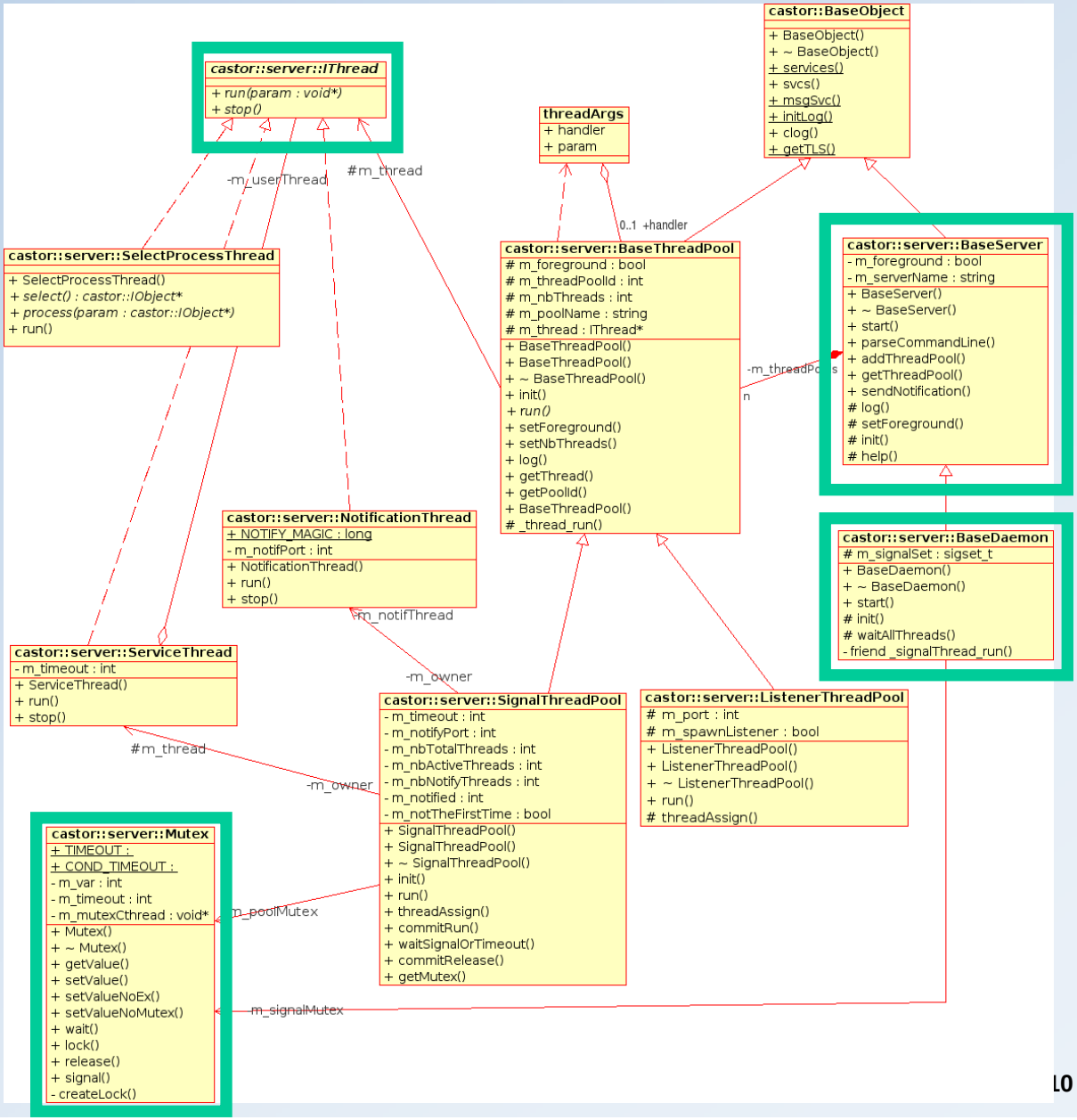
```
castor::IService *svc =
      Services::service("DbCnvSvc", castor::SVC_DBCNV);
castor::IService *svc2 =
      Services::service("StreamCnvSvc", castor::SVC_STREAMCNV);
```

Framework for multithreading

- Requirements
  - Usage of POSIX threads (*pthread* API in Unix)
  - Support for ***thread pools*** to implement **independent** services
    - Stager
  - Daemon mode
- Implementation
  - OS abstraction layer in C: *Cthread* API
    - Replicated all pthread API
    - One of the most mature parts in Castor (dated 1999)
  - C++ wrapper
    - Java-like: ***`IThread`*** interface with abstract **`run()`** method
    - On top of the Cthread API
    - Ease reusability
    - Implemented in late 2005

Framework for multithreading

- ***IThread***
  - Programmer's interface
- **BaseServer, BaseDaemon**
  - To encapsulate the **`main()`**
- Thread Pool
  - Two types:
    - Listener
    - Signal
- **Mutex**
  - encapsulate a POSIX mutex
  - largely used inside the framework



Framework for multithreading

- Reusability
  - All daemons share the same Cthread API
    - Good code reutilization, some duplication still occurs for daemons written in C
  - New C++ daemons use the C++ framework and inherit from those classes
    - RH, SRM 2
    - VDQM 2, Repack 2 to come soon
    - Maximum reutilization of the code: none of them directly have to deal with threads handling
    - Next one: the stager?
- Performance issues
  - Cthread proven to work since years for Castor1 and external software like Lemon
  - Negligible impact of the C++ wrapper on top of Cthread

Multithreaded daemon example

- Excerpt from the SRM 2 daemon **`main()`** function

```
srm::daemon::SrmDaemon daemon("SRM Daemon");

daemon.addThreadPool(
  new castor::server::SignalThreadPool("StageRequestHandler",
    new srm::daemon::StageReqSvcThread()));
// ...
daemon.addThreadPool(
  new castor::server::BaseThreadPool("GC",
    new srm::daemon::GCThread()));
daemon.getThreadPool('G')->setNbThreads(1);  // we need a single thread here

daemon.parseCommandLine(argc, argv);
daemon.start();
```

Comments, questions?