STS
Software
Technology
Systems

Software Engineering
Summer Semester 2015
Schupp/Kamph
18 May 2015

CAT – Compilation and Testing

Elevator Logic

Deadline: 15 June 2015, 12:14 CEST
Submission via ILIAS only!

1 CAT

CAT is a short term hands-on programming project that will be carried out in small groups. It focuses on two different skills necessary during software development: The first skill is the actual software creation process, i.e., programming a software component that fulfills a certain task under specific requirements. For this project the programming language of choice is C++. The second skill is the development of effective tests and their evaluation.

To make the project more interesting all participating groups will compete with their software solutions and tests: not only will your software be subject to evaluation by a test suite created by us but also all test suites submitted by your fellow students. Thus, your two main goals during CAT are:

- Creating a robust software solution that passes most tests
- Developing an effective test suite that finds most bugs

All files are to be handed in via Stud.IP using the CAT plugin. It provides means to upload your source code and tests and evaluates the uploaded files. Your results can be seen shortly after an upload.

2 Task

This semester's CAT task is the creation of a generic controller logic for different elevator systems. Thus, you are not only responsible for enforcing safety requirements imposed by the elevator configuration but also to ensure passengers reach their destination floors in a timely manner. Your logic should be completely encapsulated in a C++ class named `ElevatorLogic`. The related files, `ElevatorLogic.h` and `ElevatorLogic.cpp`, are the only source code files you need to submit.

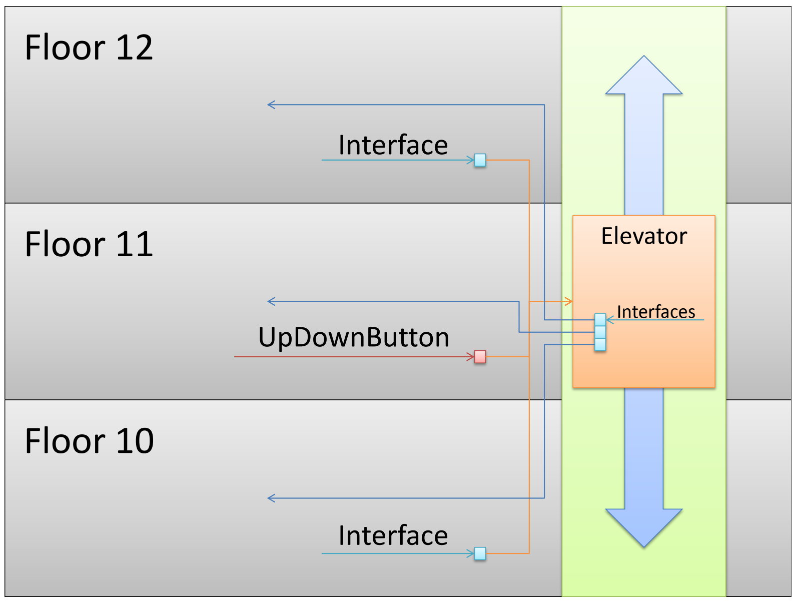

Figure 1: Example Elevator System

3 Elevator Systems

An elevator system consists of several objects, which interact by sending and receiving events. The system is timed and thus every event has a point in time when it is processed. The following components are available to define an elevator system:

Interface An `Interface` object represents an interface that a passenger may interact with.

UpDownButton An `UpDownButton` represents an interface that allows a passenger to indicate whether he or she desires to ascend or descend when calling an elevator.

Floor A `Floor` object represents a certain floor in a building. All `Floor` objects are ordered and can be imagined as a single tower of floors of a building. Every `Floor` object has a number of `Interface` and/or `UpDownButton` objects that are used to call elevators to the floor.

Elevator An `Elevator` object represents an elevator in the system. It has a number of `Interface` objects that can be interacted with to instruct the elevator to go to a particular floor.

As an example Figure 1 shows a system with 3 Floor objects, 1 Elevator and the objects to interact with the elevator.

More detailed information on the components, messages and functions can be found in Section 6.

4 Writing your ElevatorLogic

In this project one of your tasks is creating the `ElevatorLogic` class, which should react to user requests and operate the elevators accordingly. To achieve this task you need to react to messages and send messages on your own to communicate with the other system components. In the following the example the `ElevatorLogic.cpp` file you received with this PDF is explained in detail to get you started.

