Software Engineering 301:
Software Analysis and Design

Behavioural modelling:
State machine diagrams

Agenda

- What’s state?
- Why abstraction of states is needed
- Transitions between states
- Correcting non-determinism
- Showing effects
- Summary
- Examples

State

- Remember …
  - The state of a program involves
    - the value of every memory location on the machine
    - the value of every storage location on every peripheral device
    - …
- Fortunately, we can generally ignore most of that
  - The state that matters in practice involves
    - the values of every object and primitive value reachable
    - the value of every file accessed
  - **This is still a lot of information!**
  - The challenge:
    - How can we reduce it to a manageable and useful level?

A simple example

int i;

0

1

2

3

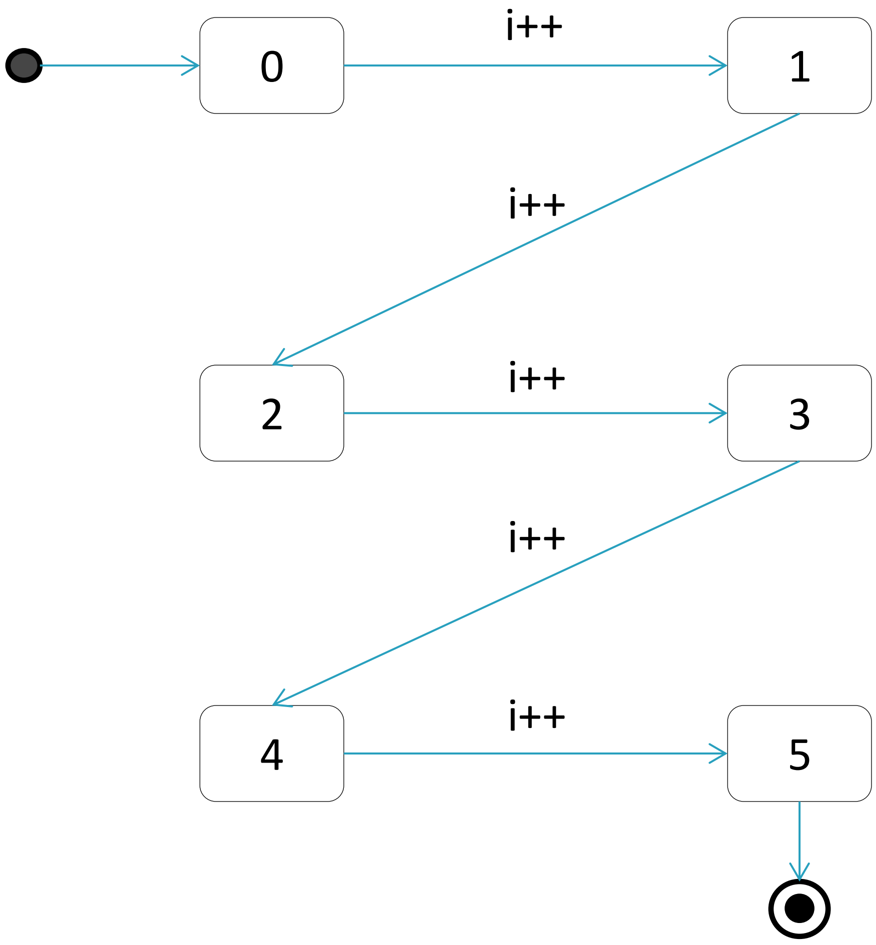
4

5

A simple example

for(int i = 0; i <= 5; i++);

this causes the state to change



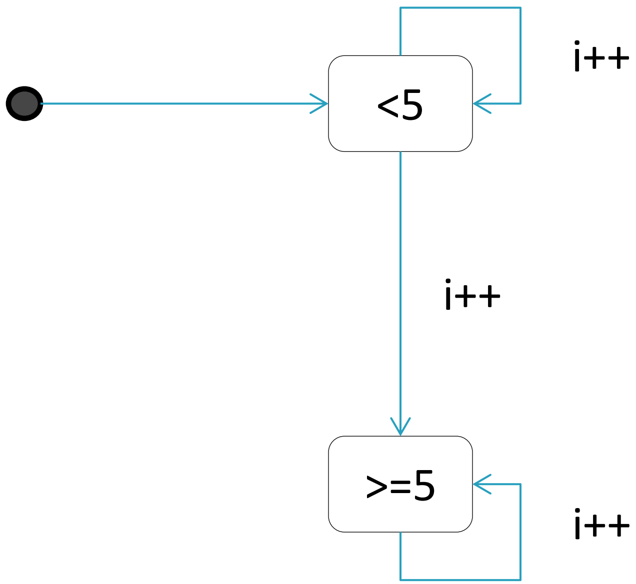
Not so interesting. Doesn't scale well if we change 5 to be a lot bigger.

Transitions, triggers, effects

- In modelling state, we are usually interested in capturing a few details:
  - What are the possible states?
  - What transitions between states are possible?
  - What causes a transition to occur?
  - Does something happen in response to a transition?

