Over–all Strategy for BRAD

1. Provide application interface between on- and offline applications and Database Manager (DBM).
2. Provided database structure for BRAHMS calibration, status, and run etc. databases.
3. Do this in an Object Oriented way.

BRAD is Brahms Run & Analysis Database

Status of BRAD

- Current underlying DBM is Objectivity.
- Provides ROOT compatible classes.
- Not all databases considered yet.
- Runs on Linux only (some compiler problems).

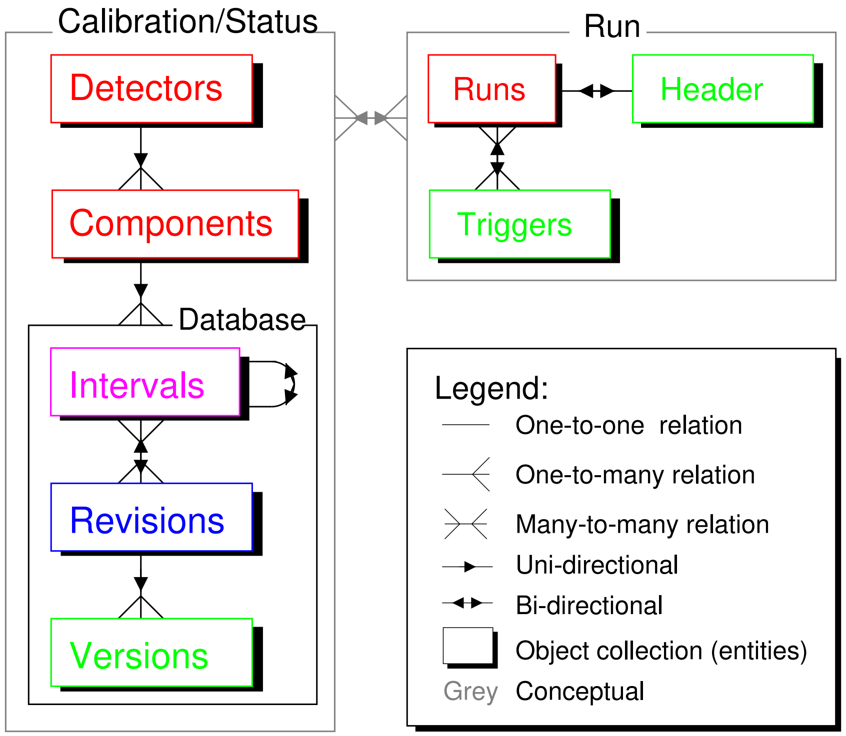
Advantages of BRAD

- Stand–alone framework, modular extension to ROOT (and BRAT, ofcourse).
- Object oriented → extentable to other underlying DBMs (ROOT, mySQL, PostGreSQL, Oracle, etc.).
- Object oriented → Data hiding — User only sees application layer.

Disadvantages of BRAD

- Objectivity is complex, and not too flexible.
- Objectivity classes are 'hard to swallow' by ROOT, needs wrappers. (Perhaps it's possible to generate *general* wrappers, and not the current context dependent).

Schema (1) — Graph



Schema (2) — Contents

Each component has a table for calibration parameters, status (such as high voltage, etc.) channel map, geometry settings (dynamic data), etc.

Additional relations could be Detector and Component geometries (static data), etc.

Also, on could think of a diary database, containing text comments.

Schema (3) — Relations

One–to–one and one–to–many relations can be expressed via containment.

However, Many–to–many can not. Needs *relations*, in DBM sense (in **ROOT**: pointers).

Also, Many–to–many calls for calls for *reference counting*.

Objectivity provides mechanisms for all kinds of relations.

Currently **BRAD** uses **Objectivity** relations for all for relations.

Conceptual relations, groupings, etc., are implemented in the application interface. E.g., the query looks up a detector, it's sub–component, then make queries the appropriate database (calibration/status).