4.1 Reacting to Events

The first step to reacting to events is defining functions that should be called when a certain event type is received. An event handler function always has the following function signature:

```
void Class::Function(Environment &env, const Event &e)
```

where `Class` is the name of the receiving class, and `Function` is a freely chosen name for the event handler function. By convention message handlers are `private` class methods and their names indicate which event type is handled, e.g., `HandleNotify`.

In the example we react to four different message types: `Interface::Notify`, `Elevator::Stopped`, `Elevator::Opened`, and `Elevator::Closed`. Thus, four different event handlers functions are present:

- `void ElevatorLogic::HandleNotify(...)`
- `void ElevatorLogic::HandleStopped(...)`
- `void ElevatorLogic::HandleOpened(...)`
- `void ElevatorLogic::HandleClosed(...)`

The next step is registering the functions in the event system so that they are actually called when processing an event of the correct type. The registering should be done in the initialization function

```
void Class::Initialize(Environment &env),
```

which every component in the event system has. Source Code 1 shows the example initialization function, which registers the newly defined event handlers at the `Environment` object. The `RegisterEventHandler` function of the `Environment` object has three parameters: the event type to react to, the object, whose method should be called, and the method, which should be called.

After the registration the four event handler functions are ready and are called when a matching event occurs. Note that if at any time you need a handler that is called for every event, not taking the event type into account, you may register an event handler for the special event type `Environment::All`. For a summary of all the events in the elevator system please refer to the reference in Section 6.

```
void ElevatorLogic::Initialize(Environment &env) {
  env.RegisterEventHandler("Interface::Notify", this, &
      ElevatorLogic::HandleNotify);
  env.RegisterEventHandler("Elevator::Stopped", this, &
      ElevatorLogic::HandleStopped);
  env.RegisterEventHandler("Elevator::Opened", this, &
      ElevatorLogic::HandleOpened);
  env.RegisterEventHandler("Elevator::Closed", this, &
      ElevatorLogic::HandleClosed);
}
```

Source Code 1: Registering Event Handlers during initialization

4.2 Processing Events

Continuing the example we assume the following scenario that should be handled by the example logic:

a) A person requests an elevator. The elevator is already at the correct floor.

b) The elevator opens its doors, the person enters and instructs the elevator to ascend to the next floor.

c) The elevator doors close, the elevator ascends, stops at the floor above, and opens its doors. The person exits.

d) The elevator closes its doors.

Therefore, the first task the `ElevatorLogic` must fulfill is opening the doors when the person calls the elevator. Source Code 2 shows the source code that handles a notification from an `Interface` object. In line 3 we obtain the `Interface` object that sent the message. In line 4 and 5 we ensure that the `Interface` actually notifies an `Elevator` object. Note that we also receive an `Interface::Notify` event when a person instructs an elevator to go to a certain floor. Then, however, the targeted objects are floors. In line 6 we then obtain the `Elevator` object that should open its doors and send an `Elevator::Open` event referencing the `Elevator` object. The four parameters of the `SendEvent` method of the `Environment` object are the event type, the delay when the message should be processed, the object that issued the event, and a reference to a different object. The reference to another component may be omitted if none is necessary. Also an additional fifth string parameter can be specified to send arbitrary data. In this case, the method call, thus, lets us (`this`) send an `Elevator::Open` event that should be processed now (`0`) and that is referencing the `Elevator` object `ele`.

The remaining example source code has no interaction with the passenger anymore. We merely react to the elevator's status events:

- When the elevator doors have finished opening, we send an `Elevator::Close` command to close the doors 4 time units later.

```
void ElevatorLogic::HandleNotify(Environment &env, const Event &e)
{
  Interface *interf = static_cast<Interface*>(e.GetSender());
  Loadable *loadable = interf->GetLoadable(0);
  if (loadable->GetType() == "Elevator") {
    Elevator *ele = static_cast<Elevator*>(loadable);
    env.SendEvent("Elevator::Open", 0, this, ele);
  }
}
```