A simple example

```
for(int i = 0;; i++)
    if(i >= 5)
        System.out.print(i);
```



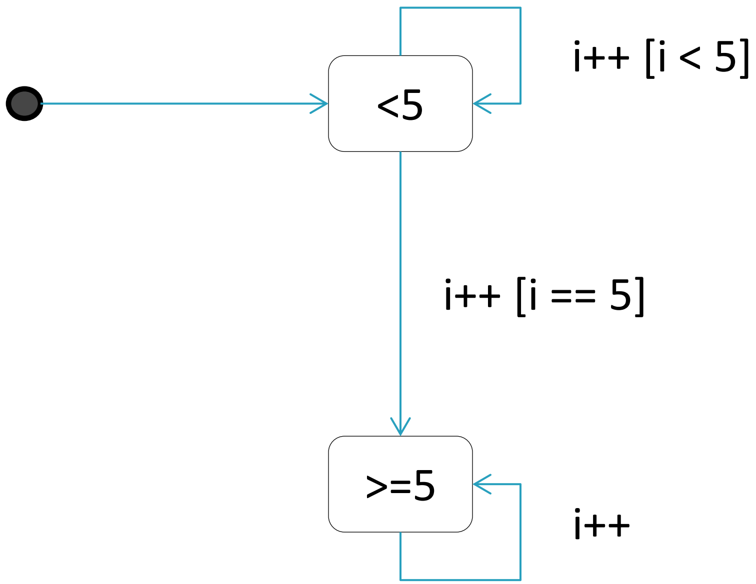
Problem: Event "i++" at state "<5" could cause either of two transitions.
We cannot determine which one will happen: this is ***non-determinism****.*
This state machine is ***non-deterministic****.*

Non-determinism

- We call such situations **non-deterministic**, because, while the semantics of the model say that the system can only be in one state, we cannot determine which of the alternatives is taken

A simple example

```
for(int i = 0;; i++)
   if(i >= 5)
      System.out.print(i);
```

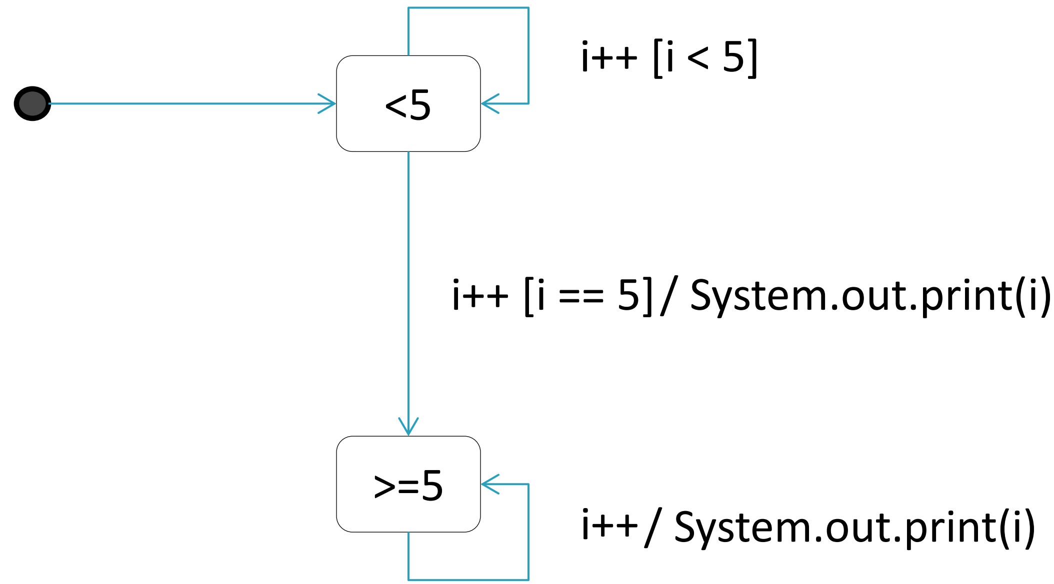


Solution: Use guards on the events. The guard expressions cannot overlap or else we still have non-determinism.

Problem: There is no difference in behaviour (in the model) due to state. This makes the model not very interesting/useful.

A simple example

```
for(int i = 0;; i++)
    if(i >= 5)
        System.out.print(i);
```

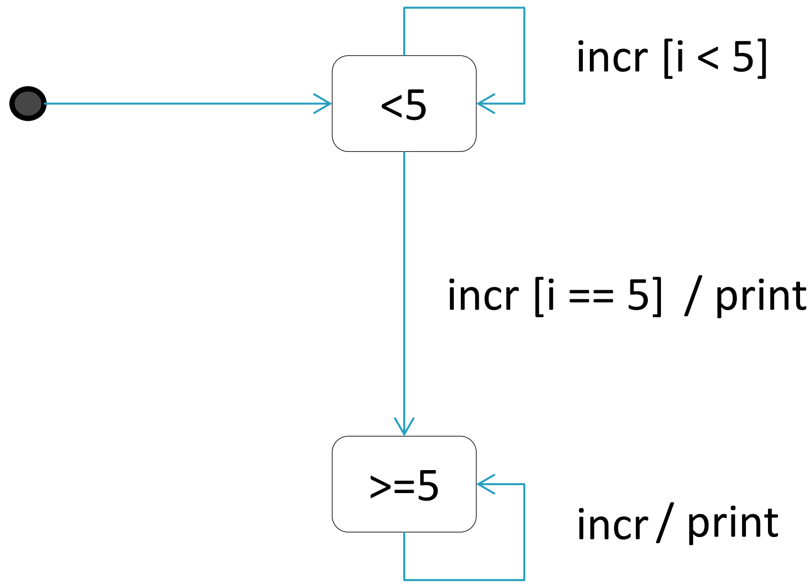


But there is difference in behaviour in the source code: “System.out.print(i)”

