2IP15 Programming Methods

From Small to Large Programs

Tom Verhoeff
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

www.win.tue.nl/~wstomv/edu/2ip15

Overview

- Concurrency to decouple GUI from computation
- Threads, `SwingWorker`
- State design pattern
- Interface Segragation Principle (ISP)
- Perfomance profiling

Need for Concurrency in GUI

- Main Event Loop (Java: *Event Dispatch Thread*) drives the GUI.
- Event handlers must return quickly to guarantee responsiveness:

  “Slow” event handlers cause the GUI to “hang”.
- In Java, it is even impossible to force screen updates while event handlers runs.
- Solution: Run slow non-GUI code in a separate thread of control.

See: `download.oracle.com/javase/tutorial/uiswing/concurrency`

Need for Concurrency to Improve Performance

- Processors (in hardware) offer limited performance.

  Even performance improvement in next generations is now limited.

- To get more work done, use more processors concurrently.

  Need to distribute the computation, and coordinate the threads.

- Beyond the scope of this course.

Concurrent Execution

- Expression evaluation and method execution are **not atomic**.

  For example, `++ x` is executed as sequence of smaller operations:

```
1   reg_i = x;            // copy x to local register
2   reg_i = reg_i + 1;    // increment register
3   x = reg_i;            // copy register back to x
```

- Concurrent execution of *threads* **interleaves** *low-level* operations.

  Thread may be interrupted at any time for work of other threads.

Concurrent Execution: Example

Two threads each execute `++ x` on shared variable `x`, initially `0`.

The final value of `x` depends on the order of interleaving.

The final can be `2` (as expected), but also possible is:

Thread 1	Thread 2
`x == 0`	
`reg_1 = x`	
`reg_1 = reg_1 + 1`	
	`reg_2 = x`
`x = reg_1`	
	`reg_2 = reg_2 + 1`
	`x = reg_2`
`x == 1`	

Concurrent Execution: Puzzle

100 threads each execute `++ x` 100 times.

Shared variable `x` is initially `0`.

The largest (and maybe expected) final value is `10000`.

What is the smallest possible final value?

Concurrent Execution: Puzzle (continued)

100 is possible:

1. All threads start by reading `x`.
2. They all read the same value `0`.
3. They all write the same value `1` into `x`.
4. This interleaving is repeated 100 times.
5. This establishes `x == 100`.

Is a smaller final value possible?

Concurrent Execution: Puzzle Solution

1. All threads read `x`, putting `0` into their register.

2. Thread T_0 completes 99 increments: now `x == 99`.

3. Threads T_1 through T_{98} complete all their 100 increments.
   (For instance, one after the other.
   This establishes `x == 100`, since each starts from `0`.)

4. T_{99} completes its first increment (having read `0`): now `x == 1`.

5. T_0 reads `x`, retrieving value `1`.

6. T_{99} does its remaining 99 increments: now `x == 100`.

7. T_0 does its final increment from `1` to `2`: now `x == 2`.

Concurrency: Dangers

- Operation interleaving is non-deterministic (not predetermined).

  Hence, not reproducible.

