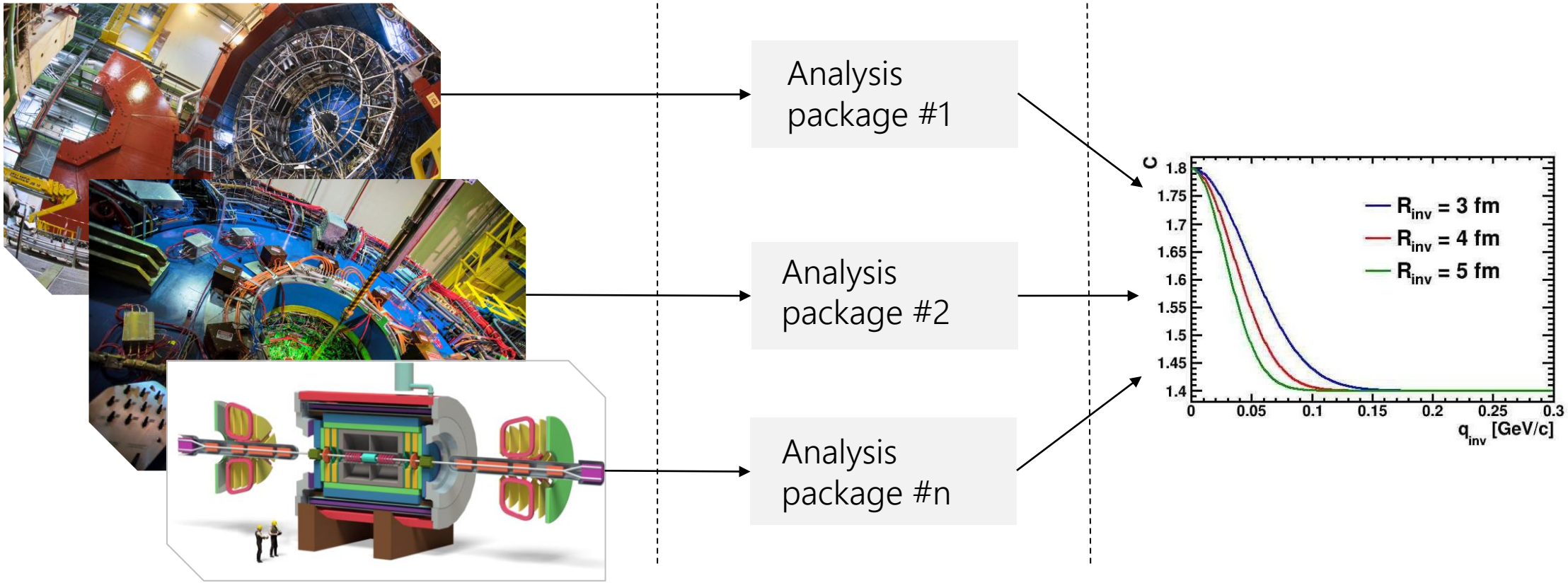
The 6th International Conference on Particle Physics and Astrophysics

Ekaterina Kuzina, Grigory Nigmatkulov

Experiment-independent framework for femtoscopic analysis

MEPHI
National Research
Nuclear University

Data flow in femtoscopic analysis



Each experiment has its own software for the analysis. But the result is technically the same.

Sources [1] [2] [3]

What is an experiment-independent framework?

➢ **Experiment-independent framework**
Software applicable to any experiment, if it is supplemented with a set of program modules representing the specifics of that experiment.