Entities (1) — Graph

Interval Object

```
time_t     StartDateTime
time_t     EndDateTime
time_t     FromStartDateTime
time_t     FromEndDateTime
BdInterval PreviousInterval
BdInterval NextInterval
```

Doubledly Linked List

Revision Object

```
int      RevisionNumber
time_t   RevisionDate
string   Author
string   Comments
```

Version Object

```
int          VersionNumber
BdDataTable  DataTable// Array of DataObjects
```

Data Object

```
double Parameter
double Error
```

Simple Data Object

Entities (2) — Contents

Entities contain *header* information — no data is explicitly stored.

Data enters the schema via the `BdData` class. An array of such objects is stored in the `BdVersion` entities.

`BdData` is a simple class, to provide flexibility. It contains a parameter, and an error on that parameter.

Users should provide custom classes, derived from `BdDataTable` which is an array of `BdData`. Custom classes should have meaningful methods.

We will see an example of such a class in a minute.

How To use BRAD

- Interactively in ROOT session (data analysis, mining, etc.).
- Via applications (various control and monitor programs, ready–made data analysis/mining).

How to setup BRAD

Objectivity enforces some restrictions on your database schema, by insisting on data definitions, described in Objectivity's Data Definition Language (DDL).

Classes `BdData` and `BdDataTable` is a sort of work–around.

Still, some work has to be put into the initial setup.

- Create detector component specific data tables.
- Setup specific databases (calibration and status) for detector and it's sub components.

Schema can evolve, but it requires intimate knowledge of Objectivity.

How to do queries in BRAD

1. Query Run database to obtain run info (start/end times, etc.)
2. Use this info to query other databases.
3. Tables of data are returned. (User specified classes.)
4. Loop until at end of request.

Using **Objectivity** as DBM, most types of queries has be hard–coded into **BRAD**.

It requires intimate knowledge of **Objectivity**, to do complex queries on for example, the run database (e.g., “SELECT run FROM runs WHERE trigger $= X$ AND events $\leq Y$ AND timestamp $\geq Z$”).

Data Tables (1) — Example

An example of a custom class is `PMTTable`:

```
1 class PMTTable : public BdDataTable {
2 private: // data is in BdDataTable.
3 public:
4   PMTTable(void) : BdDataTable(3) {}
5   void SetAdcGain(BdData* gain) {
6     SetEntry(gain,0); }
7   void SetAdcOffset(BdData* offset) {
8     SetEntry(offset,1); }
9   void SetPedestal(BdData* pedestal) {
10    SetEntry(pedestal,2); }
11  BdData* GetAdcGain(void) {
12    return GetEntry(0); }
13  BdData* GetAdcOffset(void) {
14    return GetEntry(1); }
15  BdData* GetPedestal(void) {
16    return GetEntry(2); }
17 };
```

The 3 in line 4 is the size of the array.

`BdComponent` could contain information on table class (in **ROOT** a `TClass` object).

Data Tables (2) — IO

BdDataTable are the input/output used in BRAD:

```
1 PMTTable* data = new PMTTable;
2 data->SetAdcGain(new BdData(0.1234, 5.6789));
3 data->SetAdcOffset(new BdData(9.8765, 4.3210));
4 data->SetPedestal(new BdData(3.14,2.71));
5 // Add calibration for PMT1 in H1 from
6 // 04/30/1975 to 01/01/2000.
7 // h1 is a 'pointer' to a BdDetectorIO object
8 h1->Revise("PMT1",
9            new BdTime(1975,4,30,0),
10           new BdTime("01/01/2000 00:00:00"),
11           data);
12 // Get calibration of PMT in H1 around 05/01/1985,
13 // created by me.
14 PMTTable* table =
15   (PMTTable*)h1->Fetch("PMT1",
16                        new BdTime(1985,5,1),
17                        "Christian Holm Christensen");
```

Explicit cast in line 15 is nessecary (Fetch() returns pointer to ABC BdDataTable).

What needs to be done

If we stick to Objectivity a lot of things still need to be done:

- The schema has to be updated for the status databases (almost trivial).
- The schema has to be extended for at least run information (requires some work).
- Explore what kinds of queries should be coded into BRAD (tricky).
- Explore what kinds of data objects is needed (takes some thinking).
- If more databases are needed (like catelogs of code, DSTs, and their relations, etc.), the schemas for these has to be formulated, coded, and integrated (not trivial).
- Bugs need to be identified and corrected (hard!).
- And properly much more ... (extremely hard).

