Interactive Program Design in C++

A Taxonomy for Practitioners

Massimo Fioravanti

Politecnico di Milano

25 October 2025

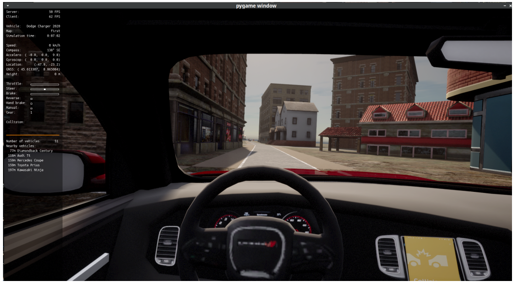
Who Am I?

- Compiler researcher at Politecnico di Milano

Research topic

How do you develop, test, refactor, maintain and reuse **hundreds** of **interactive** programs with as little overhead as possible?

Ex: Driving simulator developed with Vodafone Automotive

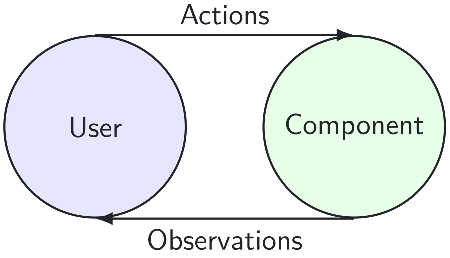


```
Server:                 50 FPS
Client:                 62 FPS

Vehicle:   Dodge Charger 2020
Map:                    first
Simulation time:      0:07:02

Speed:                 0 km/h
Compass:              138° SE
Accelero: ( -0.0,  0.0,  9.8)
Gyroscop: (  0.0,  0.0,  0.0)
Location:        (-47.9, -23.2)
GNSS: ( 45.613307,  8.865084)
Height:                   0 m

Throttle:
Steer:
Brake:
Reverse:
Hand brake:
Manual:
Gear:        1

Collision:

Number of vehicles:        51
Nearby vehicles:
  77m Diamondback Century
 116m Audi Tt
 159m Mercedes Coupe
 159m Toyota Prius
 197m Kawasaki Ninja
```

- We want to calibrate car **accident prediction** algorithms
- Corporate will **not** let us use real humans
- Simulations must be efficient because of machine learning.
- How do we maintain a **ever increasing** library of simulated scenarios?

Defining interactive components

A **interactive component** is a program component which behaviour depends on some input, and the input depends on the component behaviour.



Examples of interactive components

- A website with a multipage form
- A chess simulation to train AI agents
- The TCP protocol
- A load balancing algorithm that spawns and tears down servers depending on the state of the network

Often but not always

- Often **interactive components** are part of larger systems
- Often **interactive components** are a small part of a programmer's job

Techniques to design, implement and maintain interactive components are not commonly known

- Often **interactive components** are mandated by business requirements and/or third party specification documents
- Often **interactive components** start simple and then grow in complexity as features are added.

Changes in requirements sometimes push, without programmers noticing, the system in a entirely new category of complexity that forces to rewrite the whole component.