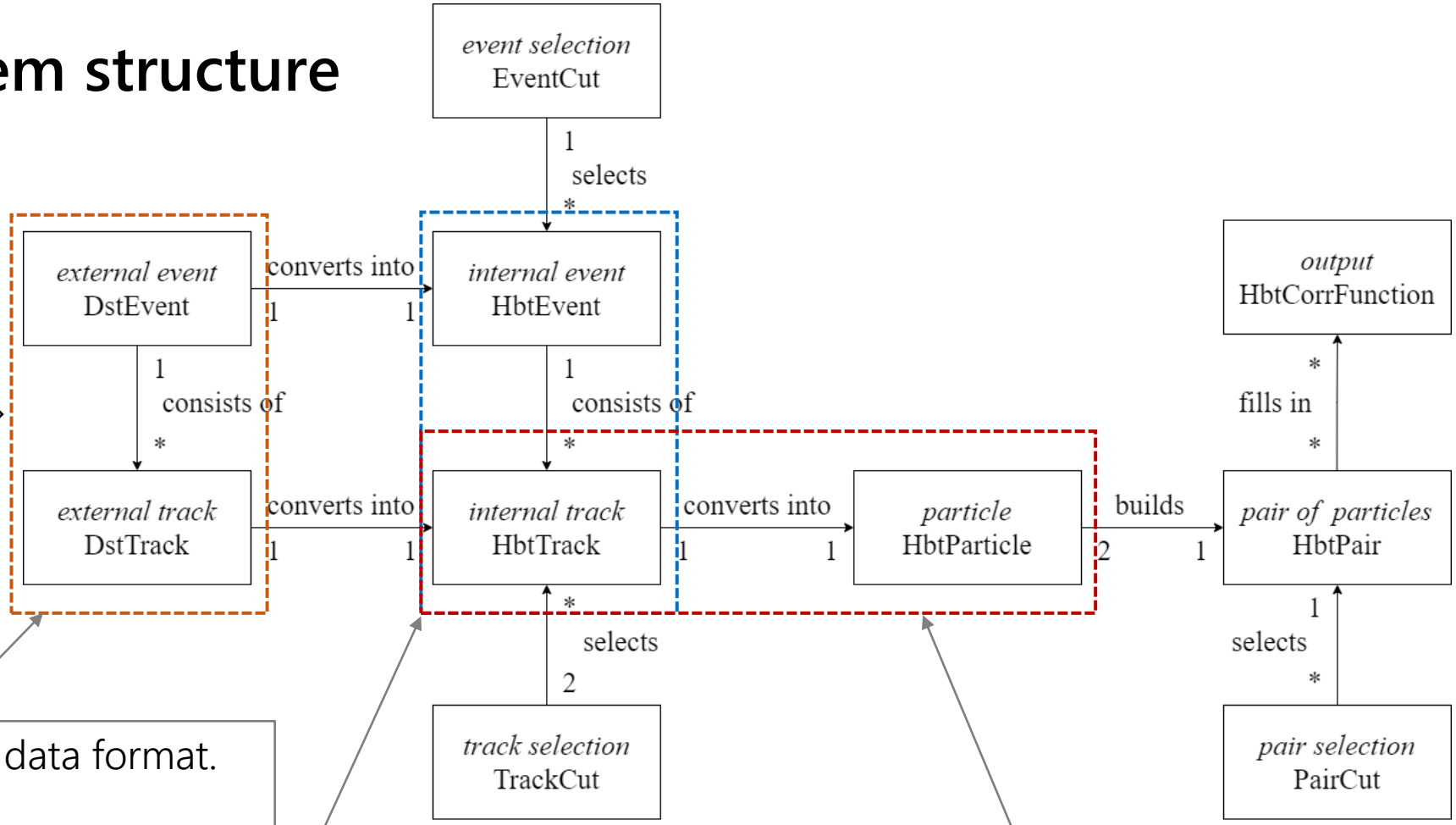
Pursue advantages of experiment independent framework

- ✓ No impact of software implementation when comparing the results of different experiments.
- ✓ An easy way to switch between analysis of different data.
- ✓ Data processing in a standalone mode.

Object-oriented system structure

$$C(\vec{q},\vec{k}) = N\frac{A(\vec{q},\vec{k})}{B(\vec{q},\vec{k})},$$

$$\vec{q} = \vec{p_1} - \vec{p_2}, \qquad \vec{k} = \frac{\vec{p_1}+\vec{p_2}}{2}$$

