Fermilab | U.S. DEPARTMENT OF ENERGY | Office of Science

Upgrading LArSoft to *art* 3

Kyle J. Knoepfel

10 October 2018

LArSoft Coordination Meeting

Upgrading LArSoft to *art* 3

- Moving LArSoft to support multi-threading is a significant effort that is underway
  - Serial event-processing is *art*'s default behavior—getting here is the first step.
  - You must opt-in to multi-threaded execution.

- There is guidance for how to do this:
  - https://cdcvs.fnal.gov/redmine/projects/art/wiki/Upgrading_to_art_3

Upgrading LArSoft to *art* 3

- Moving LArSoft to support multi-threading is a significant effort that is underway
  - Serial event-processing is *art*'s default behavior—getting here is the first step.
  - You must opt-in to multi-threaded execution.

- There is guidance for how to do this:
  - https://cdcvs.fnal.gov/redmine/projects/art/wiki/Upgrading_to_art_3

- Lynn and I have been upgrading the LArSoft repositories to support *art* 3.
  - Exposed some breaking changes I was not aware of
    - I am in the process of updating the *art* breaking-changes page
  - Exposed suboptimal practices (e.g.):
    - Lot of calls to `RandomNumberGenerator::getEngine(…)`, which will be deprecated in *art* 3.02 and removed in *art* 3.03.

Services are problematic

- They are the most poorly defined constructs within *art*.
- They are popular for their global state and easy configurability, but they cause myriad problems for multi-threading.
- If you would like to implement a typical algorithm, it is not necessary to create a service (provider).
  - An algorithm takes any number of inputs, and returns an output.

Services are problematic

- They are the most poorly defined constructs within *art*.
- They are popular for their global state and easy configurability, but they cause myriad problems for multi-threading.
- If you would like to implement a typical algorithm, it is not necessary to create a service (provider).
  - An algorithm takes any number of inputs, and returns an output.
- Stumbled across this:

```
void MyProducer::produce(art::Event& e)
{
  art::ServiceHandle<MyService> ms;
  ms->preProcessEvent(e); // Callback registered with art
  ...
}
```

Services are problematic

- They are the most poorly defined constructs within *art*.
- They are popular for their global state and easy configurability, but they cause myriad problems for multi-threading.
- If you would like to implement a typical algorithm, it is not necessary to create a service (provider).
  - An algorithm takes any number of inputs, and returns an output.
- Stumbled across this:

```
void MyProducer::produce(art::Event& e)
{
  art::ServiceHandle<MyService> ms;
  ms->preProcessEvent(e); // Callback registered with art
  ...
}
```

*Service callbacks are to be called by the framework **only**. Hands off!* ☺

Services are problematic

- They are the most poorly defined constructs within *art*.
- They are popular for their global state and easy configurability, but they cause myriad problems for multi-threading.
- If you would like to implement a typical algorithm, it is not necessary to create a service (provider).
  - An algorithm takes any number of inputs, and returns an output.
- Stumbled across this:

```
void MyProducer::produce(art::Event& e)
{
  art::ServiceHandle<MyService> ms;
  ms->preProcessEvent(e); // Callback registered with art
  ...
}
```

*Service callbacks are to be called by the framework **only**. Hands off!* ☺

Make all registered callbacks private members of the service.

NuRandomService interface change

- NuRandomService is widely used in the LArSoft repositories. Its createEngine provides a layer on top of *art*'s createEngine interface.
  - The only way to interact with the random-number engine is to call RandomNumberGenerator::getEngine
  - Such a call is expensive, exposes multi-threading details to the user, and is unnecessary.
  - In *art* 3.02 it will be deprecated; in *art* 3.03 it will be removed.

- Like *art*'s createEngine interface, NuRandomService::createEngine will return a reference to the *art*-managed engine.

```
- long seed = ServiceHandle<NuRandomService>{}->createEngine(...);
+ CLHEP::HepRandomEngine& engine = ServiceHandle<NuRandomService>{}->createEngine(...);
```

- No code breaks in LArSoft; not sure about experiment repositories.

NuRandomService interface change

Before

```
class MyProducer {
public:

  MyProducer(ParameterSet const& pset)
  {
    ServiceHandle<NuRandomService>{}
      ->createEngine(...);
  }

  void produce(art::Event& e) override
  {
    auto& engine =
      ServiceHandle<RandomNumberGenerator>{}
        ->getEngine(...);
    CLHEP::RandFlat flatDist{engine};
    flatDist.fire(...);
  }
};
```

NuRandomService interface change

Before

```
class MyProducer {
public:

  MyProducer(ParameterSet const& pset)
  {
    ServiceHandle<NuRandomService>{}
      ->createEngine(...);
  }

  void produce(art::Event& e) override
  {
    auto& engine =
      ServiceHandle<RandomNumberGenerator>{}
        ->getEngine(...);
    CLHEP::RandFlat flatDist{engine};
    flatDist.fire(...);
  }
};
```

Expensive operations:

- *`ServiceHandle` created for each event*
- *`getEngine` called on each event*
- *`RandFlat` distribution created for each event*

NuRandomService interface change

Before

```
class MyProducer {
public:

  MyProducer(ParameterSet const& pset)
  {
    ServiceHandle<NuRandomService>{}
      ->createEngine(...);
  }

  void produce(art::Event& e) override
  {
    auto& engine =
      ServiceHandle<RandomNumberGenerator>{}
        ->getEngine(...);
    CLHEP::RandFlat flatDist{engine};
    flatDist.fire(...);
  }
};
```