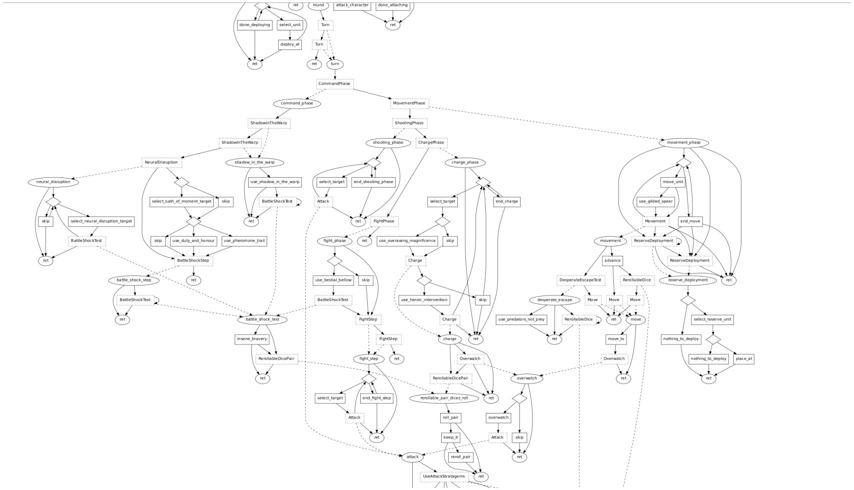
Running example

The user rolls to dices and sums them. If the result is less than 7 the user is allowed to reroll. Otherwise the user can add a number between 0 and 5 to the result.

```cpp
void runningExample() {
    int result = rollTwoDice();
    if (result < 7 && userWantsToReroll()) {
        result = rollTwoDice();
        return;
    }

    result += userDecidedQuantity();
}
```

userWantsToReroll and **userDecidedQuantity** are user actions.
rollTwoDice is a random event, independent from user actions.

Slides are only so large, pretend you have hundreds of user actions



Interactive components original sin: thread blocking

No main loop

Functions either block the current thread or they do not.

```cpp
void runningExample() {
    int result = rollTwoDice();
    if (result < 7 && userWantsToReroll()) { // waits for user input
        result = rollTwoDice();
        return;
    }

    result += userDecidedQuantity(); // waits for user input
}
```

Blocking is often unacceptable. Spawning a thread is sometimes too costly.

Examples:

```
void graphical_engine_main_loop(Engine& engine) {  // interactive
    while (not engine.is_done()){
        engine.render_and_display_frame(); // not interactive
        engine.query_inputs();             // not interactive
        engine.simulation_step();          // interactive
    }}
```

```
void machine_learning_chess_engine(NeuralNetwork& nn, Game& game) {
    while (not game.is_done()){
        Move move = nn.select_action(game.observe());  // not interactive
        game.apply_move(move);                         // interactive
    }}
```

The engine owns the main loop, the application logic cannot have it.

Class rewriting

```
void runningExample() {


  int result = rollTwoDice();
  if (result < 7 &&

        userWantsToReroll()) {


    result = rollTwoDice();
    return;
  }
  result += userDecidedQuantity();
}
```

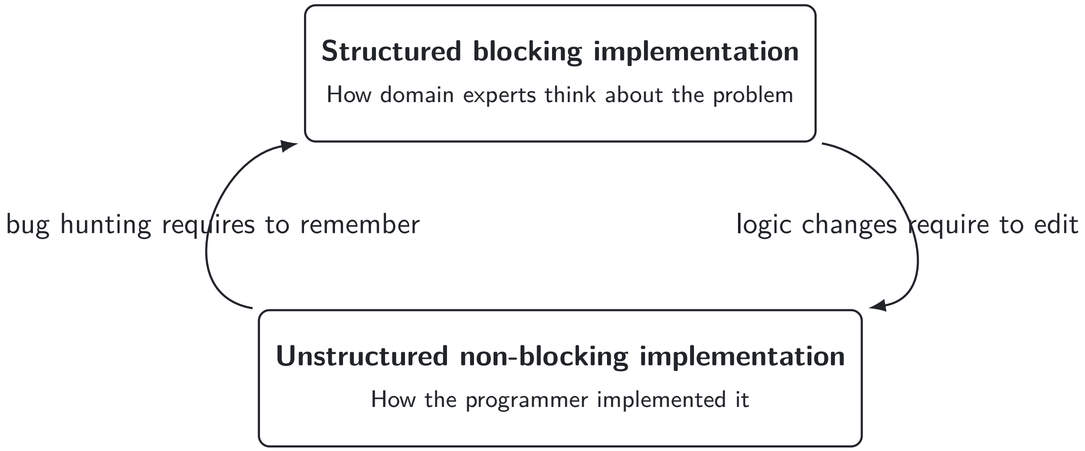
```
struct RunningExample {
 int next = 0; int result;
 void start() { assert(next == 0);
  result = rollTwoDice();
  next = result < 7 ? 1 : 2;
 }
 void userRerolls(bool it_does) {
  assert(next == 1);
  if (it_does)
    result = rollTwoDice();
  next = it_does ? -1 : 2;
 }
 void userDecidesQuantity(int q) {
  assert(next == 2);
  result += q;
  next = -1;
 }
```

Manual state managment $\equiv$ Unstructured control flow

Exploiting a new variable to keep track of the point we are at in the program is equivalent to unstructured programming.

