Programming Project

Terry Marris January 2010

2 Specification

In the specification we state *what* is to be done.

You are not likely to be given a detailed specification. So you have to write one for yourself. In doing so you are forced to think about the system you are going to create, you discover points you had not previously thought about, you develop a deeper understanding of the system. Further, a specification provides criteria against which you can evaluate your finished program.

2.1 Data Flow Diagram - Entries and Results Management System

A person makes an application by completing an application form and sending it, together with the fee, to The Organisers. The organisers check the form and the fee and, if both are correct, record the applicant's details. The applicants name, age, address, ... are recorded in the entries file.

A week before the race is due to start, runners are sent their pack. A pack includes instructions on when and where to start, the runner's number to be pinned on the front and back of their vest, and a race-timing chip that is to be attached to the ankle.

On race day, the time race started is recorded, as is the time when the runner passes over the finishing line. We are dealing with race time here, rather than personal time. The personal time starts when the runner crosses the start line. It is usually after the race time start because it can take several minutes for the runner to reach the start line once the race has started. At the end of the race volunteers collect the race-timing chip from each competitor.

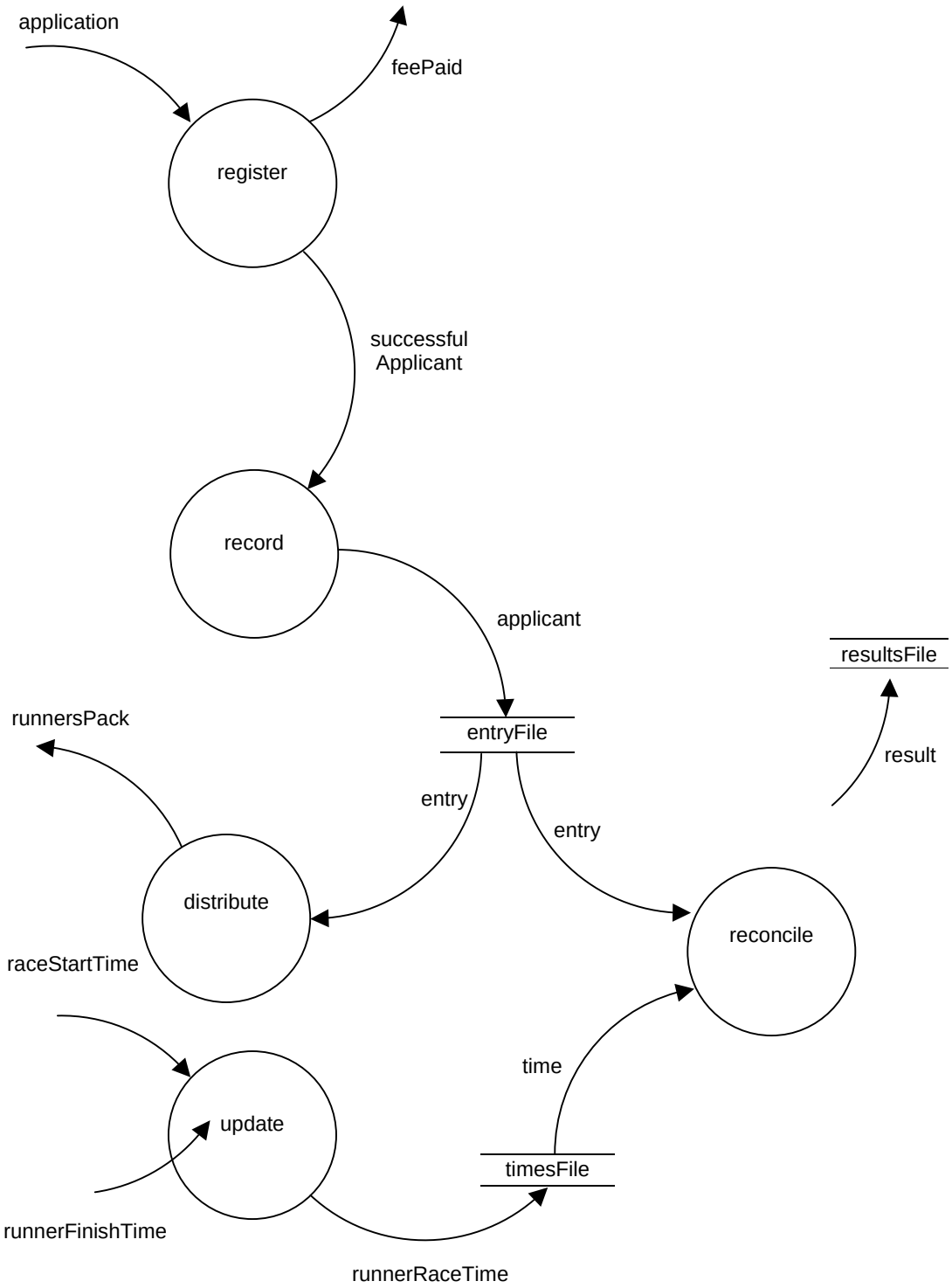
A runner's race time is recorded in the times file. A runner's race time is matched, by competitor (or runner) number, with the name of the runner from the entry file, to create the results file in time order. This file is used to print the results.

