Optional user actions and MT

Mihaly Novak (CERN, EP-SFT)

Getting Started with **`Geant4`** at CERN, Geneva (Switzerland)

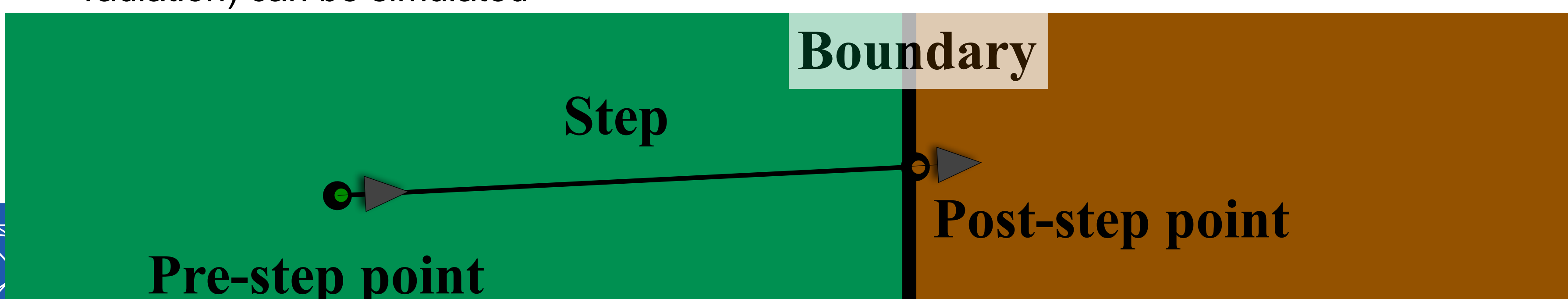
- Key concepts of **`Geant4`** tracking and related optional User-Actions
- Muti-threaded (MT) related notes on the User-Actions and Run

Optional user actions and MT

KEY CONCEPTS OF GEANT4 TRACKING

- **G4Track**:
  - a **G4Track** object represents/describes **the state of a particle** that is under simulation in a given instant of the time (i.e. **a** given **time point**)
  - a snapshot of a particle **without** keeping any **information regarding the past**
  - its **G4ParticleDefinition** stores **static** particle **properties** (charge, mass, etc.) as it describes a particle type (e.g. **G4Electron**)
  - its **G4DynamicParticle** stores **dynamic** particle **properties** (energy, momentum, etc.)
  - while all **G4Track**-s, describing the same particle type, share the same, unique **G4ParticleDefinition** object of the given type (e.g. **G4Electron**) while each individual track has its own **G4DynamicParticle** object
  - the **G4Track** object is propagated in a step-by-step way during the simulation and the dynamic properties are continuously updated to reflect the current state
  - manager: **G4TrackingManager**; optional user hook: **G4UserTrackingAction**
  - step-by-step? what about the difference between two such states within a step?

- **G4Step:**
  - a **G4Step** object can provide the information regarding the **change in** the **state of the particle (**that is under tracking) **within a** simulation **step** (i.e. **delta)**
  - has two **G4StepPoint**-s, pre- and post-step points, that stores information (position, direction, energy, material, volume, etc…) that belong to the corresponding point (space/time/step)
  - these are updated in a step-by-step way: the post-step point of the previous step becomes the pre-step point of the next step (when the next step starts)
  - **(important)** if a step is limited by the geometry (i.e. by a volume boundary), the post-step point:
    - **physically** stands **on the boundary** (the step status of the post step point i.e. `G4Step::GetPostStepPoint()->GetStepStatus()` is `fGeomBoundary`)
    - **logically** belongs **to the next volume**
    - since these "*boundary*" **G4Step**-s have information both regarding the previous and the next volumes/materials, boundary processes (e.g. reflection, refractions and transition radiation) can be simulated



- **G4Step:**
  - a **G4Step** object can provide the information regarding the **change in** the **state of the particle (**that is under tracking) **within a** simulation **step** (i.e. **delta)**
  - has two **G4StepPoint**-s, pre- and post-step points, that stores information (position, direction, energy, material, volume, etc…) that belong to the corresponding point (space/time/step)
  - these are updated in a step-by-step way: the post-step point of the previous step becomes the pre-step point of the next step (when the next step starts)
  - **(important)** if a step is limited by the geometry (i.e. by a volume boundary), the post-step point:
    - **physically** stands **on the boundary** (the step status of the post step point i.e. `G4Step::GetPostStepPoint()->GetStepStatus()` is `fGeomBoundary`)
    - **logically** belongs **to the next volume**
    - since these "*boundary*" **G4Step**-s have information both regarding the previous and the next volumes/materials, boundary processes (e.g. reflection, refractions and transition radiation) can be simulated
  - the **G4Track** object, that is under tracking i.e. generates information for the **G4Step** object, can be obtained from the step by the **G4Step::GetTrack()** method and the other way around **G4Track::GetStep()**
  - manager: **G4SteppingManager**; optional user hook: **G4UserSteppingAction**

- **`G4Step` – `G4UserSteppingAction`:**
  - optional user action class with the possibility to obtain information after each simulation steps
  - **`virtual UserSteppingAction(const G4Step* theStep)`** method:
    - is **called at the end of each step** by **`G4SteppingManager`**
    - providing access to the **`G4Step`** object representing the simulation step that has just done (see this class in Geant4 `/source/tracking/include/G4UserSteppingAction.hh`)
  - users can implement their own **`YourSteppingAction`** class by:
    - extending the **`G4UserSteppingAction`** class
    - providing their own implementation of the method mentioned above
    - creating and registering the corresponding object in the **`ActionInitialisation::Build()`** interface method (see later)

How to get information regarding the simulation when the **`G4Step* theStep`** is given?

