Self-Filling Histograms: A toolkit for object-oriented histogram filling

Jenny List, University of Wuppertal
& Benno List, ETH Zurich
CHEP '04
Interlaken, Sept. 30 2004

- **Introduction: What is an SFH?**
- **Design Specs: Why have we developed it?**
- **Simple Examples: How does it work?**
- **Basic Abstractions: The main ideas inside SFH**
- **An advanced Example**
- **Conclusions & Outlook**

Introduction: What is an SFH?

- Self-filling means that the histogram object „knows“
  - with which quantity it should be filled
  - with which weight
  - and under which conditions
- This implies that
  - the Fill() method doesn't need any arguments
  - the call to Fill() doesn't need to be within nested “**`if`**” statements

=> the histogram filling can be done automatically!

- Self-filling **Sets/Matrices** of Histograms allow easy handling of large amounts of similar histograms
  (-> differential x-sections, data / MC comparisons ....)

Design Specifications I

- Put **data & functionality** which belongs together **inside the same object**:
  - a histogram get's **completely** defined at booking time
  - the user doesn't have to pay attention to fill the right quantity into the right histogram in event loop...
    => easier to <u>maintain</u> consistency
- **Declarative** instead of **procedural** programming:
  - no need for many nested loops or if statements
  - several **small classes** instead of a few big ones make code more **understandable & <u>maintainable</u>**

Design Specifications II

- Detect programming errors early
  - **compiled code** instead of macro
  - use **strong typing** ability of C++
  - **avoid string parsing** at run time
- Efficient handling of large number of histograms
  - treat similar histograms as one object
    (ex: same quantity for data & MC samples)
- Minimal performance penalty
  - **loop** over events only **once**
    (≠filling several histos with TTree::Draw())
  - supply **caching** for complicated functions

Usage Examples

Before looking at the design:

.....let's see how it can be used!

All examples assume you to have:

- a class `MyTree` representing your tree
  (handwritten or generated by `TTree::MakeClass()`)
- a class `AnalysisLoop`
  derived from the SFH class `EventLoop`:

```
class AnalysisLoop : public EventLoop
```

Histogram a variable from a RooT tree

Constructor of `AnalysisLoop`:

```
AnalysisLoop (MyTree& tree) {
    FloatFun& METFun = ntfloatfun (tree, &MyTree::MET);
    METHist = new SFH1F("methist","Missing Transverse Energy",
                        50, 0., 200.,
                        this,        // histo registers itself for filling
                        METFun);     // histo now knows what to fill!
  }
```

METHist is filled automatically during event loop of the base class

- afterwards you maybe want to plot & store it:

```
output (TFile *psfile, TFile *rootfile) {
    TCanvas *c1 = new TCanvas("c1", "MET", 600, 800);
    TPostScript ps (psfile, 111);
    METHist->Draw();

    TFile file (rootfile, "RECREATE");
    this->Write();             // writes all histos of AnalysisLoop
    file.Write();
  }
```

Add a cut & a weight to your events

Use only events with exactly one b-tagged jet:

```
IntFun& NBJetFun = ntintfun (tree, &MyTree::NBJet);
METHist = new SFH1F("methist","Missing Transverse Energy", 50, 0., 200.,
                    this, METFun, NBJetFun == 1);
```

Now weight events according to p_t of the b-jet:

```
// ptBJetWeight is a user class derived from FloatFun
FloatFun& ptBJetWFun = *new ptBJetWeight (tree, .... );
METHist = new SFH1F("methist","Missing Transverse Energy", 50, 0., 200.,
                    this, METFun, NBJetFun == 1, ptBJetWFun);
```

Plotting the p_t of **all** jets: just need an iterator!

```
FillIterator& jetiter = ntfilliterator (tree, &MyTree::NJet);
FloatFun& ptJetFun = ntfloatfun (tree, &MyTree::ptJets, jetiter);
ptJetHist = new SFH1F("ptjethist","Pt of jets", 50, 0., 200., this, ptJetFun);
```

Caching the value of a function:

```
FloatFun& ptJetFun = cached (cachedObjects,
                        ntfloatfun(tree, &MyTree::ptJets, jetiter));
```

The ideas inside....

After a first impression what SFH is:

... how does it work internally?