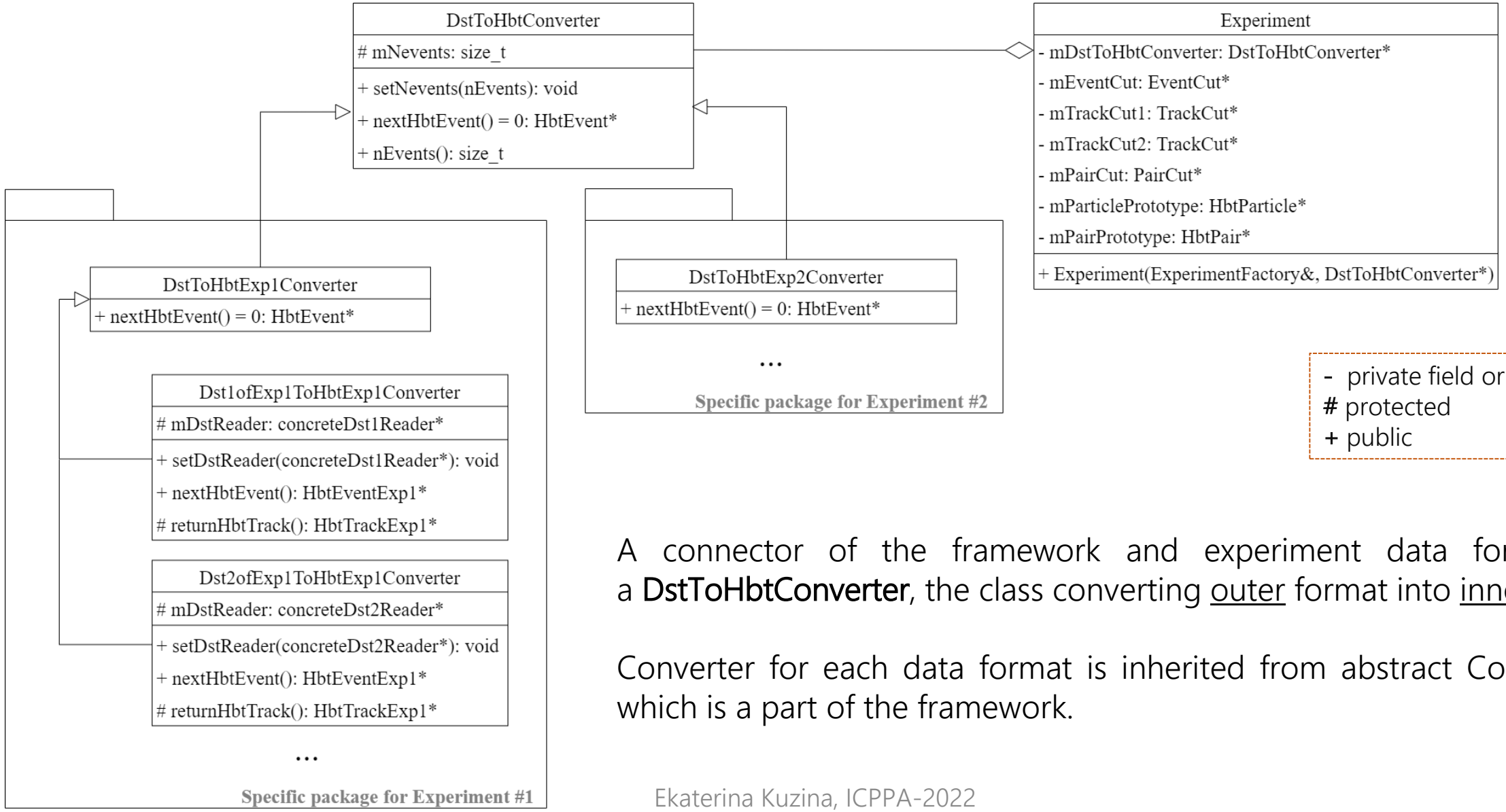


Outer event/track is an experiment data format. May vary for one experiment.

Inner event/track is an experiment-dependent data representation. One for one experiment.

Particle is a selected **track** with expected mass.