Source Code 2: Opening the doors when called

```
Floor { 10 11 0 4 1 40 }
Floor { 11 0 10 4 1 40 }
Interface { 40 1 20 }

Elevator { 20 1 10 11 2 30 31 }
Interface { 30 1 10 }
Interface { 31 1 11 }

Person { 50 11 10 10 5 0 }
```

Test Case 1: Example Scenario

- When the elevator doors have finished closing, we instruct the elevator to ascend for 4 time units by sending an `Elevator::Up` command and following it up with a `Elevator::Stop` command 4 time units later. Those messages are only sent if we have not yet moved the elevator to prevent an infinite loop.
- When the elevator stops, we instruct it to open its doors by sending an `Elevator::Open` command.

This small example logic is already capable of handling the previously defined scenario. Obviously, it still has many flaws, but you surely will be able to fix those and handle many more scenarios.

5 Writing a Test Case

A test case in the elevator scenario needs to define the elevator system to operate, the persons interacting with the elevator system, and, optionally, non-deterministic events, such as malfunctions of elevators.

A test case is a plain text file that specifies all of those components. As an example, Test Case 1 specifies the previously introduced scenario. At first the two floors are defined using a `Floor` specification:

```
Floor { <ID> <ID of floor below> <ID of floor above>
    <height> <number of interfaces>
    <list of interface IDs> }
```

Here `Floor 10` and `11` are defined where `Floor 10` is above `Floor 11`. Both floors have a height of 4 and a single interface that is used to call the elevator: `Interface 40`. This interface is then defined using a `Interface` specification:

```
Interface { <ID> <number of entities>
    <list of entity IDs> }
```

This interface only notifies a single object: `Elevator 40`. Note that an interface can not only contain references to elevators but also to floors. This is used later in the example test case to specify the elevator. The format for an elevator specification is

```
Elevator { <ID> <speed> <maximum load>
    <ID of starting floor> <number of interfaces>
    <list of interface IDs> }
```

In the test case an elevator with a speed of 1 distance units per time unit that can carry at maximum 10 weight units is specified. Initially, the elevator is located at `Floor 11` and the elevator has two interfaces, `Interface 30` and `Interface 31`, to interact with it. Those interfaces are specified next and each of them references the floor the elevator should go to. With these specifications the elevator system is completely defined.

Next, the person interacting with the system is defined using a person specification:

```
Person { <ID> <ID of starting floor>
    <ID of destination floor> <give up delay>
    <weight> <start time> }
```

The person in this scenario starts on `Floor 11` and wants to ascend to `Floor 10`. The person is willing to wait 10 time units for an elevator to arrive and weights 5 weight units. Furthermore, the person starts interacting with the elevator system at time 0, i.e., directly at the beginning of the simulation.

In this scenario no non-deterministic events such as elevator malfunctions are specified. Please refer to Section 6 for a complete reference on objects, events, and specifications.

6 Object Reference

In this section first all the objects part of the general event system are introduced. Then all objects part of the elevator system are presented in detail.

6.1 Environment

Description The `Environment` object controls the event system. It executes the event queue and allows sending and receiving events.

Super Class `Loadable`

Methods

`int SendEvent(const std::string &type, int delay, EventHandler *src, EventHandler *tgt, const std::string &data)`
: Sends an event of the given `type` after the given `delay` where `src` is the sending `EventHandler`. The parameters `tgt` and `data` are optional. They are used to pass an `EventHandler` object and/or arbitrary data along with the event. The return value is an event ID that may be used to cancel an event if it was not executed yet.

`bool CancelEvent(int id)`
: Cancels the event with the given `id` in case it has not yet been executed. Only useful if an event was sent with a delay.

`int GetClock() const`
: Returns the current logical time in the event system.

`void RegisterEventHandler(const std::string &type, EventHandler *obj, void (EventHandler::*fnc)(Environment&, const Event&))`
: Registers the function `fnc` of the object `obj` as a event handler that should be called whenever an event of the given `type` is processed.

6.2 Event

Description The `Event` object represents a single event in the event system.

Super Class `Loadable`

Methods

`const std::string &GetEvent() const`
: Returns the event type of the event.

`int GetTime() const`
: Returns the time this event should be processed.

`int GetId() const`
: Returns the ID of this event.