Solution: Add effects (reactions) to the transitions.

A simple example

```
for(int i = 0;; i++)
    if(i >= 5)
        System.out.print(i);
```



This is equivalent but focuses on the idea instead of the code. The fact that I can't think of a way to abstract away "i" suggests that it doesn't represent an abstract idea. That's a potential warning sign that the model isn't meaningful more abstractly.

Syntax

- States (rounded rectangles)
  - Have meaningful names
- Transitions
  - Unidirectional arrows between states
  - Labelled: *trigger* [*guard*] / *effect*
    - Any of these three things can be absent
    - e.g., an unlabelled transition is immediately followed
- Pseudo-states (start and end)

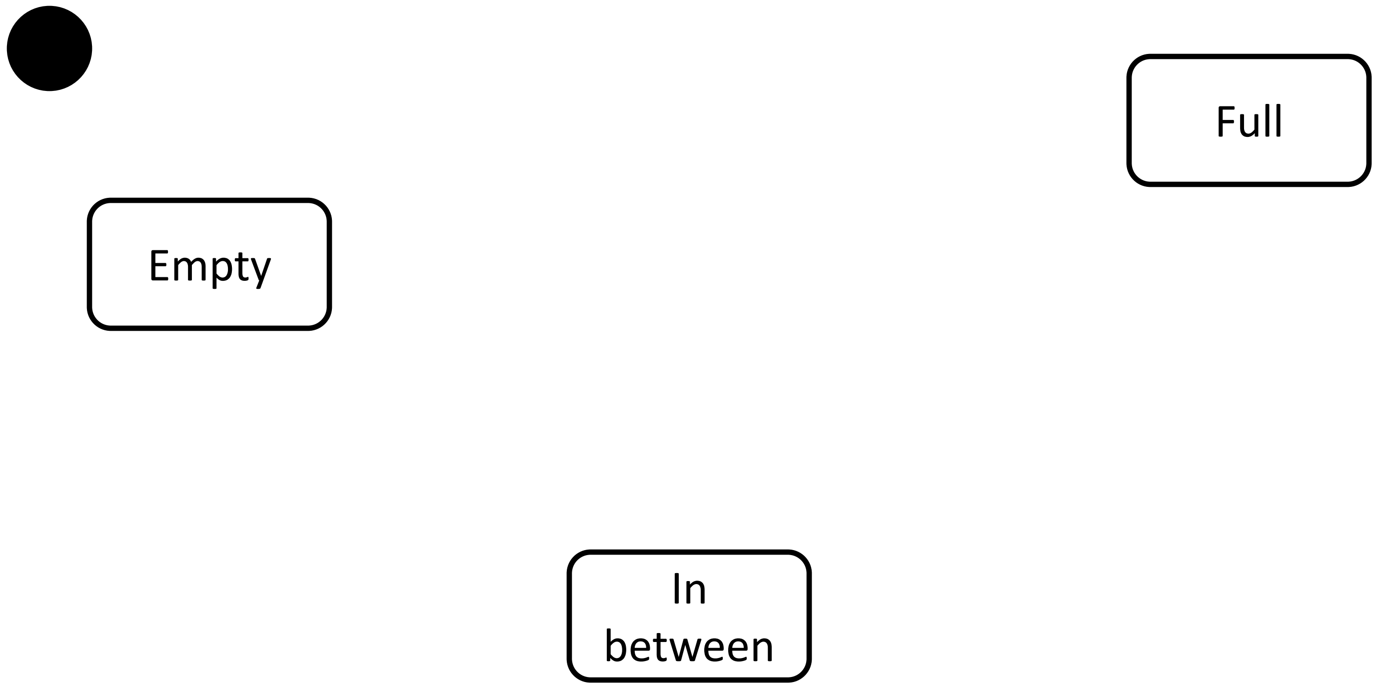
Semantics

- Imagine that you have a “token” that you place on the state where the machine currently is
- The token moves between states according to the possible paths (called transitions) and according to the events associated with each
- The token immediately moves from the start pseudo-state unless a guard is in place
- The machine stops if the end pseudo-state is reached

Stack

- Let’s model some simple behaviours of fixed-size stacks with states
- Fixed-size stacks have 3 basic states (at least, that’s how I choose to model them here): they are empty, they are full, or they are somewhere in between