Class implementation	**Unstructured C**	**Assembly**
`next`		`program counter`
`next = 2`	`goto label2`	`jmp label2`
`next = cond ? 1 : 2`	`switch (cond) {...}`	`cbr cond label1 label2`
`next = -1`	`return`	`ret`

Class rewrites are inherently complex to manage



Structured blocking implementation
How domain experts think about the problem
logic changes require to edit
Unstructured non-blocking implementation
How the programmer implemented it
bug hunting requires to remember

Class rewriting, general methods

Questions?

- Can any blocking function be rewritten as a class?
  yes
- Is there a general algorithm to convert a function into a class?
  yes: Control flow flattening in compilers, state machine syntesys in hardware design
- Can GCC, CLANG or MSVC do it for me?
  If coroutines are enough, yes

Coroutines digression

```
Task runningExample(Input<bool>&
    reroll, Input<int>& quantity) {
 int result = rollTwoDice();
 if (result < 7) {
  bool do_reroll = co_await reroll;
  if (do_reroll) {
   result = rollTwoDice();
   co_return result;
  }
 }
 result += co_await quantity;
 co_return result;}
```

```
int main() {
  Input<bool> reroll;
  Input<int>  quantity;
  Task t = runningExample(reroll,
       quantity);
  t.start();
  reroll.supply(false);
  quantity.supply(3);
  if (t.done()) {
    print("done")
  }
}
```

CPP coroutines are not copiable or serializable. Copiable/serializable strongly typechecked zero-overhead coroutines are very challenging to implement. We have a paper on this topic: https://arxiv.org/abs/2504.19625

Original sin, conclusion

- Some use cases require the main loop. (web servers, graphical engines...)
- The interactive component cannot have it too.

Solutions:

- Spawn a thread. **costly** **simple**
- Coroutines. **one malloc / free per coroutine creation, but manually optimizable** **complex at the start, easy afterward** **not copiable/serializable**
- Rewrite as a class. **1 extra integer cost in most situations** **Easy at the start, complex to maintain**

Acceptable implementations

	No calls	Non Recursive	Recursive	Turing complete
Non serializable	threads - coroutines			
Serializable				

Systems grow in complexity

- Domain experts often promise to never copy the state of interactive components, and then do it anyway. (Example: "In the car simulator, I want to copy the behaviour of a car, modify it, and after a while restore it to the previous behaviour")
- Often you want to isolate user actions in their own functions.

If your solution cannot handle one of the previous points, you will have to rewrite the system.

Turing complete?

Interactive systems where a user can provide an arbitrary program as input:

- The python interpreter
- A moddable videogame

Solution: interpreter / just in time compiler

	No calls	**Non Recursive**	**Recursive**	**Turing complete**
Non serializable	threads - coroutines			interpreter - jit
Serializable				interpreter - jit

Serializable interactive components that scales

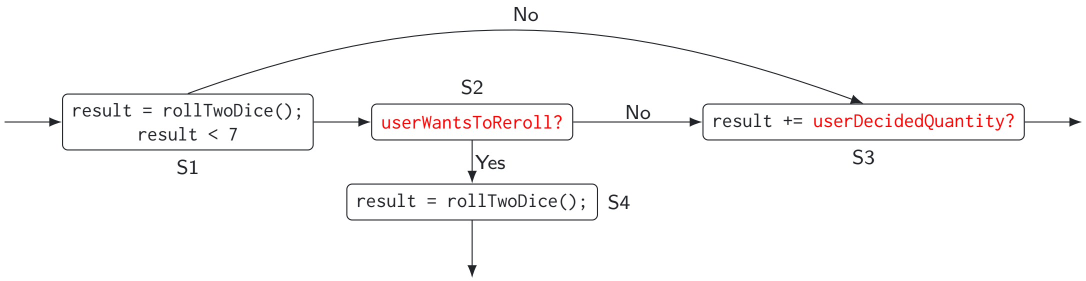
We need a way to implement those systems:

- Minimizing the distance between the mental model of the domain experts and of the implementation
- With as little overhead as possible
- With theoretical guarantees that expanding the implementation will not break the design.