How to get information regarding the simulation when the `G4Step* theStep` is given?

```
// get the pre-step point
G4StepPoint*         preStp = theStep->GetPreStepPoint();
// get the volume which the step was done
G4VPhysicalVolume* physVol = preStp->GetPhysicalVolume();
// get the energy deposit and length of the step
G4double            stpEdep = theStep->GetTotalEnergyDeposit();
G4double          stpLength = theStep->GetStepLength();
// get the track
G4Track*           theTrack = theStep->GetTrack();
// get the static and dynamic particle properties
const G4ParticleDefinition* partDef =  theTrack->GetParticleDefinition();
const G4DynamicParticle*    partDyn =  theTrack->GetDynamicParticle();
// get the charge of the particle that is under tracking in this step
G4double         partCharge = partDef->GetPDGCharge();
// get the post step point kinetic energy
G4double        postStpEkin = theStep->GetPostStepPoint()->GetKineticEnergy();
// or since the track is already updated to reflect the post-step point
// G4double        postStpEkin = partDyn->GetKineticEnergy();
// which is different in case of the pre-step point kinetic energy that can be
// obtained only from the pre-step point as
G4double         preStpEkin = preStp->GetKineticEnergy();
```

Key concepts of **`Geant4`** tracking: from an instant to a complete simulation time coverage

- **`G4Event`**:
  - a **`G4Event`** is the basic simulation unit that represents a **set of `G4Track` objects**
  - **at the beginning** of an event:
    - **primary G4Track object**(s) is(are) **generated** (with their static and initial dynamic properties) and pushed **to a track-stack**
    - **one G4Track object** is **popped from** this track-**stack** and transported/tracked/simulated:
      - ✳the track object **is propagated in a step-by-step way** and its dynamic properties as well as the corresponding **`G4Step`** object are updated at each step
      - ✳the step is limited either by physics interaction or geometry boundary
      - ✳**transportation** (to the next volume through the boundary) will take place in the later while **physics** interaction in the former case
      - ✳**secondary G4Track-s**, **generated by** these **physics interactions**, are **also pushed to** the track-**stack**
      - ✳a **G4Track object** is kept tracking till:
        - + leaves the outermost (World) volume i.e. goes out of the simulation universe
        - + participates in a destructive interaction (e.g. day or photoelectric absorption)
        - + its kinetic energy becomes zero and doesn't have interaction that can happen “at-rest”
        - + the user decided to (artificially) stop the tracking and kill
      - ✳when one track object reaches its termination point, **a new G4Track object** (either secondary or primary) is popped from the stack for tracking
    - processing an event will be terminated when there is **no any G4Track objects in the track-stack**
  - **at the end** of an event, the corresponding **`G4Event`** object will store its input i.e. the list of primaries (and possible some of its outputs like hits or trajectory collection)

- **G4Event – G4UserEventAction:**
  - optional user action class, with the possibility to get the control before/after an event processing
  - `virtual BeginOfEventAction(const G4Event* anEvent):`
    - **called before** a new **event processing** starts by the **G4EventManager**
  - `virtual EndOfEventAction(const G4Event* anEvent):`
    - **called after** an **event processing completed** by the **G4EventManager**
  - in both cases, the **G4EventManager** provides access to the corresponding **G4Event** object (see this class in Geant4 `/source/event/include/G4UserEventAction.hh`)
  - users can implement their own **YourEventAction** class by:
    - extending the **G4UserEventAction** class
    - providing their own implementation of the two methods mentioned above
    - creating and registering the corresponding object in the **ActionInitialisation::Build()** interface method (see later)
  - at the beginning of the event (`BeginOfEventAction`) we usually clear some data structures that we want to use to accumulate information during the processing of the current event (populated after each step in the **UserSteppingAction**) while at the end of the event (`EndOfEventAction`) we usually write the accumulated information to an upper (Run global) level

- **G4Run** (as already discussed in the DetectorConstruction):
  - **G4Run** is a collection of **G4Event-s** (a **G4Event** is a collection of **G4Track**-s)
  - during a run, events are taken and processed one by one in an event-loop
  - **before the start of a run** i.e. at run initialisation (`G4RunManager::Initialize()`): the **geometry** is **constructed** and **physics** is **initialised**
  - **at the start of a run** (`G4RunManager::BeamOn()`): the **geometry is optimised** for tracking (voxelization), **physics tables are built**, then event processing starts i.e. entering into the event-loop
  - as log as the event processing is running, i.e. during the run, the user cannot modify **neither the geometry** (i.e. the detector setup) **nor the physics** settings
  - they **can be changed** though **between run**-s but the **G4RunManager** needs to be informed (re-optimise or re-construct geometry, re-build physics tables):
    - if the **geometry** has been changed, depending on the modifications:
      – `GeometryHasBeenModified()` re-voxelization but no re-`Construct`
      – `ReinitializeGeometry()` complete re-`Construct`

      or with the UI commands **/run/geometryModified** or **/run/reinitializeGeometry**
    - same for the **physics**: `PhysicsHasBeenModified()` or **/run/physicsModified**
  - manager: **G4RunManager**; optional user hook: **G4UserRunAction**