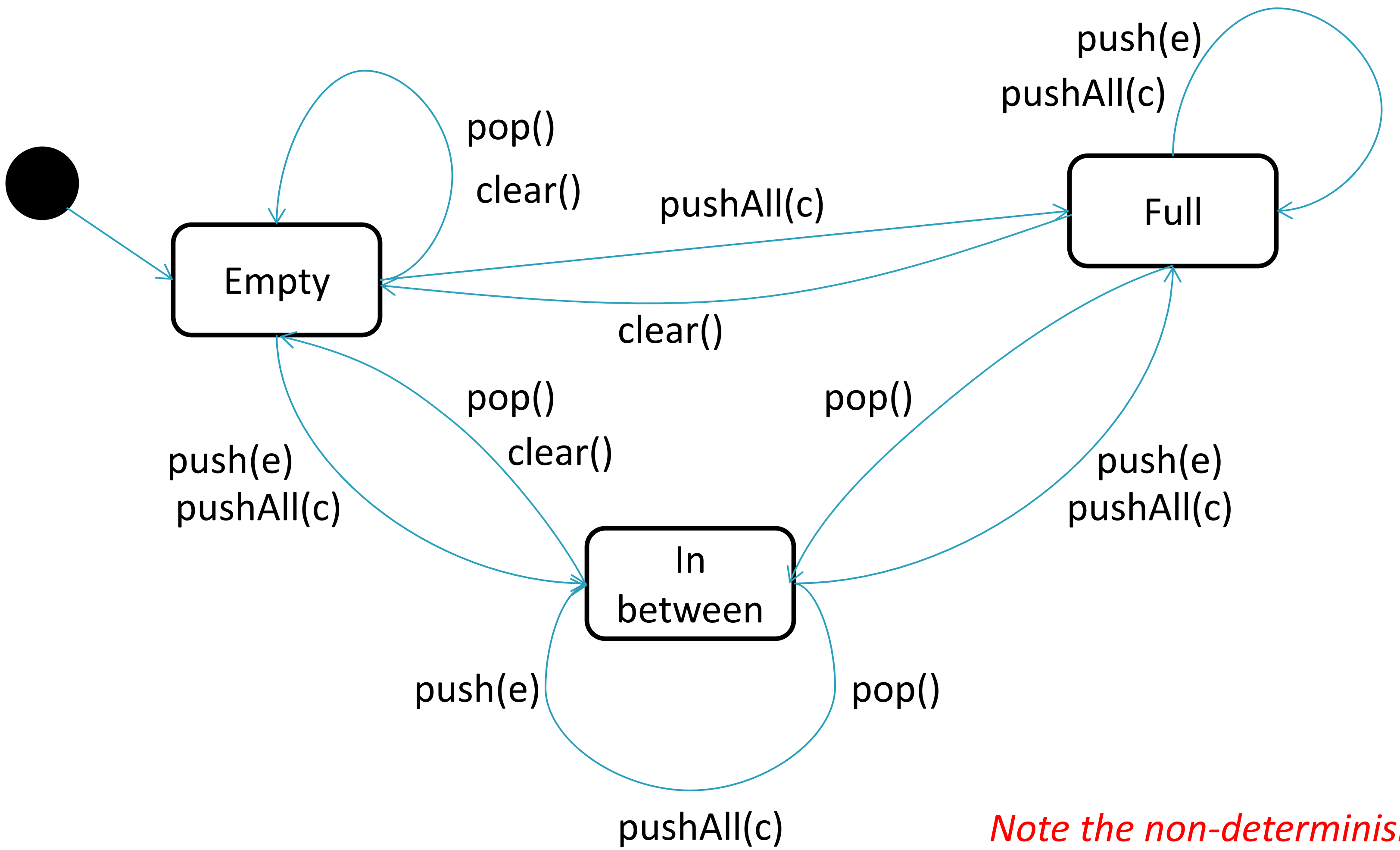
Chosen states



Events (or triggers)