Advantages of using Objectivity

- Direct use of classes/objects, as **ROOT** and **BRAT**.
- Backup facilities (haven't studied those).
- Recovery facilities (haven't studied those).
- Transaction model makes for more secure updating.
- Support from **RCF**.

Disadvantages of using Objectivity

- Too many features. Promotes confusion, and isn't needed!
- DDL — extremely hard to dance with. Requires 'experts'!
- Too hard to easily use with **ROOT**.
- Queries doesn't come easily.
- Object Oriented features really isn't call for in **BRAD**. Relations can do it.
- Commercial product!

Other DBMs (1) — SQL

Better query support is available in SQL (Standard Query Language) DBMs.

Candidates are: **mySQL**, **Oracle**, and **PostGreSQL**. All have C++ APIs.

PostGreSQL is the best. **mySQL** is simple. **Oracle** is really out of the picture (too large, too bad).

PostGreSQL has all security features, **mySQL** only a subset.

Run, Detector, etc. information, are the ones typically queried often, and with complex queries. This data could be managed by SQL DBM.

Other DBMs (2) — Object Oriented

Other data is best expressed using classes/objects, since that's what we use! Hence we still want OO DBM.

Candidates are: **Objectivity**, **ROOT**, ... (?)

Even though we don't query these databases that much, **Objectivity** is still a bad choice.

Objectivity is too complex. It alienates the user. Is very rigid (at least in our context).

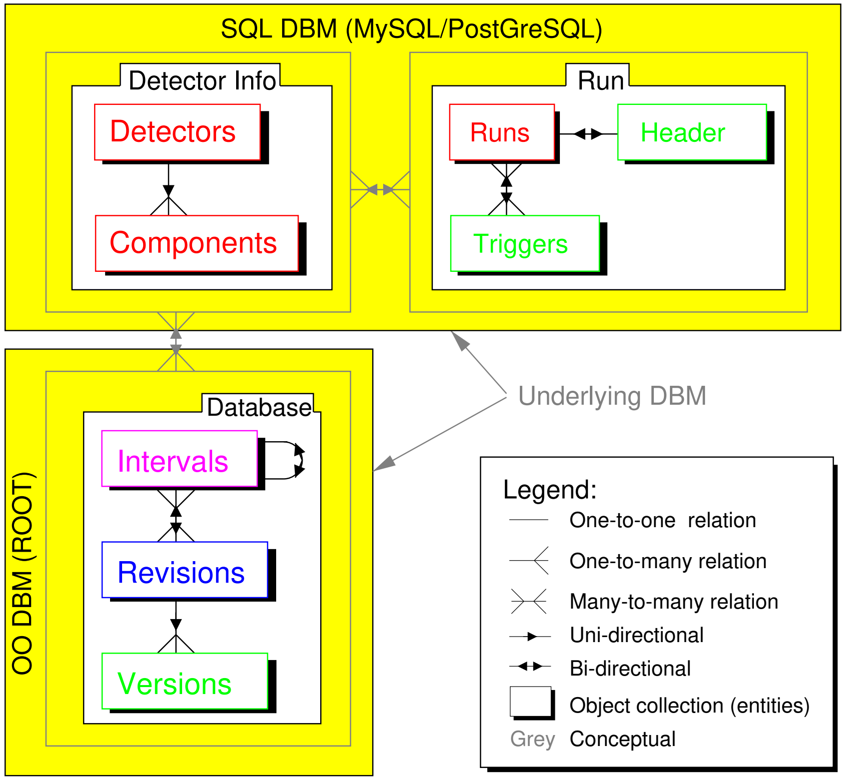
ROOT is our *main framework*!!!

However, **ROOT** lacks transactions, but this can properly be overcome.

ROOT lacks backup — simple solution, make copies!

ROOT has reasonable recovery support.

My recommendation!?



Overview