Outer and inner systems connection



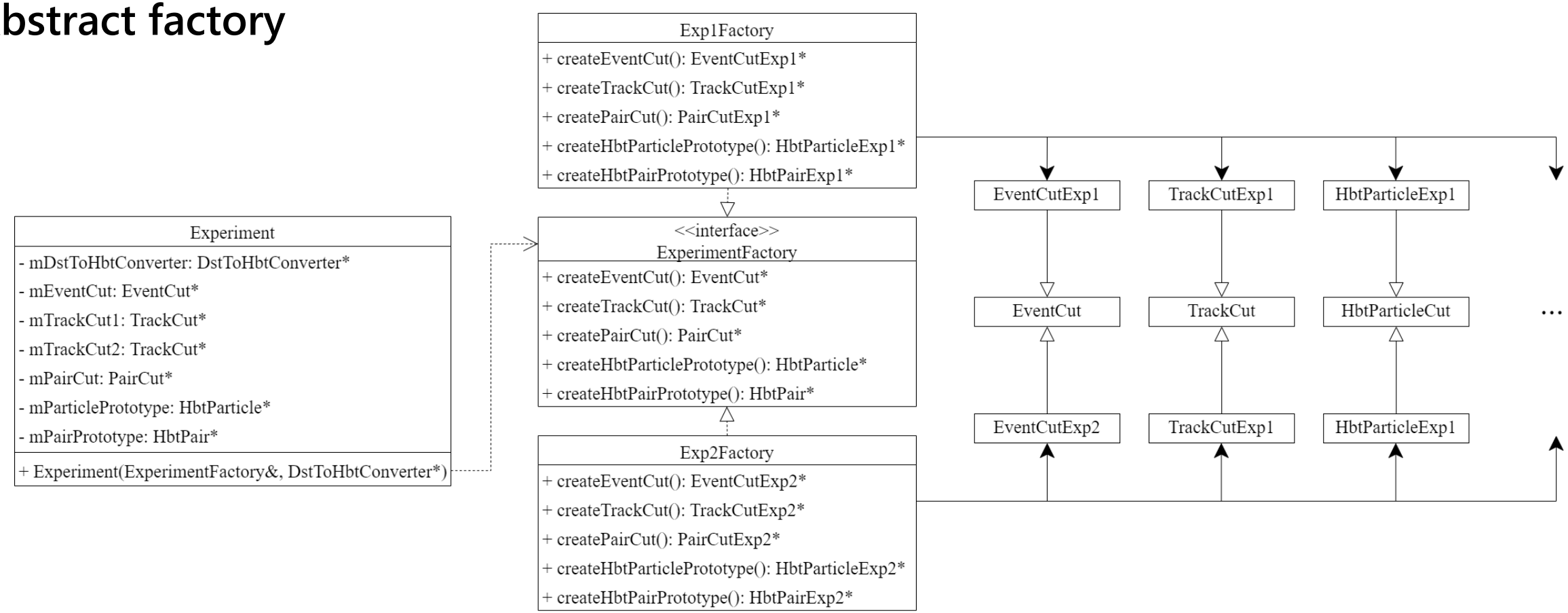
- private field or method
protected
+ public

A connector of the framework and experiment data format is a **DstToHbtConverter**, the class converting <u>outer</u> format into <u>inner</u> one.

Converter for each data format is inherited from abstract Converter, which is a part of the framework.

A set of experiment-dependent objects creation

Abstract factory

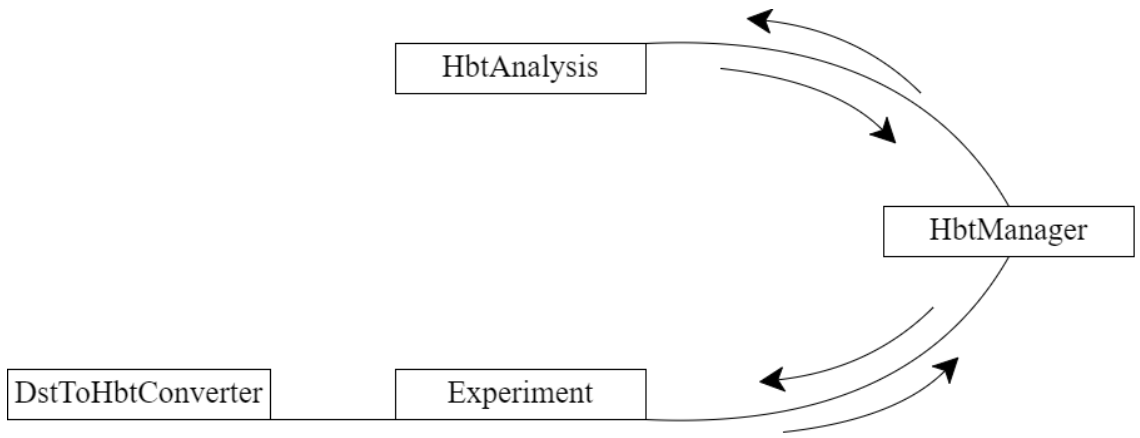


The abstract factory design pattern allows to create a set of heterogeneous classes united in meaning.
In the framework it is used to initialize Cut classes and create prototypes of Particle and Pair.
Prototypes are intended to create clones of class instances.