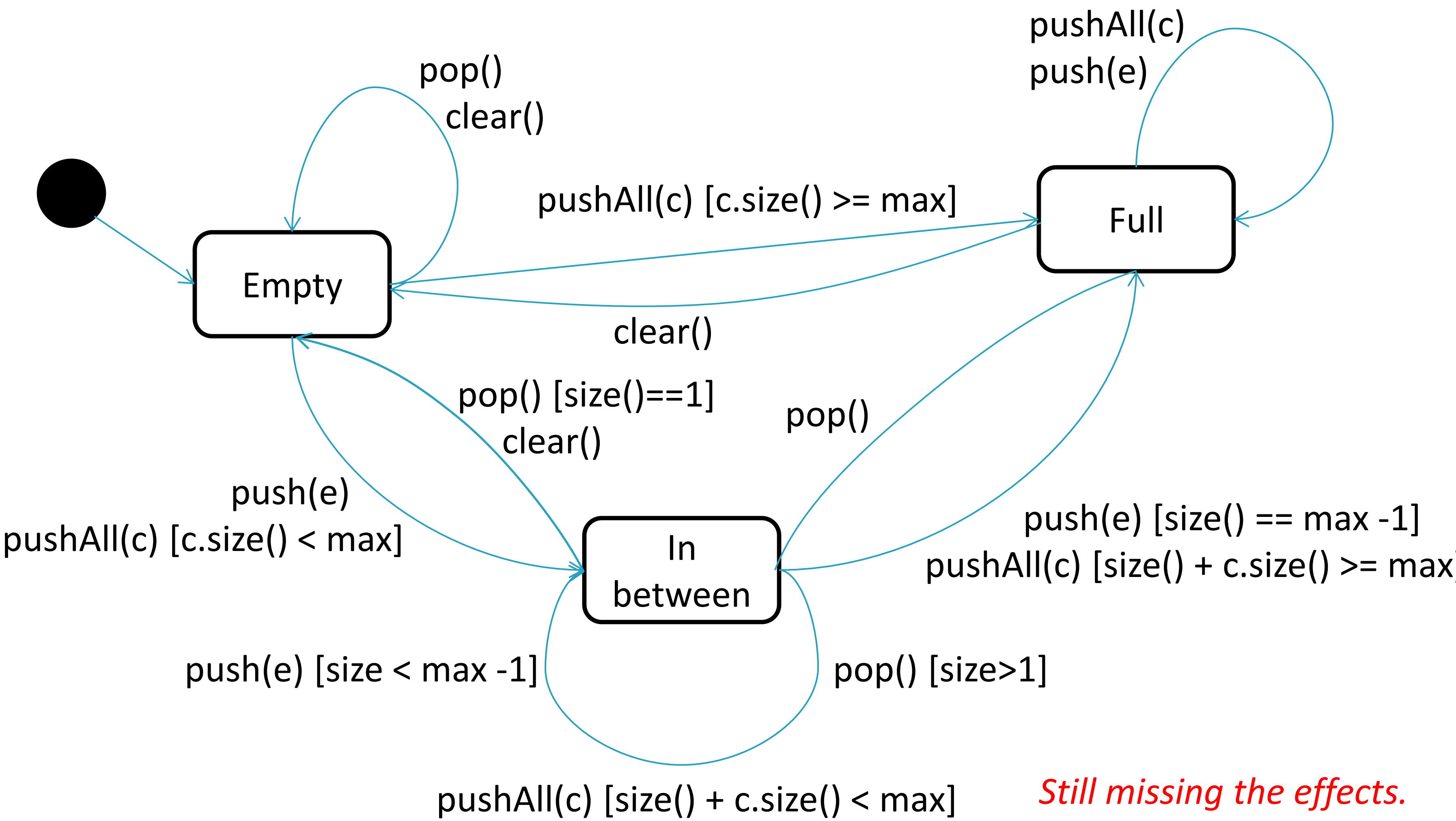
- These operations can affect the state and be affected by the state:
  - Add element: push(e)
  - Add multiple elements from another collection: pushAll(c)
  - Remove element: pop()
  - Clear out entire stack: clear()
- Other events don't alter the state, so we need not worry about them (but they may useful later…):
  - Look at top element: peek()
  - Check capacity: getCapacity()
  - Check current size: getSize()
  - Check if empty: isEmpty()
  - Check if full: isFull()