State machines with extra tricks meet our requirements.

Serializable, no-calls interactive component

No calls interactive components are interactive components where all user actions appear in a single function.



Common, but often you want to isolate sections into subfunctions to reuse them, even when you could just write everything in a single one.

A possible implementation of a state machine library

```
STATE_MACHINE(resume) {

    STATE(S1)
    NEXT(S2)
    ....
    DECISION(S2):
    ...
}
```

```
void resume(Args args) {
    switch(state) {
labelS1: case S1: // S1 == 0
    goto labelS2;
    ....
labelS2: state=S2; return; case S2:
    ...
    }
}
```

labelS2: state=S2; allows us remember where we are.
return; stops the execution.
case S2: resumes the execution from the current line.

Conversion to CPP

```
class RunningExample {
  STATE_MACHINE(resume, {
    STATE(S1):
      result = rollTwoDice();
    NEXT(result < 7, S2, S4)
    DECISION(S2):
    NEXT(userWantsToReroll, S3, S4)
    STATE(S3):
      result = rollTwoDice();
    NEXT(END)
    DECISION(S4):
      result += userDecidedQuantity;
    NEXT(END)
    });
```

```
  enum State {
    S1,S2,S3,S4,END
  };
  int result; States state;
  void start() {
    assert(state == S1);
    resume();
  }
  ACT(S2, do_reroll,
      bool, userWantsToReroll)

  ACT(S4, decide_quantity,
      int, userDecidedQuantity)
};
```

```
void resume() {
 switch (state) {
  case S1:
   result = rollTwoDice();
   if (result < 7)
    goto labelS2;
   else
    goto labelS3
labelS2: state=S2; return; case S2:
   if (userWantsToReroll)
    goto labelS4;
   else
    goto labelS3;
labelS3:
   result = rollTwoDice();
   goto labelEND;
labelS4: state=S4; return; case S4:
   result += userDecidedQuantity;
   goto labelEND;
labelEND: state = END; case END:
        return;
    }}
```

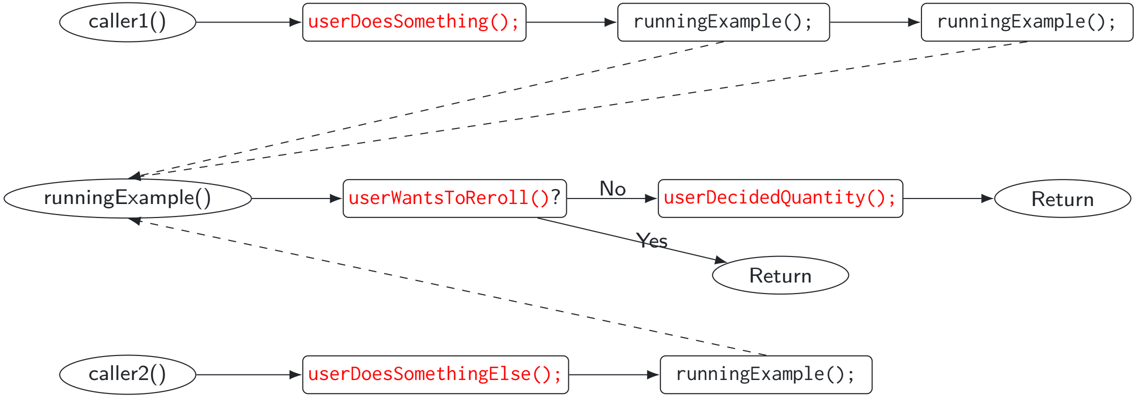
```
enum State {S1, S2, S3, END};
int result; States state;
void start() {
  assert(state == S1);
  resume();
}
bool userWantsToReroll;
void do_reroll(bool
     userWantsToReroll) {
  assert(state == S2);
  this->userWantsToReroll =
      userWantsToReroll;
  resume();
}
int userDecidedQuantity;
void decide_quantity(int
     userDecidedQuantity) {
  assert(state == S4);
  this->userDecidedQuantity =
      userDecidedQuantity;
  resume();
}
};
```

Acceptable implementations

	No calls	Non Recursive	Recursive	Turing complete
Non serializable	threads - coroutines - rewrites			interpreter
Serializable	STM			interpreter

Non-recursive non-blocking interactive functions

Actions may be located in multiple functions, but no functions is ever active more than once. Isolation of concerns makes this the most common scenario.