- **`G4Run - G4UserRunAction:`**
  - optional user action class, with the possibility to get the control before/after a run and to provide custom run-object (see later)
  - `virtual BeginOfRunAction(const G4Run* aRun):`
    - **called before** the **run starts** i.e. **before the first event processing** starts by the **`G4RunManager`**
  - `virtual EndOfRunAction(const G4Run* aRun):`
    - **called after** a **run completed** i.e. **after the last event processing competed** by the **`G4RunManager`**
  - in both cases, the **`G4RunManager`** provides access to the corresponding **`G4Run`** object (see this class in Geant4 `/source/run/include/G4UserRunAction.hh`)
  - users can implement their own **`YourRunAction`** class by:
    - extending the **`G4UserRunAction`** class
    - providing their own implementation of the two methods mentioned above
    - creating and registering the corresponding object in the **`ActionInitialisation::Build()`** interface method (see later)
  - note, at the beginning of the run (`BeginOfRunAction`) we usually allocate/initialise some data structures/histograms that we want to use during the whole run to collect the final simulation results that we usually print out at the end of the run (`EndOfRunAction`)

- **G4Run - G4UserRunAction:**
  - an additional method, `virtual G4Run* GenerateRun()` is also available:
    - users can implement their own **YourRun** class by:
      - ✳extending the G4Run class and defining their own run-global data structure (i.e. quantities, objects to be collected during the complete run as the result of the simulation/ run)
      - ✳instantiation of the custom **YourRun** object needs to be done in this `GenerateRun()` method of **YourRunAction**
  - this will be invoked by the **G4RunManager** at initialisation to generate your (extended) **YourRun** instead of the (base) **G4Run** object
  - the generated **YourRun** object can be accessed during the simulation as the Run (in UserAction-s) and can be populated by information (after each simulation Step, Event, etc.)
  - we will add our method (`RunSummary()`) that prints the final information at the end of the run, i.e. will be invoked from **YourRunAction**`::EndOfRunAction` method to report the final results collated during the entire run
  - see next section on its MT related parts !!!

- **G4Run - G4UserRunAction:**
  - an additional method, `virtual G4Run* GenerateRun()` is also available:
    - users can implement their own **YourRun** class by:
      - ✳extending the G4Run class and defining their own run-global data structure (i.e. quantities, objects to be collected during the complete run as the result of the simulation/ run)
      - ✳instantiation of the custom **YourRun** object needs to be done in this `GenerateRun()` method of **YourRunAction**
  - this will be invoked by the **G4RunManager** at initialisation to generate your (extended) **YourRun** instead of the (base) **G4Run** object
  - the generated **YourRun** object can be accessed during the simulation as the Run (in UserAction-s) and can be populated by information (after each simulation Step, Event, etc.)

Time to add our own user actions to the application!

BUT THE NEXT SECTION FIRST!