Transitions between states



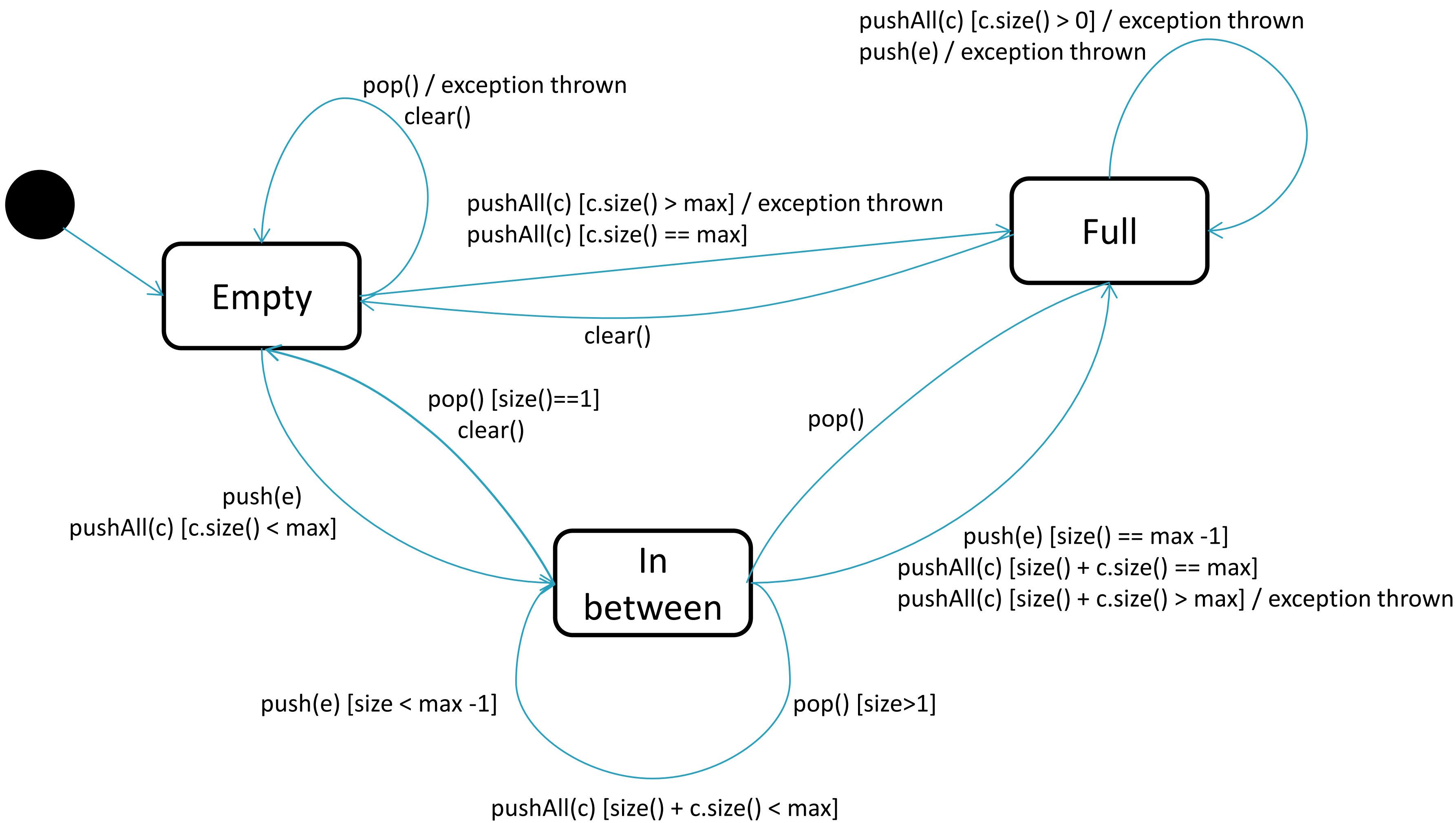
Note the non-determinism.

Guarded transitions



Still missing the effects.

Effects


pushAll(c) [c.size() > 0] / exception thrown
push(e) / exception thrown
pop() / exception thrown
clear()
pushAll(c) [c.size() > max] / exception thrown
pushAll(c) [c.size() == max]
Full
Empty
clear()
pop() [size()==1]
clear()
pop()
push(e)
pushAll(c) [c.size() < max]
In between
push(e) [size() == max -1]
pushAll(c) [size() + c.size() == max]
pushAll(c) [size() + c.size() > max] / exception thrown
push(e) [size < max -1]
pop() [size>1]
pushAll(c) [size() + c.size() < max]

Tracing for analysis

- We can figure out the sequence of events that can cause us to arrive at a particular state
- We can also figure out the sequence of events that can cause a particular reaction
- Or we can simply figure out what state we would be in at any given moment
- Assume max = 3:
  - pop(), push(e), push(e)
  - Are there any observable effects?
  - What state are we in?

Issues

- This is a very simple system, but we can already see that there is a problem with representing this much detail
- With more states, more transitions, etc., the complexity will quickly make such models not useful
- State machine diagrams have more advanced features for dealing with this (beyond scope of course)