Solution, introduce CALL/RETURN macros

```
CALL(runningExample, C1)
```

```
ret_addresses.push_back(C1);
goto runningExample;
case C1:
```

```



RETURN()
```

```
while (true) {
  switch(state) {
    ...
    state = ret_addresses.back()
    ret_addresses.pop_back();
    continue;
    ...
  }
}
```

Extra memory footprint < 1 integer per function $+ 1$.

Acceptable implementations

	No calls	Non Recursive	Recursive	Turing complete
Non serializable	threads - coroutines			interpreter
Serializable	STM	STM+CALL/RET		interpreter

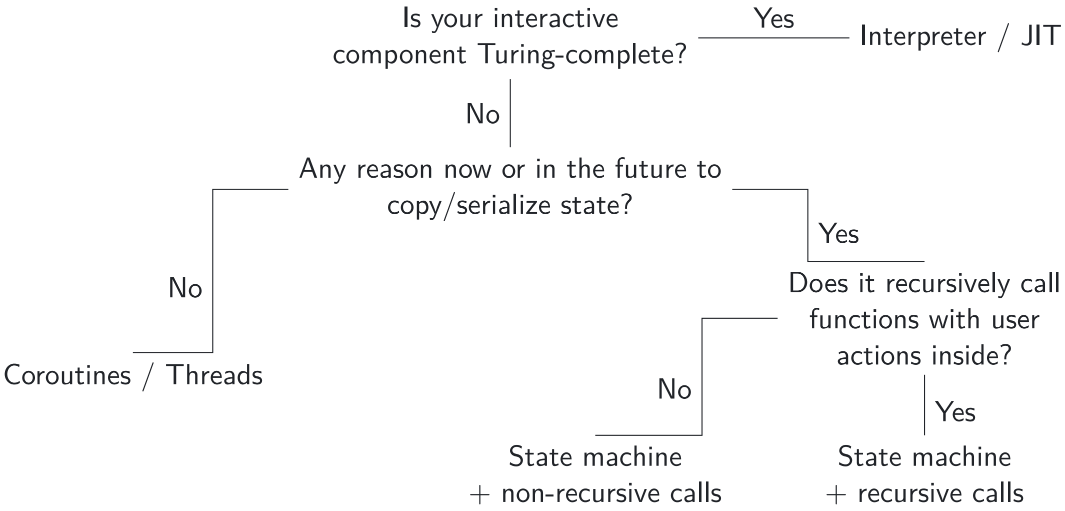
Recursive interactive systems

User actions are located in functions that can call directly or indirectly themselves. Rare, appears in videogames.

```
CALL(runningExample, C1)
```

```
stack_frames.emplace_back(
    RunningExample());
stack_frames.back().start();
state = C1;
return;
```

Requires dynamic memory (unless recursion is bounded), cost is proportional to the longest recursion chain.


Is your interactive component Turing-complete?
Yes
Interpreter / JIT
No
Any reason now or in the future to copy/serialize state?
No
Coroutines / Threads
Yes
Does it recursively call functions with user actions inside?
No
State machine + non-recursive calls
Yes
State machine + recursive calls

Conclusions

- Non blocking interactive systems force you into unstructured control flow
- The compiler knows what to do, but will not do it when you need serializable/copiable objects. All the complexities mentioned in this presentation would vanish if the compiler could do it.
- State machines of various complexity are the best tool you have to keep track of the complexity when it snowballs, while having guaranteed bounds on their cost, and guarantees on what is theoretically possible to implement.

Thanks!

Massimo Fioravanti massimo.fioravanti@polimi.it
Compleate Examples and slides: https://shorturl.at/9DkZw