Optional user actions and MT

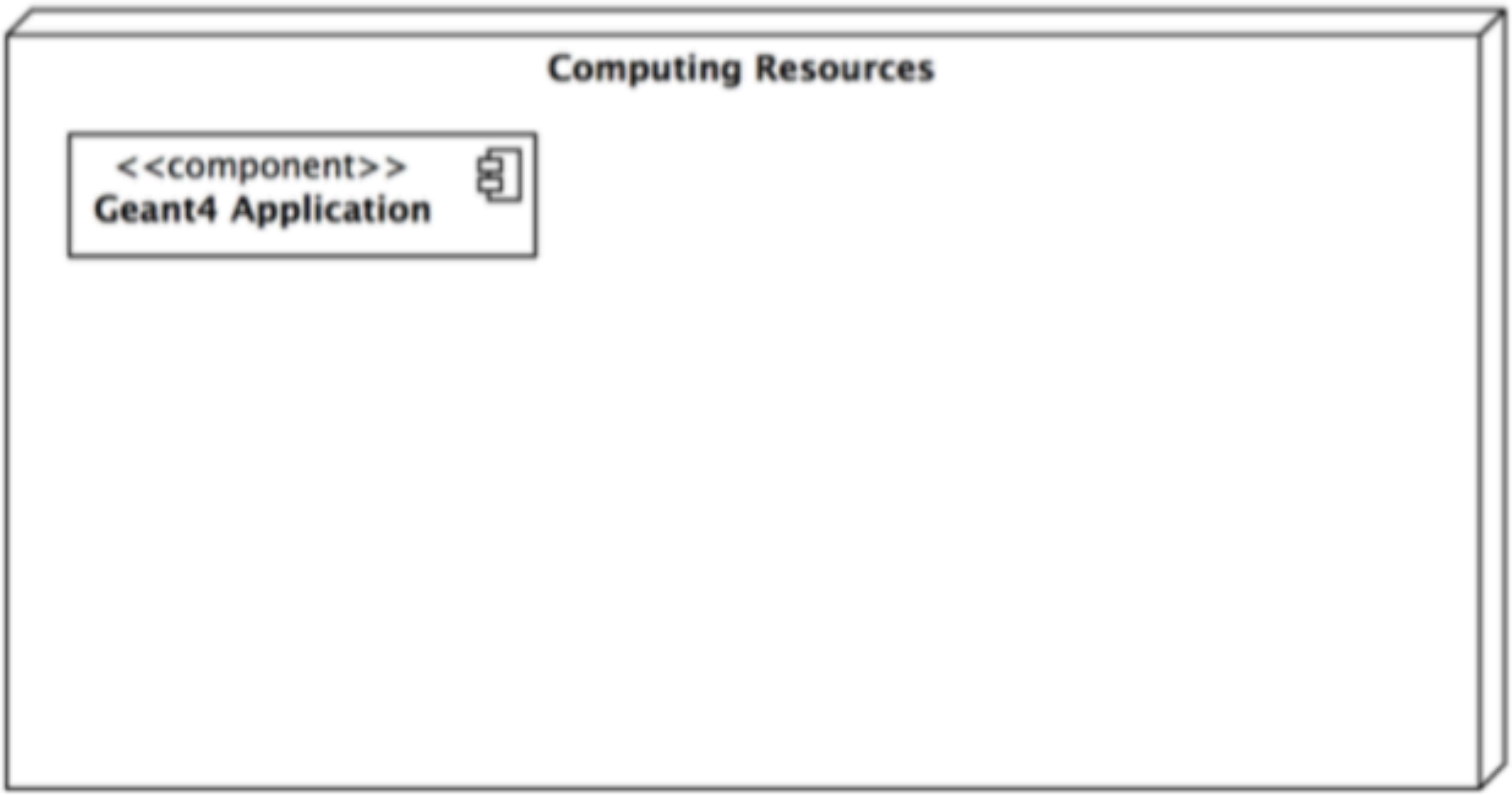
MULTI-THREADED RELATED NOTES ON THE USERACTIONS AND RUN

- multiple CPU-s and/or multiple processing units (i.e. cores) per CPU
- offers the possibility of executing parallel tasks
- particle transport simulation is **nearly embarrassingly parallel**:
  - ➡ can be divided to completely independent parts
  - ➡ then these parts can be computed separately, i.e. parallel
  - ✓ in the Geant4 simulation, the individual Events are independent
  - ✓ so they can be simulated independently, i.e parallel

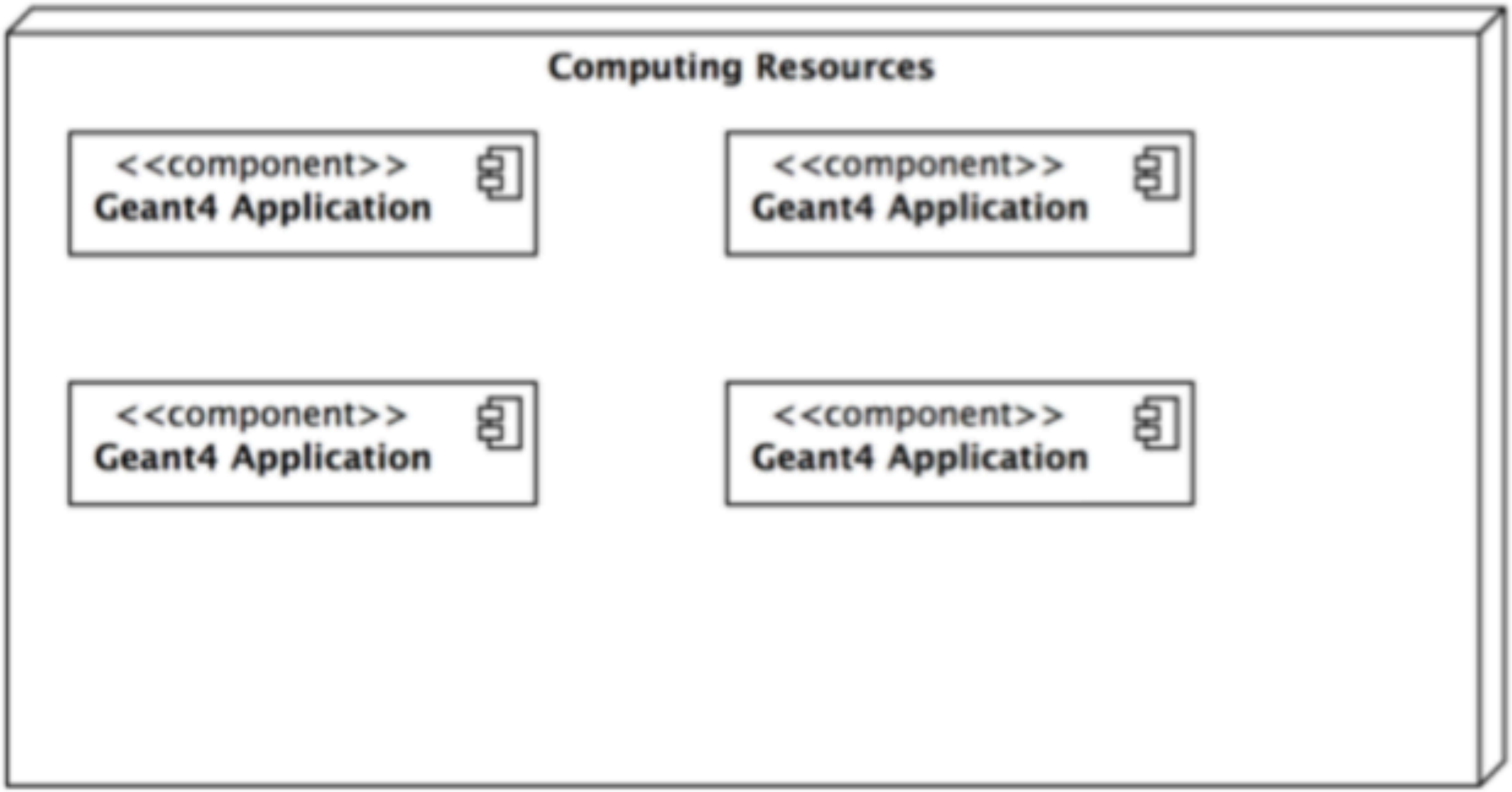
- **multi-threading** v.s. **multi-processing**: (if N Events to simulate and C c.units)
  - **multi-processing(MP)**: (can be done in all cases)
    - execute the same program C-times, with different random number seeds, taking N/C events each: → **distinct memory** place, no data shared between the C processes (takes C-times memory)
  - **multi-threading(MT)**: (only if the program is written such that !)
    - execute the program only once, that can utilise multiple threads, taking N/C events each → **shared memory** place, data can be shared between the C threads (saves lost of memory)
    - ➡ requires some data to be distributed before the start of the parallel processing
    - ➡ requires some other data to be collected after the end of the parallel processing