Injecting experiment specifics into data processing

Prototypes and mediator

```
int HbtAnalysis::processEvent( HbtEvent* hbtEvent) {
  //...
  HbtParticle* passedCut1Particle = particlePrototype();
  passedCut1Particle->setup(currentTrack, particleMass(0));
  //...
}
HbtParticle* HbtAnalysis::particlePrototype() {
  return mManager->particlePrototype();
}
```




```
HbtParticle* HbtManager::particlePrototype() {
  return mExperiment->particlePrototype()->prototype();
}
```

The prototype pattern is used to instantiate a Particle or Pair inside an abstract data processing.

```
HbtParticleExp1::HbtParticleExp1* prototype() override;
HbtParticleExp1* HbtParticleExp1::prototype() {
  return new HbtParticleExp1;
}
```

To introduce the specifics of the Experiment data into the Analysis, a mediator, called Manager, is used. There are no direct calls from the HbtAnalysis to the Experiment or DstToHbtConverter.

Pulling experiment specifics to user
Dynamic casting

```
// running macro
// ...
Experiment experiment( factoryExp1, dstToHbtConvExp1 );

EventCutExp1* evCut = dynamic_cast< EventCutExp1* >( experiment.eventCut() );

// Set specific cuts
evCut->setNChargedParticles( 1, 30000 );
```

An abstract object can be **unambiguously** converted to an instance of concrete class defined by the specifics of the experiment.

This is possible due to using the Abstract factory pattern to create this object and C++ runtime (dynamic) casting features.

Functional testing
User interface [1/2]

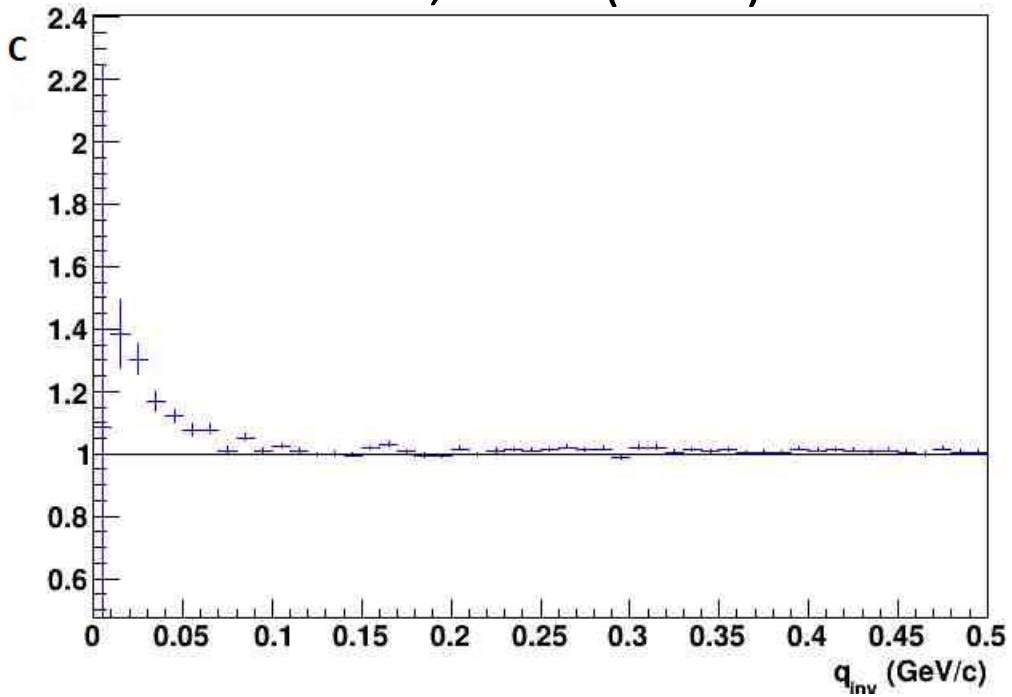
```
int main(int argc, char const *argv[]) {
 const char* inFile = "dstData.femtoDst.root";
 StFemtoDstReader dstReader(inFile);

 FemtoDstToStarHbtConverter dstToHbtConv(&dstReader);
 StFactory factory;
 Experiment experiment(factory, &dstToHbtConv);

 StHbtBasicEventCut* eventCut = dynamic_cast<StHbtBasicEventCut*>(experiment.eventCut());
 eventCut->setVertRPos( r[0], r[1] ); eventCut->setTriggerId( 350003 );
 //...

 StHbtBasicTrackCut* trackCut1 = dynamic_cast<StHbtBasicTrackCut*>(experiment.trackCut1());
 trackCut1->setCharge( -1 );
  trackCut1->setPt( pTtrack[0], pTtrack[1] );
 //...

 StHbtBasicPairCut* pairCut = dynamic_cast<StHbtBasicPairCut*>(experiment.pairCut());
 pairCut->setKt( kT[0], kT[1] );
 //...
```