`EventHandler *GetSender() const`
: Returns a reference to the `EventHandler` object that sent this event.

`EventHandler *GetEventHandler() const`
: Returns a reference to the passed `EventHandler` object if it was specified.

`const std::string &GetData() const`
: Returns the arbitrary data string if any was specified.

Test Declaration

```
Event { <type> <time> <ID of sender>
    <ID of passed entity> (data string) }
```

`<type>` The type of this event. Refer to the other objects' references to see which events are permitted in a test declaration.

`<time>` The absolute time when this event should be processed.

`<ID of sender>` The ID of the entity that sends this event.

`<ID of passed entity>` The ID of a an entity to pass along with the event. Specify 0 for none.

`(data string)` An optional data string to pass along with the event.

6.3 EventHandler

Description The `EventHandler` class represents an object in the event system that may send and receive events.

Super Class None

Methods

`bool ForMe(const Event &e) const`
Checks whether the given event is for the `EventHandler` object, i.e., checks if `e.GetEventHandler()` is equal to `this`.

`bool FromMe(const Event &e) const`
Checks whether the given event is from the `EventHandler` object, i.e., checks if `e.GetSender()` is equal to `this`.

`const std::string &GetName() const`
Returns the name of the `EventHandler` object.

6.4 Loadable

Description The `Loadable` object represents an object that may be loaded from a test file in the event system.

Super Class None

Methods

`int GetId() const`
Returns the unique ID of this `Loadable` object.

`const std::string &GetType() const`
Returns the type of this `Loadable` object.

6.5 Entity

Description The `Entity` object represents an entity in the event system, i.e., a loadable object that takes part in the event system by sending and/or receiving events.

Super Class `Loadable`, `EventHandler`

6.6 Floor

Description The `Floor` object represents a single floor in the elevator system. It has a number of `Interface` objects that should allow a `Person` object to call one or more elevators to the floor.

Super Class `Loadable`

Methods

`bool HasElevator(Elevator *ele) const`
Checks whether the given elevator stops on this floor.

`bool IsAbove(Floor *floor) const`
Checks whether the given floor is above this floor.

`bool IsBelow(Floor *floor) const`
Checks whether the given floor is below this floor.

`int GetHeight() const`
Returns the height of this floor.

`int GetInterfaceCount() const`
Returns the number of `Interface` objects of this floor.

`Interface *GetInterface(int n) const`
Returns the `Interface` object with the given index.

Outgoing Events

None

Incoming Events

None

Test Declaration

```
Floor { <ID> <ID of floor below> <ID of floor above>
    <height> <number of interfaces>
    <list of interface IDs> }
```

`<ID>` A unique ID that refers to this floor.

`<ID of floor below>` The ID of the floor directly below this floor. Specify 0 if this is the bottom floor.

`<ID of floor above>` The ID of the floor directly above this floor. Specify 0 if this is the top floor.

`<height>` The height of this floor in distance units.

`<number of interfaces>` The number of `Interface` objects of this floor.

`<list of interface IDs>` The IDs of the `Interface` objects for this floor separated by spaces.

6.7 Elevator

Description The `Elevator` object represents an elevator in the elevator system. It moves along its elevator shaft when instructed and updates its status accordingly.

Super Class `Entity`

Invariants
These are requirements your elevator controller should try to meet.

- An elevator should only open its doors when it is not moving.
- An elevator should only open its doors when it is in the center of a floor.
- An elevator should not move when it is malfunctioning.
- An elevator should not move when its doors are open.
- An elevator should not move when it is overloaded.
- An elevator should not move beyond its top and bottom floors.
- An elevator should only beep when it is overloaded.
- An elevator should only beep when its doors are open.
- An elevator should not close its doors while it is beeping.

Methods

`bool HasFloor(Floor *floor) const`
Checks whether this elevator has an exit on the passed `floor`.

`bool IsLowestFloor(Floor *floor) const`
Checks whether the passed `floor` is the lowest floor this elevator can stop at.

`bool IsHighestFloor(Floor *floor) const`
Checks whether the passed `floor` is the highest floor this elevator can stop at.

`int GetSpeed() const`
: Returns the speed of this elevator.