After

```
class MyProducer {
  CLHEP::RandFlat flatDist_;

public:

  MyProducer(ParameterSet const& pset)
    : flatDist_{ServiceHandle<NuRandomService>{}
                  ->createEngine(...)}
  {}

  void produce(art::Event& e) override
  {
    flatDist_.fire(...);
  }
};
```

NuRandomService interface change

Before

```
class MyProducer {
public:

  MyProducer(ParameterSet const& pset)
  {
    ServiceHandle<NuRandomService>{}
      ->createE
  }

  void produce(art::Event& e) override
  {
    auto& engine =
      ServiceHandle<RandomNumberGenerator>{}
        ->getEngine(...);
    CLHEP::RandFlat flatDist{engine};
    flatDist.fire(...);
  }
};
```

createEngine returns art-owned reference to engine; no need to directly interact with it

After

```
class MyProducer {
  CLHEP::RandFlat flatDist_;

public:

  MyProducer(ParameterSet const& pset)
    : flatDist_{ServiceHandle<NuRandomService>{}
                  ->createEngine(...)}
  {}

  void produce(art::Event& e) override
  {
    flatDist_.fire(...);
  }
};
```

NuRandomService interface change

Before

```
class MyProducer {
public:

  MyProducer(ParameterSet const& pset)
  {
    ServiceHandle<NuRandomService>{}
      ->createE
  }

  void produce(art::Event& e) override
  {
    auto& engine =
      ServiceHandle<RandomNumberGenerator>{}
        ->getEngine(...);
    CLHEP::RandFlat flatDist{engine};
    flatDist.fire(...);
  }
};
```

createEngine returns art-owned reference to engine; no need to directly interact with it

After

```
class MyProducer {
  CLHEP::RandFlat flatDist_;

public:

  MyProducer(ParameterSet const& pset)
    : flatDist_{ServiceHandle<NuRandomService>{}
                  ->createEngine(...)}
  {}

  void produce(art::Event& e) override
  {
    flatDist_.fire(...);
  }
};
```

Feature branch to be added soon.

Header guards are for headers

- I've seen many instances of:

```
// MyProducer_module.cc
#ifndef MyProducer_module_h
#define MyProducer_module_h
…
#endif
```

- An implementation file (".cc" file) should never be included in another file—i.e. there should never be a need for a header guard.
- Please do not put them in files that are not intended to be included.

To reconfigure or not to reconfigure…

- Consider this code:

```
class MyProducer {
  LargeObject obj_;

public:

  MyProducer(ParameterSet const& pset)
  {
    reconfigure(pset);
  }

  void reconfigure(ParameterSet const& p)
  {
    obj_ = LargeObject{p.get<std::string>("some_label")};
  }
};
```

To reconfigure or not to reconfigure…

- Consider this code:

`LargeObject()` called before `reconfigure` is called

```
class MyProducer {
  LargeObject obj_;

public:

  MyProducer(ParameterSet const& pset)
  {
    reconfigure(pset);
  }

  void reconfigure(ParameterSet const& p)
  {
    obj_ = LargeObject{p.get<std::string>("some_label")};
  }
};
```

To reconfigure or not to reconfigure…

- Consider this code:

LargeObject( ) called before reconfigure is called

LargeObject(string const&) called

```
class MyProducer {
  LargeObject obj_;

public:

  MyProducer(ParameterSet const& pset)
  {
    reconfigure(pset);
  }

  void reconfigure(ParameterSet const& p)
  {
    obj_ = LargeObject{p.get<std::string>("some_label")};
  }
};
```

To reconfigure or not to reconfigure…

- Consider this code:

LargeObject( ) called before reconfigure is called

LargeObject(string const&) called

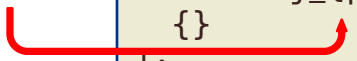
```
class MyProducer {
  LargeObject obj_;

public:

  MyProducer(ParameterSet const& pset)
  {
    reconfigure(pset);
  }

  void reconfigure(ParameterSet const& p)
  {
    obj_ = LargeObject{p.get<std::string>("some_label")};
  }
};
```

To boot: ***module reconfiguration is not supported by art***

To reconfigure or not to reconfigure…

- Consider this code:

LargeObject(string const&) called



```
class MyProducer {
  LargeObject obj_;

public:

  MyProducer(ParameterSet const& pset)
    : obj_{p.get<std::string>("some_label")}
  {}
};
```

To reconfigure or not to reconfigure…

- Consider this code:

LargeObject(string const&) called

```
class MyProducer {
  LargeObject obj_;

public:

  MyProducer(ParameterSet const& pset)
    : obj_{p.get<std::string>("some_label")}
  {}
};
```

Please do not add `reconfigure` functions to your modules.

Takeaways

- Think about what you are coding—every character counts.
  - Do you know why your function is private or public?
  - Do you know why you're creating a class/service instead of a function?

- You should be able to ask yourself such questions, and anybody else. Ask me!

- Over the next year, members of the SciSoft team will be working on LArSoft code, preparing it for multi-threading.
  - Getting there will take time, and it will be gradual.
  - We intend to polish code as we go.
  - This is also a time for the *art* project to determine how to better support users.

Thanks for your time.