We continue the analysis of the system with a data flow diagram. We use the Tom De Marco notation because it is simple and meets our requirements. The arrows represent data. The circles represent processes. The parallel lines represent data stores e.g. files.

In constructing a data flow diagram you begin to understand what the main processes could be, the files that may be required, and the data involved.

You may remember developing data flow diagrams in the Systems Analysis unit.

Data Flow Diagram - Entries and Results Management System



The explanation is in the narrative on the previous page. It is customary to support a data flow diagram with a data dictionary. But I will not bother because that level of detail could be left to the design or implementation stages.

You could use plain text to specify what the system does, to explain any restrictions on variables, and the connections between them. But I shall use the Z Notation. It does not matter if you do not understand Z - just read the accompanying descriptive narrative to understand what is going on.

2.2 Specification - Marathon Management System

Entry

An entry has a competitor number, a race category, a name and an address. We introduce, as given types,

```
[ADDRESS,NAME]
```

A category is either half marathon or full marathon or withdrawn.

```
CATEGORY ::= halfMarathon | fullMarathon | withdrawn
```

We define competitor number as an integer: ...-2, -1, 0, 1, 2, 3, ...

```
NUMBER == ℤ
```

We introduce the entry type.

```
┌─Entry──────────────────────────────────────────────┐
│ category : CATEGORY                                │
│ name : NAME                                        │
│ address : ADDRESS                                  │
└────────────────────────────────────────────────────┘
```

There is a limit to the number of entries we can have. So we define

```
│ maxEntries : ℤ
```

and leave its value undefined.

We define entry base, the collection of entries.

```
┌─EntryBase──────────────────────────────────────────┐
│ entry : NUMBER ⤔ Entry                             │
├────────────────────────────────────────────────────┤
│ # entry ≤ maxEntries                               │
│ entry = { n : NUMBER; e : Entry | n > 0 ∧ n ≤ maxEntries
│           • n ↦ e }                                │
└────────────────────────────────────────────────────┘
```

entry is a set of *(NUMBER, Entry)* pairs. No two entries can have the same competitor number. Competitor numbers must be more than zero and not more than *maxEntries*. Given a competitor number we can find the corresponding entry. We cannot have duplicate entries. If two different entries have the same name, they must have different addresses. The order in which entries are held is not significant.

Initially, to begin with, there are no entries.

```
┌─InitEntryBase──────────────────────────────────────┐
│ EntryBase'                                         │
├────────────────────────────────────────────────────┤
│ entry' = ∅                                         │
└────────────────────────────────────────────────────┘
```

We add a new entry to the entry base. We require that the new competitor number has not been used before and that the new entry is not already in the entry base.

```
AddEntry
ΔEntryBase
n? : NUMBER
e? : Entry
─────────────────
n? > 0
n? ∉ dom entry
e? ∉ ran entry
entry′ = entry ∪ {n? ↦ e?}
```