`int GetMaxLoad() const`
: Returns the maximum load this elevator may carry.

`Floor *GetCurrentFloor() const`
: Returns the floor this elevator is currently at.

`double GetPosition() const`
: Returns the position of this elevator within the current floor. This is a value in the range [0,1] where 0 indicates the elevator is at the bottom of the current floor and 1 indicates the elevator is at the top of the floor, i.e., has traveled the complete height of the current floor.

`Movement GetState() const`
: Returns the current state of this elevator.
`Movement` is a `public enum` of `Elevator`:

  - **Idle** The elevator is currently not moving.
  - **Up** The elevator is currently ascending.
  - **Down** The elevator is currently descending.
  - **Malfunction** The elevator is malfunctioning.

`int GetInterfaceCount() const`
: Returns the number of interfaces this elevator has.

`Interface *GetInterface(int n) const`
: Returns the `Interface` object with the given index.

Outgoing Events

`GetSender()` of the `Event` object references the `Elevator` object.

`Elevator::Moving` Sent when the elevator starts to move.

`Elevator::Stopped` Sent when the elevator stops moving.

`Elevator::Opening` Sent when the elevator starts opening its doors.

`Elevator::Opened` Sent when the elevator finishes opening its doors.

`Elevator::Closing` Sent when the elevator starts closing its doors.

`Elevator::Closed` Sent when the elevator finishes closing its doors.

`Elevator::Beeping` Sent when the elevator starts beeping.

`Elevator::Beeped` Sent when the elevator stops beeping.

`Elevator::Malfunction` Sent when the elevator is malfunctioning. This message may be specified in an `Event` in a test declaration.

`Elevator::Fixed` Sent when the elevator is no longer malfunctioning. This message may be specified in an `Event` in a test declaration.

Incoming Events

`GetEventHandler()` of the `Event` object must reference the elevator to instruct.

`Elevator::Up` Instructs the elevator to begin ascending.

`Elevator::Down` Instructs the elevator to begin descending.

`Elevator::Stop` Instructs the elevator to stop moving.

`Elevator::Open` Instructs the elevator to open its doors.

`Elevator::Close` Instructs the elevator to close its doors.

`Elevator::Beep` Instructs the elevator to start beeping.

`Elevator::StopBeep` Instructs the elevator to stop beeping.

Test Declaration

```
Elevator { <ID> <speed> <maximum load>
    <ID of starting floor> <number of interfaces>
    <list of interface IDs> }
```

`<ID>` A unique ID that refers to this elevator.

`<speed>` The speed of this elevator in distance units per time unit.

`<maximum load>` The maximum load this elevator can support in weight units.

`<ID of starting floor>` The ID of the floor where this elevator is initially located.

`<number of interface>` The number of `Interface` objects this elevator has.

`<list of interface IDs>` The IDs of the `Interface` objects for this elevator separated by spaces.

6.8 Interface

Description The `Interface` object represents an interface in the elevator system. It can be interacted with and should notify all the `Loadable` objects referenced by it.

Super Class `Entity`

Methods

`int GetLoadableCount() const`
Returns the number of `Loadable` objects of this interface.

`Loadable *GetLoadable(int n) const`
Returns the `Loadable` object with the given index.

`bool HasLoadable(Loadable *e) const`
Checks whether the passed `Loadable` object is referenced by this interface.

Outgoing Events

`GetSender()` of the `Event` object references the `Interface` object.

`Interface::Notify` Sent when the interface was interacted with. `GetEventHandler()` of the `Event` object returns the entity that interacted with the interface.

Incoming Events

`GetEventHandler()` of the `Event` object must reference the interface to interact with.

`Interface::Interact` Interacts with the interface.

Test Declaration

```
Interface { <ID> <number of loadables>
    <list of loadable IDs> }
```

`<ID>` A unique ID that refers to this interface.

`<number of loadables>` The number of `Loadable` objects this interface references.

`<list of loadable IDs>` The IDs of the `Loadable` objects this interface references separated by spaces.

6.9 UpDownButton

Description The `UpDownButton` object is an `Interface` object that allows a person to express the direction the elevator should move after the person enters an elevator.

Super Class `Interface`

Methods

None

Outgoing Events
`GetSender()` of the `Event` object references the `Person` object.