Example: Polymorphism

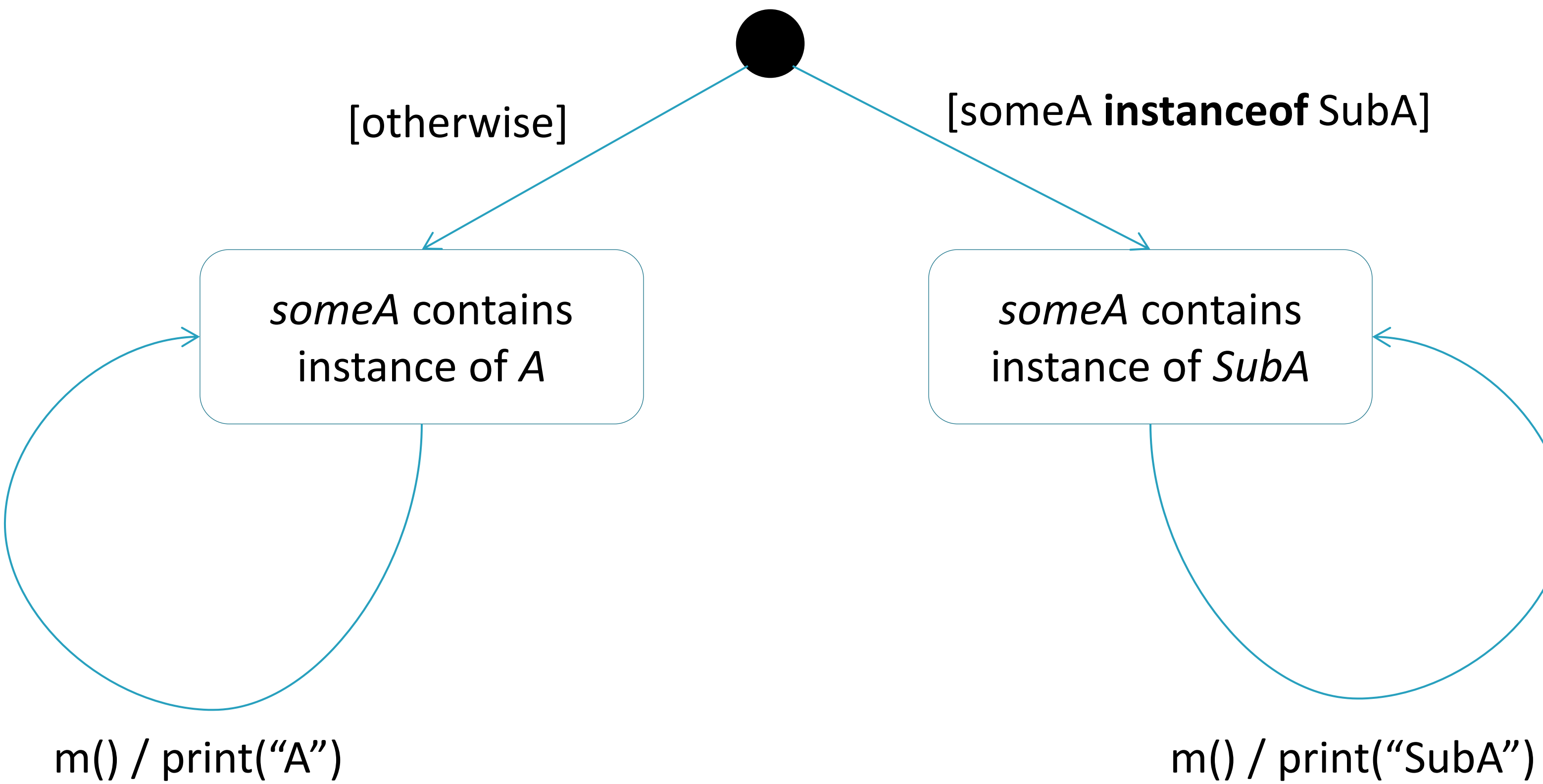
```
class A {
    void m() {
        System.out.print("A");
    }
}

class SubA extends A {
    void m() {
        System.out.print("SubA");
    }
}
```

```
class Client {
    void doit(A someA) {
        someA.m();
    }
}
```

Create a state machine diagram to represent the behaviour of doit

Example: Polymorphism



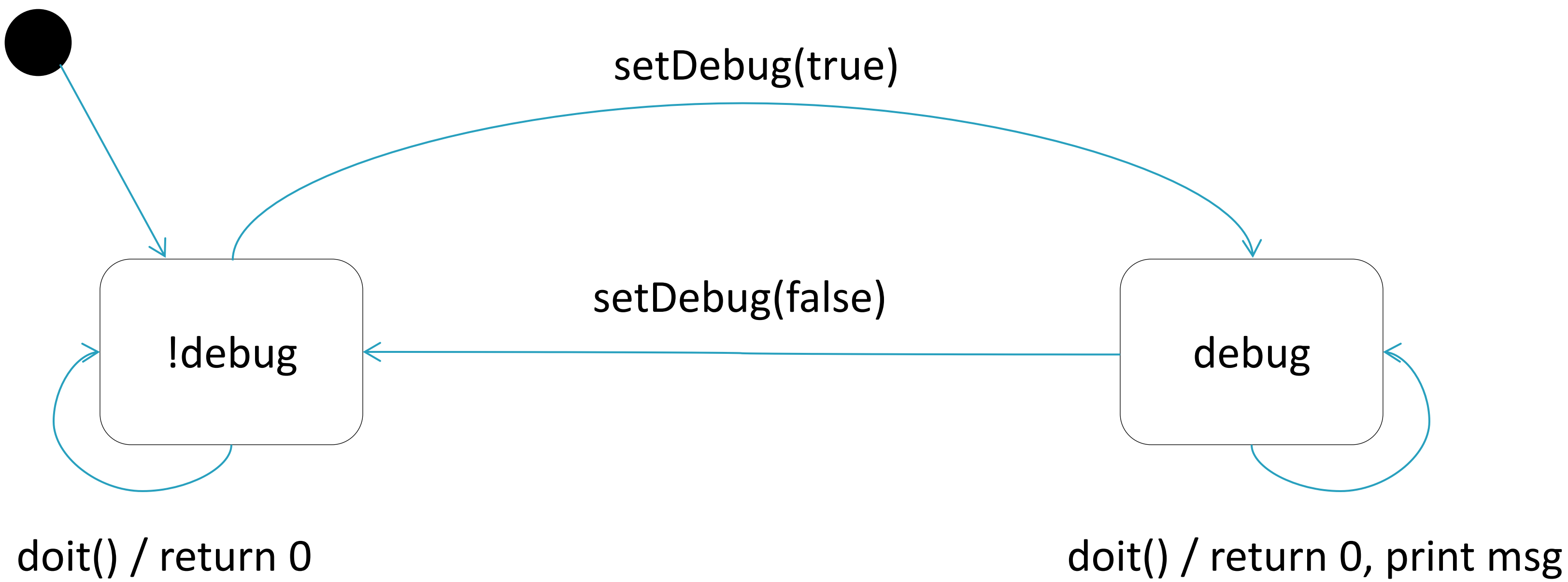
Example: A simple program

```
public class MyClass {
    private boolean debug = false;

    public void setDebug(boolean shouldDebug) {
        debug = shouldDebug;
    }

    public int doit() {
        if(debug)
            System.err.println("entered doit");
        return 0;
    }
}
```