MT related notes on UserActions and Run (from M. Asai)

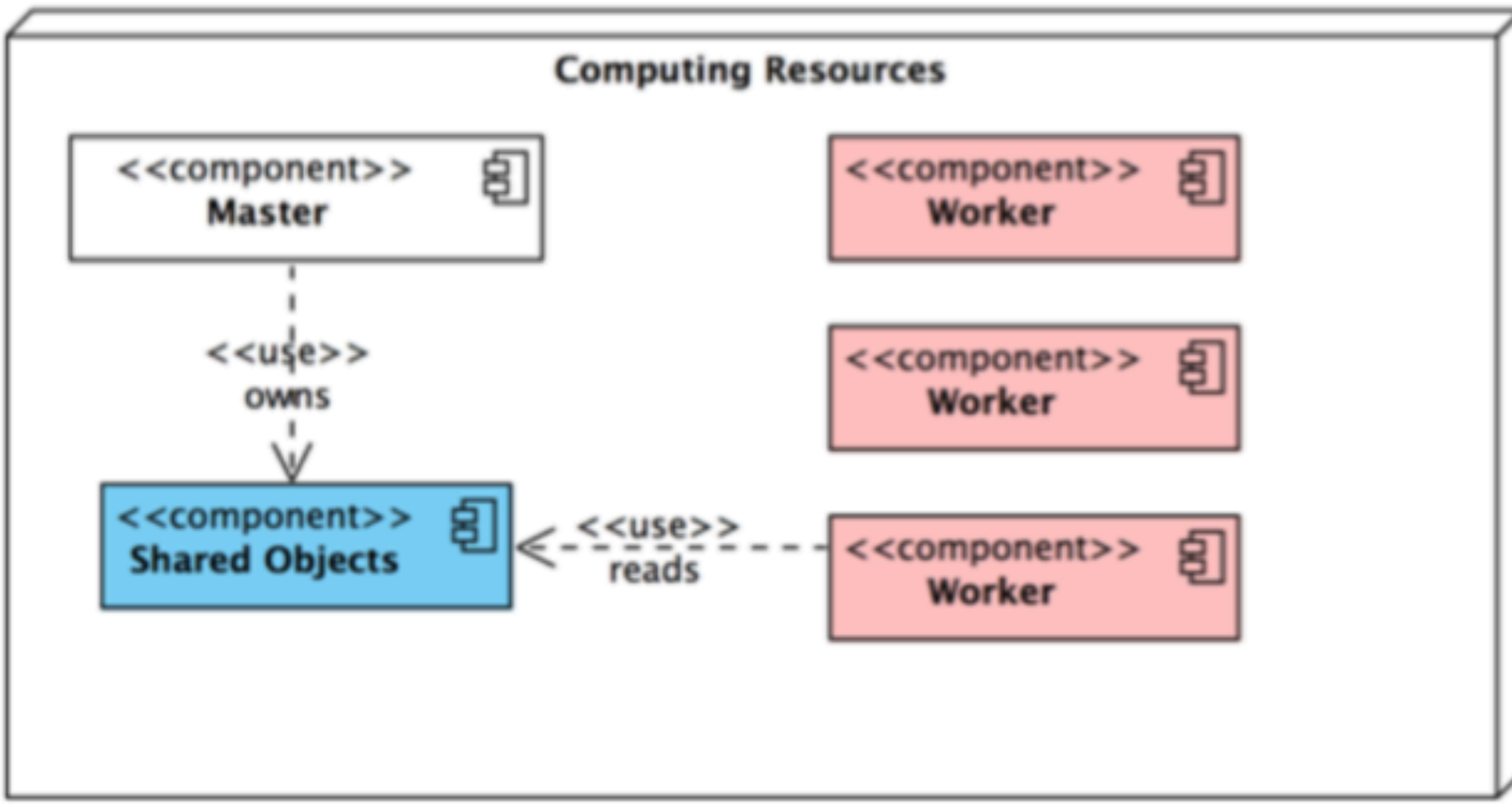
Sequential application:



Multi-processing : C copies of the same program with their own memory space

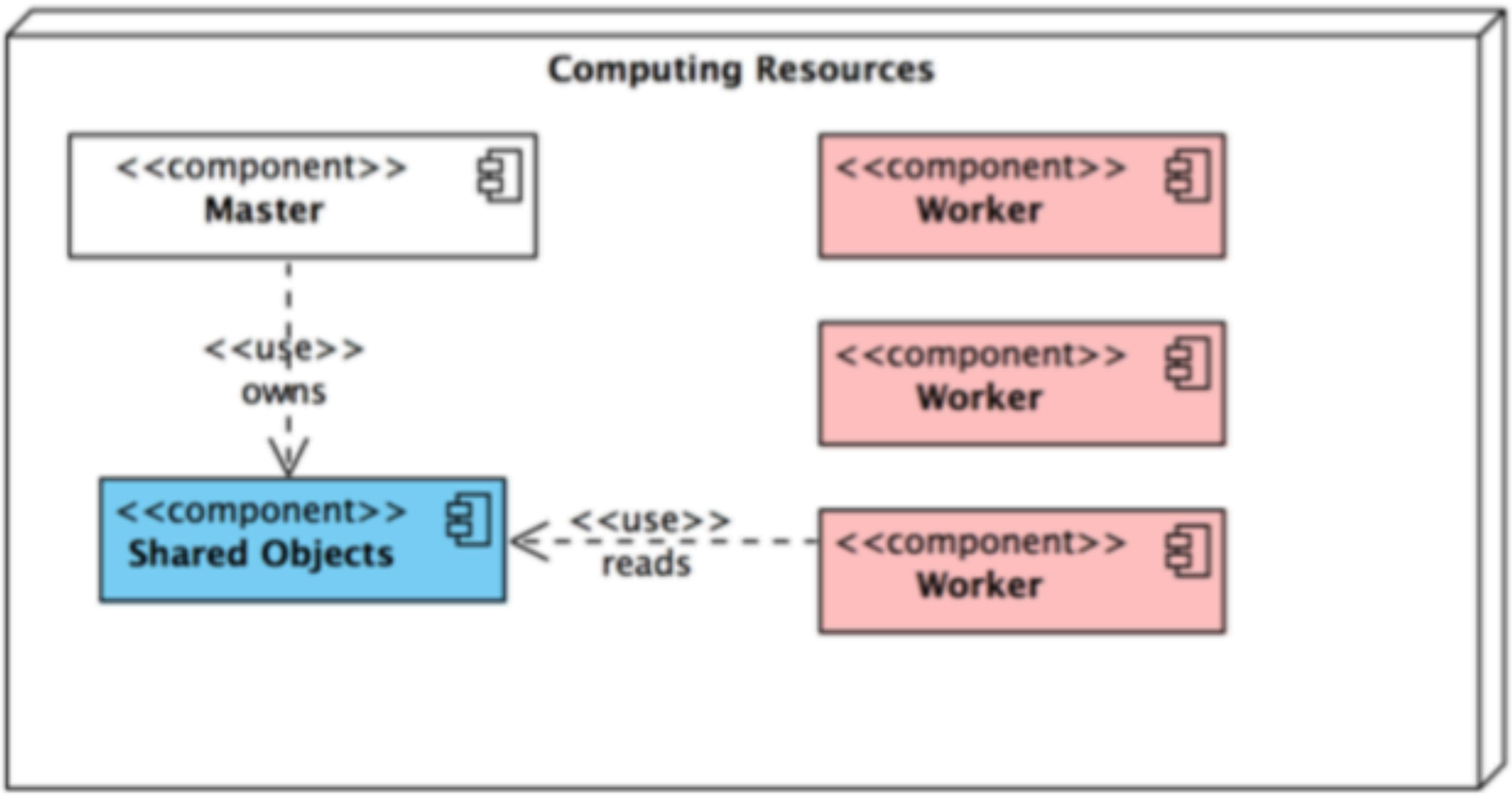


Multi-threading: one program with its **Master** and **Worker** threads own memory space



MT related notes on UserActions and Run (from M. Asai)