`UpDownButton::Decide` Sent when an entity interacts with this `UpDownButton` object. `GetEventHandler()` of the `Event` object returns the event handler that interacted with the button.

`Interface::Notify` Sent when an entity decides on a direction. `GetEventHandler()` of the `Event` object returns the entity that interacted with the interface. `GetData()` of the `Event` object returns either `Up` or `Down` depending on the decision.

Incoming Events
`GetEventHandler()` of the `Event` object must reference the `UpDownButton` object to interact with.

`UpDownButton::Up` Instructs the `UpDownButton` to relay that the interacting entity wants to ascend.

`UpDownButton::Down` Instructs the `UpDownButton` to relay that the interacting entity wants to descend.

Test Declaration

```
UpDownButton { <ID> <number of loadables>
    <list of loadable IDs> }
```

`<ID>` A unique ID that refers to this `UpDownButton` object.

`<number of loadables>` The number of `Loadable` objects this button references.

`<list of loadable IDs>` The IDs of the `Loadable` objects this button references separated by spaces.

6.10 Person

Description The `Person` object represents a person in the elevator system. A person autonomously calls elevators to floors it currently is on in an attempt to get to its desired destination floor. It will enter and exit elevators it called and instruct the elevator to go to certain floors by interacting with the `Interface` objects. If an elevator starts beeping persons inside the elevator will exit an try to resolve the overloading problem.

Super Class `Entity`

Invariants
These are requirements your elevator controller should try to meet.

- A person should not have to wait longer than his or her maximum wait time for an elevator to arrive.

Methods

`Floor *GetCurrentFloor() const`
: Returns the floor the person is currently on. Not valid when the person is in an elevator.

`Floor *GetFinalFloor() const`
: Returns the desired destination floor of the person.

`int GetGiveUpTime() const`
: Returns the duration the person is willing to wait for an elevator.

`int GetWeight() const`
: Returns the weight of the person in weight units.

`Elevator *GetCurrentElevator() const`
: Returns the elevator this person is currently riding or `nullptr` if not in an elevator.

Outgoing Events

`GetSender()` of the `Event` object references the `Person` object.

`Person::Entering` Sent when the person starts entering an elevator. `GetEventHandler()` of the `Event` object returns the `Elevator` object.

`Person::Entered` Sent when the person finishes entering an elevator. `GetEventHandler()` of the `Event` object returns the `Elevator` object.

`Person::Exiting` Sent when the person starts exiting an elevator. `GetEventHandler()` of the `Event` object returns the `Elevator` object.

`Person::Exited` Sent when the person finishes exiting an elevator. `GetEventHandler()` of the `Event` object returns the `Elevator` object.

`Person::Canceled` Sent when the person aborts its current entering or exiting acting.

Incoming Events

None

Test Declaration

```
Person { <ID> <ID of starting floor>
    <ID of destination floor> <give up delay>
    <weight> <start time> }
```

`<ID>` A unique ID that refers to this person.

`<ID of starting floor>` The ID of the floor this person starts at.

`<ID of destination floor>` The ID of the desired destination floor.

`<give up delay>` The time the person is willing to wait for an elevator in time units.

`<weight>` The weight of the person in weight units.

`<start time>` The time when this person starts interacting with the elevator system.

7 Notes

7.1 Compilation

The C++ source code is written according to the `C++0x` standard. To successfully compile the source code, in GCC/G++ you need to pass two commandline arguments to the compiler:

- `-D__GXX_EXPERIMENTAL_CXX0X__`
- `-std=c++0x`

On other compilers the necessary settings are probably different. MS Visual C++ requires `/vmg`. Please see your compiler manual on whether or not `C++0x` is supported and how to enable it. A `Makefile` for GCC/G++ to compile the example source code and a MS Visual C++ project file is supplied.

7.2 Standard Library

The C++ standard library may come in handy during development of your `ElevatorLogic` class. Especially the `std::map`, `std::set`, and `std::vector` classes are useful to store status information. For example, you may need to track the current load of elevators. Then a `std::map<Elevator*, int>` allows you to create a lookup table for the load. See a reference of the standard library for more information, e.g., `http://www.cplusplus.com/reference/`.