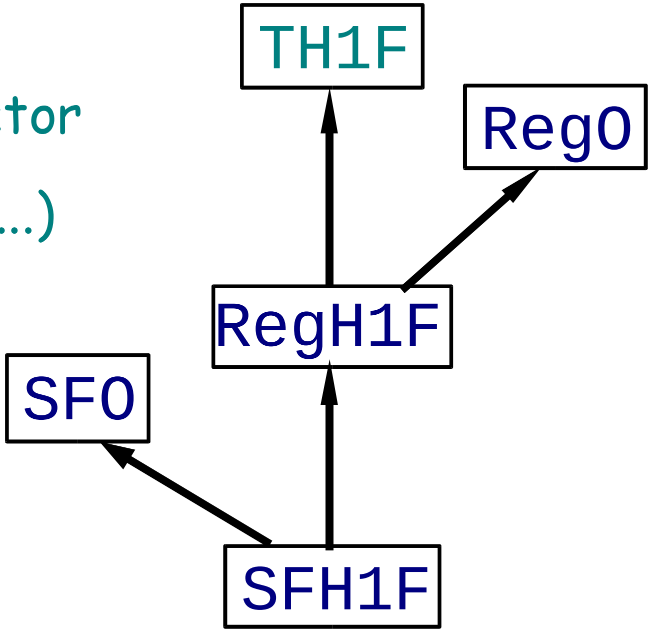
- registered objects
- self-filling objects
- function objects
- cached objects
- groups of registered objects
- visitors

.... and finally a more advanced example!

Basic Abstractions 1: Self-Fillingness

- Registered objects (`class RegO`):
  - registers itself in the `ROList` given in contructor
  - an `ROList` can notify its elements (for filling...)
  - the `EventLoop` base class is an `ROList`
- Self-filling objects (`class SFO`):
  - "interface" for argumentless filling: `virtual SFO::Fill() const = 0;`
  - **implementation** of `Fill()` using pointers to function objects & iterators in derived classes:

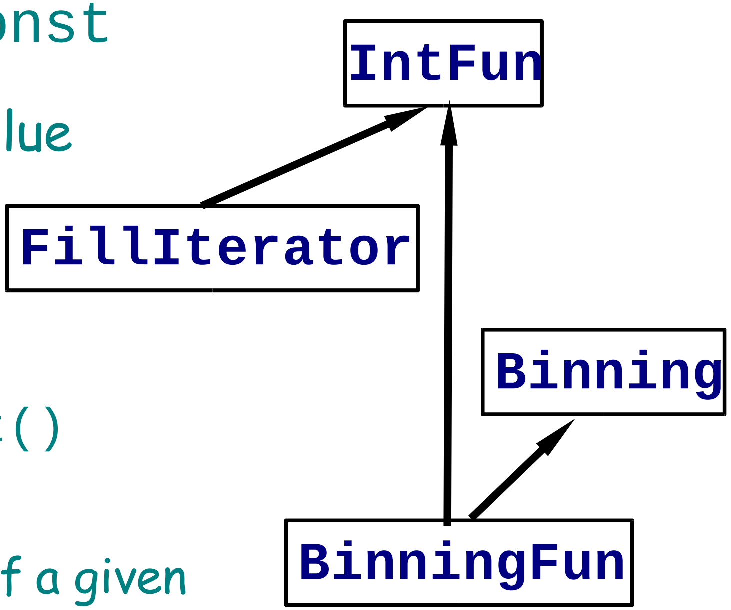
```
void SFH1F::Fill() {
  for (iter->reset(); iter->isValid(); iter->next()) {
    if ((*cut)())
      this->TH1F::Fill ((*xfun)(), (*wfun)());
  }
}
```



same for:
TH2F, TProf, ...