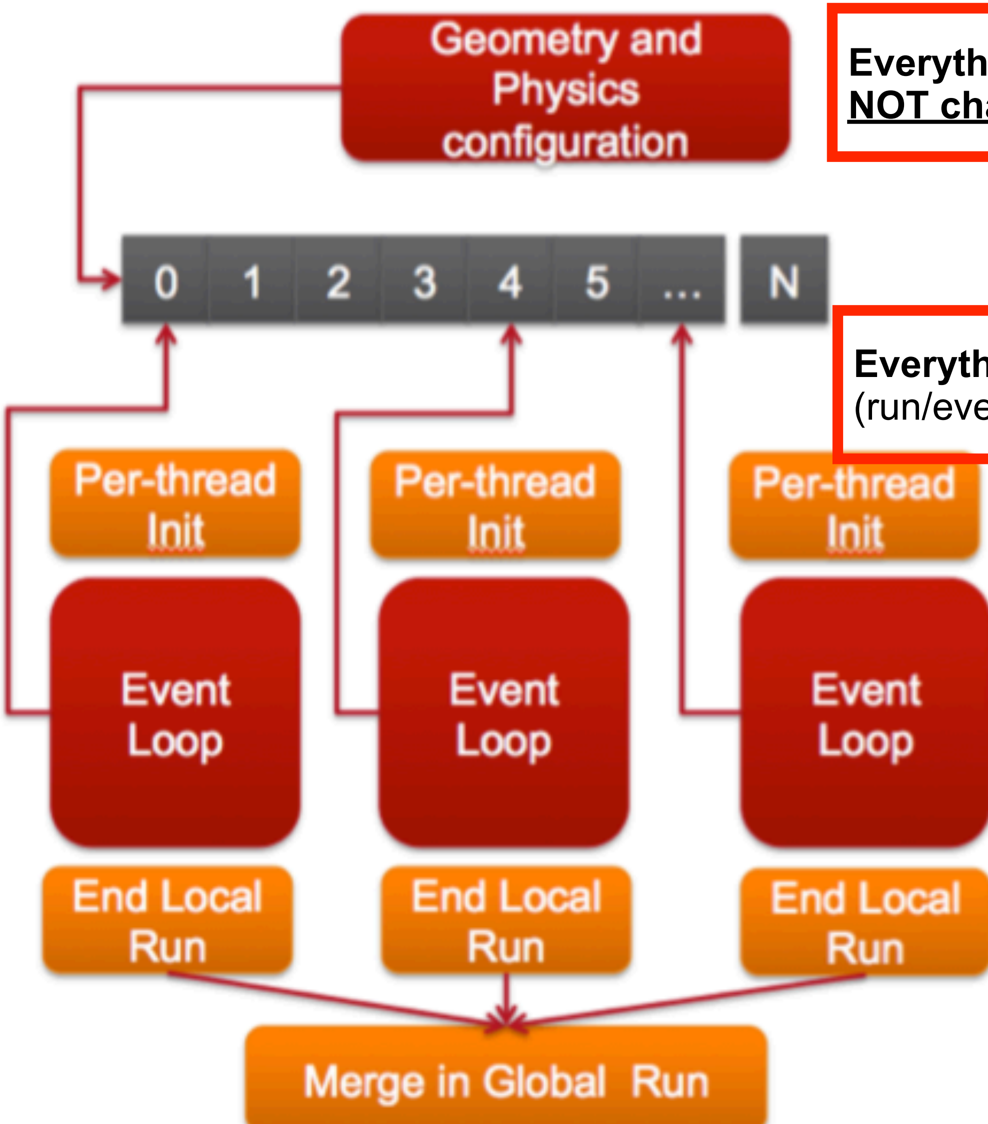
Multi-threading: one program with its **Master** and **Worker** threads own memory space



MT related notes on UserActions and Run (from M. Asai)

Multi-threading: one program with its **Master** and **Worker** threads own memory space

Everything, that the (worker) threads DO NOT change: shared between workers (only



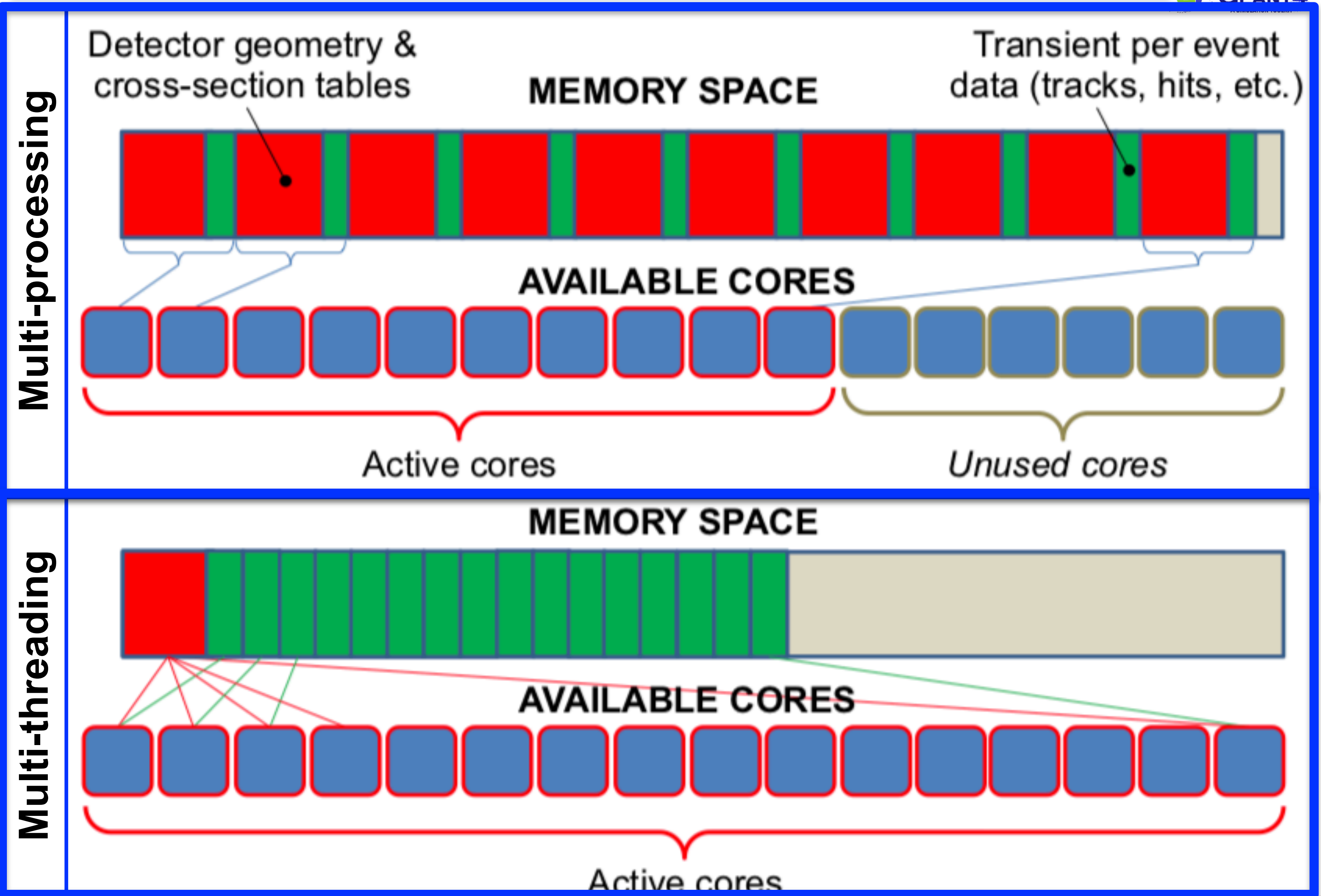
Per-event seeds pre-prepared in a "queue"