Runner

We need to record the race time for each runner. By race time we mean the time taken to complete the course. This is calculated as the difference between the race start time e.g. 10:15 am and the runner's finish time e.g. 12:30 pm. The individual runner's start time is ignored. It can take several minutes for a runner to cross the starting line after the starting gun or klaxon has sounded. We introduce, as a given type

```
[DATETIME]
```

and define race time as an integer number of seconds.

```
RACETIME == ℤ
```

We need a function to calculate the race time from race start time and runners finish time.

```
raceTime : DATETIME × DATETIME ⤖ RACETIME
─────────────────
∀ r : ran raceTime • r ≥ 0
```

We intend the first *DATETIME* to represent the race start time, and the second *DATETIME* the runners finish time. We cannot have a negative race time. A race time of zero means either the competitor did not start, or started but did not finish. We are not concerned with non-starters and non-finishers.

A runner has a start time and a finish time. Only runners that have started can have a finish time.

```
Runner
starter : NUMBER ⤖ DATETIME
finisher : NUMBER ⤖ DATETIME
─────────────────
dom finisher ⊆ dom starter
```

Initially, there are no runners.

```
InitRunner
Runner′
─────────────────
starter′ = ∅
finisher′ = ∅
```

When a runner crosses the starting mat on the start line their start time is set as the race start time. They are included in the set of those who have started, but only if their competitor number is for a genuine entry.

```
Start
ΔRunner
ΞEntryBase
time? : DATETIME
number? : NUMBER
─────────────────
number? ∈ dom entry
number? ∉ dom starter
starter′ = starter ∪ {number? ↦ time?}
finisher′ = finisher
```

When a runner crosses the timing mat on the finishing line their finish time is recorded. Only those who have started may have a finish time. We require that the finish time be after the start time.

The current (2010) world record for a half marathon is 58.23 held by Zersenay Tadese. So we require the difference between the start time and the finish time to be greater than, say, 40 minutes, that is 40 x 60 seconds or 2400 seconds. That is for a half marathon.

The current (2010) world record for a full marathon is 2:03:59 held by Haile Gebreselassie. So we require the start and finish times to be greater than, say, 100 minutes, that is 100 x 60 seconds, or 6000 seconds.

```
Finish
ΔRunner
ΞEntryBase
time? : DATETIME
number? : NUMBER
─────────────────
number? ∈ dom starter
(entry number?).category = halfMarathon
∧ raceTime (starter number?,time?) > 2400
∨ (entry number?).category = fullMarathon
  ∧ raceTime (starter number?,time?) > 6000
finisher′ = finisher ∪ {number? ↦ time?}
starter′ = starter
```

These two values, 2400 and 6000, should really be set as constants. But we shall leave them as they are for now.

Result

A result has a competitor number, a category, a name and a time. The number must be for a finisher. The category must be either half marathon or full marathon. The time is the race time and is derived from the runner's start and finish times.

```
┌─Result──────────────────────────────────────────────┐
│ ΞEntryBase                                          │
│ ΞRunner                                             │
│ number : NUMBER                                     │
│ category : CATEGORY                                 │
│ name : NAME                                         │
│ time : RACETIME                                     │
├─────────────────────                                │
│ number ∈ dom finisher                               │
│ category = (entry number).category                  │
│ category ∈ {halfMarathon,fullMarathon}              │
│ name = (entry number).name                          │
│ time = raceTime (starter number,finisher number)    │
└─────────────────────────────────────────────────────┘
```

Results are partitioned by category. A result is either for the half marathon or the full marathon. Further, the results are in finishing position order as determined by race time order. We require that there are no duplicates in the results: no runner may appear twice in the results and each runner has just one position.

```
┌─ResultByCategory────────────────────────────────────┐
│ ΞRunner                                             │
│ ΞEntryBase                                          │
│ ΞResult                                             │
│ result! : iseq Result                               │
│ category? : CATEGORY                                │
├─────────────────────                                │
│ category? ∈ {halfMarathon,fullMarathon}             │
│ ∀ i : dom result! • (result! i).category = category?│
│ ∀ i,j : dom result! | #result! > 1 ∧ i < j •        │
│   (result! i).time ≤ (result! j).time               │
└─────────────────────────────────────────────────────┘
```

Total Specification

We could include the error cases to make the specification complete. But we shall not bother since error conditions are implied by the constraints placed on the variables in the various schemas.