Basic Abstractions 2: Function Objects

Function objects implement `operator()() const`

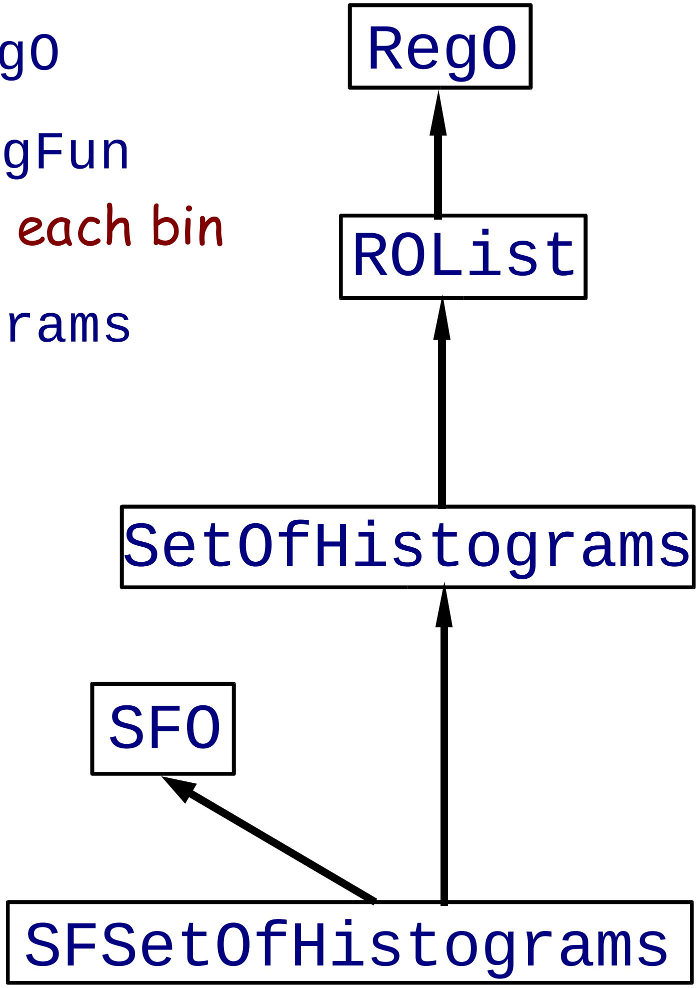


- `FloatFun`: typically used as fill or weight value
- `BaseCut`: “BoolFun“ - used for cutting
- `IntFun`: returns an integer - special cases:
  - `FillIterator`: additionally `next(), reset()` -> access to multiple entries per event
  - `BinningFun`: returns the number of the bin of a given `Binning` into which a value belongs
- function objects can be combined with + - * / && || sqrt ...
- there are global functions for some standard cases:
  - create function objects from tree variable -> ntfloatfun(...), ntfilliterator(...)
  - cache result of function until next event is read -> cached(...)

=> Cached object: base class `CachedO`, derived from `RegO`

Basic Abstractions 3: Groups of Histos

- Histograms can be grouped...
  - a list of registered objects `ROList`: any `RegO`
  - `SetOfHistograms`: a histogram + 1 `BinningFun`
    => the set creates & manages histograms for each bin
  - `MatrixOfHistograms`: a 2D `SetOfHistograms`
    => 2 `BinningFuns`
  - Sets/Matrices can
    - be made self-filling
    - be added, multiplied, ...
    - create summary histograms of themselves
- ... & treated together => “visitor pattern“
  - many visitors predefined for drawing, attribute setting, fitting,...
  - easy to derive your own visitor from `HVisitor`



A more advanced Example

- The next two pages will show you how to
  - implement a jet parton association à la SFH
  - plot the energy difference between jet and parton
  - plot this energy difference differentially in bins of the parton energy
- Using
  - `IntFun, FloatFun, FillIterator`
  - caching
  - `SFH1F, BinningFun` & `SetOfHistograms`
- A **<u>real</u>** example from an ATLAS analysis done by a PhD student in Wuppertal....

Jet parton association in η-φ-space

```
class JetPartonFun : public IntFun {
 public:
  JetPartonFun (MyTree* tree, FillIterator *jetIter, FillIterator *partIter_)
    : partIter(partIter_) {
      jphi = ntfloatfun (tree, &MyTree::JetPhi, jetIter);
      jeta = ntfloatfun (tree, &MyTree::JetEta, jetIter);
      pphi = ntfloatfun (tree, &MyTree::PartonPhi, partIter);
      peta = ntfloatfun (tree, &MyTree::PartonEta, partIter);
  }
  virtual int operator()() {
    int result = -1;  float mindist = 9999.;
    // loop over partons
    for (partIter->reset(); partIter->isValid(); partIter->next()) {
     float dist = calcdist((*jphi)(), (*jeta)(),(*pphi)(), (*peta)());
        if (dist < mindist) {
         mindist = dist;
         result = (*partIter)(); }  }   // index of parton with min. dist.
    return result;
  }
 protected:
  FillIterator *partIter;
  FloatFun *jphi, *jeta, *pphi, *peta;
};
```

Using the Jet Parton Association

Define iterators:

```
FillIterator& jetIter  = ntfilliterator (tree, &MyTree::NJets);
FillIterator& partIter = ntfilliterator (tree, &MyTree::NPartons);
```

Make a jet parton association and **cache** it:

```
IntFun& jpasso = cached (cachedObjects,
                         *new JetPartonFun (tree, jetIter, partIter));
```

Now you can use it - instead of an **iterator**:

```
FloatFun& partEFun = ntfloatfun (tree, &MyTree::PartonEnergy, jpasso);
```

... and plot the **energy difference** between jet and parton:

```
FloatFun& jetEFun = ntfloatfun (tree, &MyTree::JetEnergy, jetIter);
deltaEHist = new SFH1F("deltaE","rel. energy diff.", 50, -10., 10.,
                       (jetEFun-partEFun)/partEFun, jpasso >=0);
```

With a **SFSetOfHistograms** it's trivial to do it in bins of E_{parton}:

```
BinningFun& ePartBinning = *new FloatFunBinnning (partEFun, 10, 0., 100.);
deltaEHistos = new SFSetOfHistograms ("deltaE","rel. energy diff.", 50,-10.,10.,
                         (jetEFun-partEFun)/partEFun, jpasso >=0, 0, 0,
                          ePartBinning);
```

Summary & Outlook

- the SFH toolkit encourages **object-oriented** analysis of RooT trees
- self-filling histograms “know“ what **they** have to fill into themselves
- small function objects encapsulate the algorithms
- facilitates a **declarative** programming style
- large numbers of histograms can be handled easily
  via sets of histograms
- used within H1, ATLAS, D0
- further reading:
  https://www.desy.de/~blist/sfh/doc/html/index.html
- contact: Benno.List@desy.de, Jenny.Boehme@desy.de