Everything, that the (worker) threads MODIFY (run/event/track/step, etc.)**: thread-local data**

Threads compete for next event to be processes

Command line scoring and G4tools automatically merge results from threads

MT related notes on UserActions and Run (from M. Asai)



How it is achieved in Geant4?:

- the **Detector-Construction** and the **Physics-List** need to be **created directly in the main** program and **registered directly in the** `G4RunManager` object
- all **User-Actions** need to be **created** and **registered** in the **User-Action-Initialisation** (**including the** only **mandatory Primary-Generator-Action** as well as **all** other, **optional User-Actions**)
- we have already created `YourActionInitialization` for this, derived from the `G4VUserActionInitialization` interface, and the mandatory `YourPrimaryGeneratorAction` is already construct and registered in its `YourActionInitialization::Build()` method
- the base `G4VUserActionInitialization` has two build methods:
  - `BuildForMaster()`: invoked **only by the Master**
  - `Build()` : invoked **by all Worker**-s (we used only this so far)
- we will understand why these two and what they are for

- `BuildForMaster()`: invoked **only by the Master**
  - the **only user action,** that is supposed to be created and registered **in this build method**, is the `G4UserRunAction`

- `Build()`: invoked **by all the Worker**-s
  - **all user actions,** including the mandatory `G4VUserPrimaryGeneratorAction` and all other optional `G4UserRunAction`, `G4UserEventAction`, etc., must be constructed and registered **in this build method**

- Consequence: (the why?)
  - both **the Master** and **all the Worker**-s have their own **`G4UserRunAction`** object
  - before the run starts, all invokes the `G4Run*` **`G4UserRunAction::`**`GenerateRun()` (interface) method of its own run action object
  - therefore, both **the Master** and **all the Worker**-s will have **their own `G4Run` object**

  - **<u>during</u> the run** (i.e. between Begin and End of run):
    ➡ the **Master's `G4Run` object** is **<u>not used</u>**
    ➡ **Worker**-s **<u>populate their own</u> `G4Run` object** with the information processed by

  - **<u>at the end</u> of the run**:
    ➡ the **Master invokes** the **`G4Run::`**`Merge(const G4Run* run_i)` method **of its own `G4Run` object**
    ➡ passing by **all the Worker**-s **`G4Run` object** (pointer) **as arguments** (i.e. `run_i`)
    ➡ equivalent to:

$$runMaster = \sum_i runWorker_i$$

  - **<u>the end</u>**: the **Master's `G4Run`** object contains all information collected during the run

- Consequence: (the why?)
  - **to collect our** own **data** while the simulation is running:
    - e.g. mean or distribution of energy deposit in the target per event
    - i.e. beyond what is already in the **base `G4Run`**
    - we **derive our own** run from **the base `G4Run`**: this will be **`YourRun`**
    - this will contain our data that we are interested to collect during the run (beyond what is already in the base **`G4Run`**)
    - we will create our own **`YourRun`** object in our **`YourRunAction::GenerateRun`** (**`YourRun`** is a **`G4Run`** as well since it extends **`G4Run`** with our data objects)
    - therefore, we need to implement the **`YourRun`**`::Merge(const G4Run* run_i)` method :
      - ➡ to tell how to add our data objects when this method is invoked (by the **Master** on its own **`YourRun`** object and passing **`Worker`**-s **`YourRun`** objects)
      - ➡ calling **`G4Run::Merge(run)`** at the end to merge all data that are in the base **`G4Run`**
    - (e.g. we will implement how to add our energy deposit data or our histogram that we will store in **`YourRun`**)

- Other consequences (just to note some):
  - the **Master** does not have any user actions but **RunAction**
  - **Worker**-s **have all** user **actions**:
    - their **RunAction** can be the **same as for Master or different**
      - ➡you might have a dedicated RunAction **for the Master** that **implements** writing the final data in its `G4UserRunAction::EndOfRunAction()`
      - ➡other **for Worker**-s as they have **nothing to do** at the **end of the run**
      - ➡we will have the same RunAction (**`YourRunAction`**) and use the `G4UserRunAction::IsMaster()` method to **differentiate Master/Worker**
    - each has **their own** set of **objects** from the actions, **but** all have **the same type**
      - ➡e.g. all **Worker**-s **have their own object** for primary event generation and they are all **`YourPrimaryGeneratorAction`** type
      - ➡need to **ensure that** the individual **Worker**-s generate **distinct set of events**
      - ➡otherwise, all **Worker**-s simulate the same events!
      - ➡important only when events are different: not in our simple particle gun example! **`G4Run`**

Optional user actions and MT

SO WHAT'S NEXT? SLIDE 14: WE IMPLEMENT THE OPTIONAL (RUN, EVENT, STEPPING) USER ACTIONS WITH OUR OWN RUN TO COLLECT DATA