Functional testing
User interface [2/2]

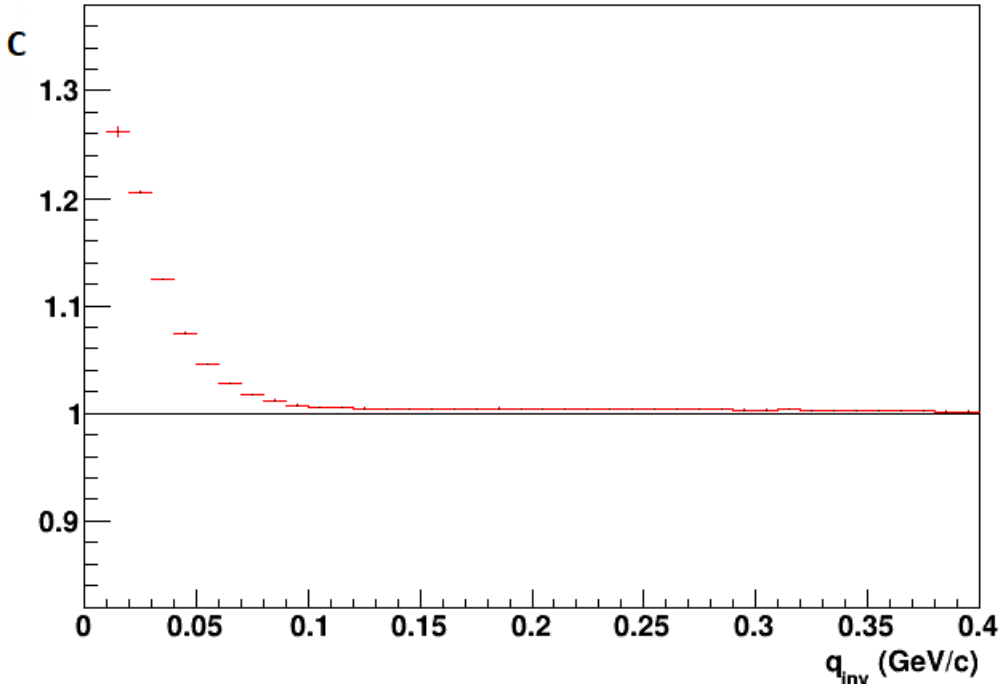
```
//...
HbtVertexMultAnalysis analysis = HbtVertexMultAnalysis(mZBins, mZ[0], mZ[1],
                                                       mMultBins, mMult[0], mMult[1], M_PION_PLUS);
 std::vector<HbtCorrFunction*> cfs = { new HbtCorrFunction("q_inv", 50, 0., 0.5, // qInv
                                                           1, 0.25, 0.35)}; // kT
 HbtManager manager;
 manager.setExperiment(&experiment);
 manager.setAnalysis(&analysis);
 analysis.setManager(&manager);
 manager.setFileName("testOut.root");
 manager.setCFs(&cfs);

 int rcode = manager.processEvents();
 return rcode;
}
```

Functional testing
Results

$\pi^+\pi^+$, STAR (RHIC)



$\pi^+\pi^+$, MPD (NICA)



STAR and MPD data formats are plugged in the framework.
One-dimension correlation functions are constructed.

Conclusions

Results

✓ The minimal framework is developed and tested.

✓ The first tests were performed for STAR (RHIC) and MPD (NICA) data.

Further development

❑ More interfaces for other experiments

❑ Containerization

❑ Functional development (in aspects of physics)

gitlab.com/ekuzina/multiexp-hbtmaker

MEPhI | 80 years National Research Nuclear University