Example: A simple program



I didn't show setDebug *on the self-transitions because nothing changes; they would simply clutter the diagram.*

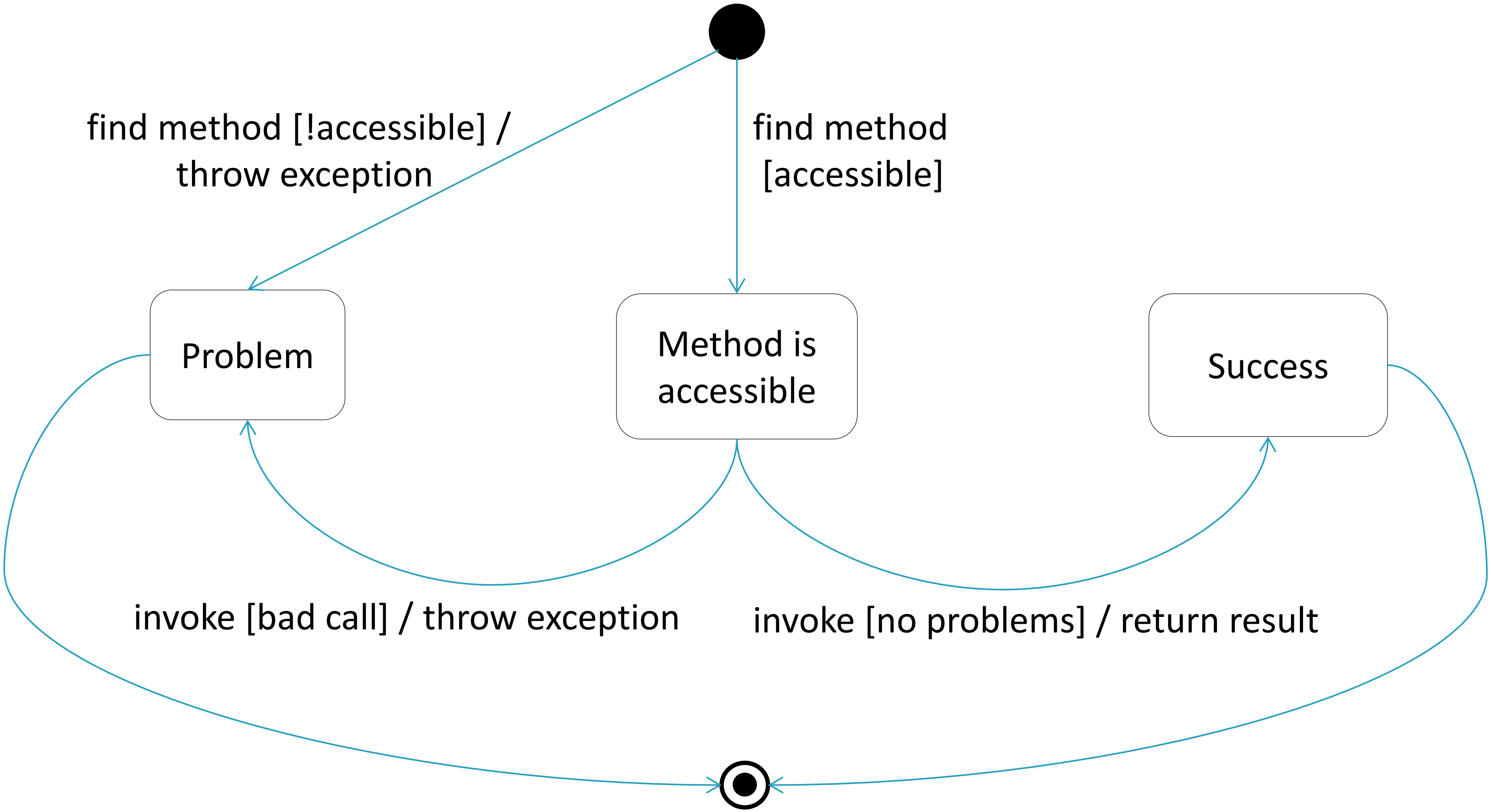
Example: Reflection

```
public class ReflectionExample {
  public Object doit(Class<?> aClass, String methodName, Class<?>[] params,
                         Object target, Object[] args)
    throws SecurityException, NoSuchMethodException,
           IllegalArgumentException, IllegalAccessException,
           InvocationTargetException {

    Method m = aClass.getMethod(methodName, params);

    return m.invoke(target, args);
  }
}
```

Example: Reflection



Next time

- Requirements