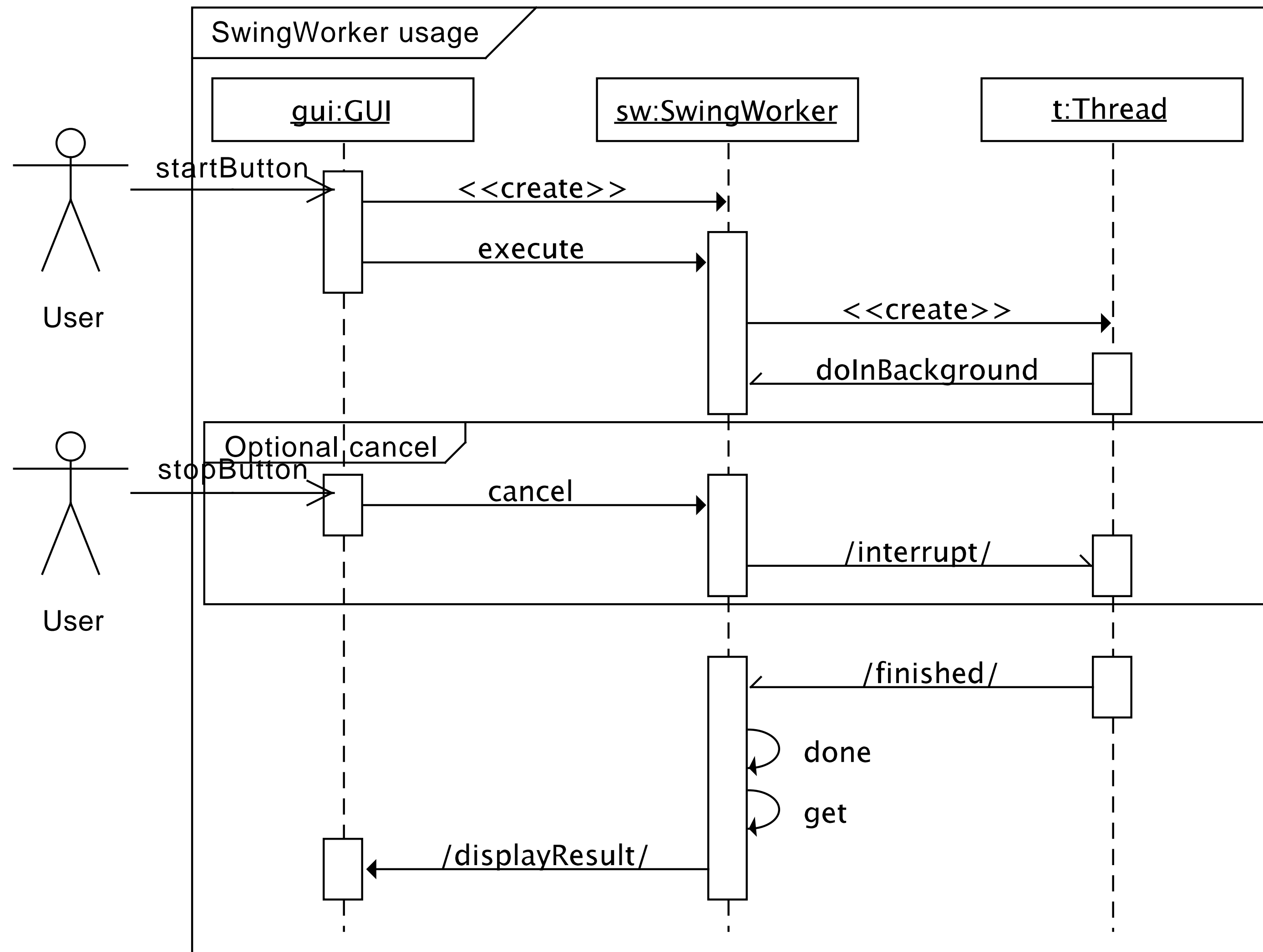
  This hinders reasoning, testing, and debugging.

- Shared data: How to guarantee invariants?
  - Result of two concurrent increments of variable `x`?
  - Concurrent update and printing of a time or date?

- *Thread interference, memory consistency errors, race conditions*

- First encounter: `SwingWorker`

SwingWorker Sequence Diagram



`SwingWorker<T,V>`: Start and End

- `T`: the result type returned by the SwingWorker's `doInBackground` and `get` methods; to ignore result, use `Void`
- Create new SwingWorker for each run of background task
- `@Override T doInBackground()`: implement background task here; do not call; is called automatically after `execute`
- **`void`** `execute()`: call this in GUI thread to start background task
- `@Override` **`void`** `done()`: implement finalization here; do not call; is called automatically in GUI thread when background task ends
- `T get()`: call this in `done` to get the result

See `SwingWorkerDemo.zip`

`SwingWorker<T,V>`: Communication to GUI

- `V`: the type used for delivering intermediate results by this Swing-Worker's `publish` and `process` methods; to ignore intermediate results use `Void`
- `publish(V ...)`: call in background task to deliver items to GUI
- `@Override process(List<V> chunks)`: implement processing of delivered items here; do not call; is called automatically in GUI

  Separately published items may be delivered in one process call.

See `SwingWorkerWithPublish.zip`

Also see `SwingWorkerWithProgressBar.zip`

State Pattern Motivation

- Multiple states/modes: with same events, different behaviors
- Compare to Finite State Machine

Action → / Next State		Event A	Event B	Event C
State	**S1**	`inc(A)` → S1	→ S3	→ S2
	S2	→ S1	`inc(B)` → S3	→ S2
	S3	→ S1	→ S3	`inc(C)` → S2

Concern: How to prevent clumsy conditional selection of behavior?

State Anti-Pattern

- Use a state variable and **`if`** or **`switch`** to vary behavior per event

See `StateAntiPattern.zip`

State Design Pattern (adapted from Eddie Burris)

Intent

- If an object goes through clearly identifiable states [modes],
- and the object's behavior is especially dependent on its state,
- it is a good candidate for the State design pattern.

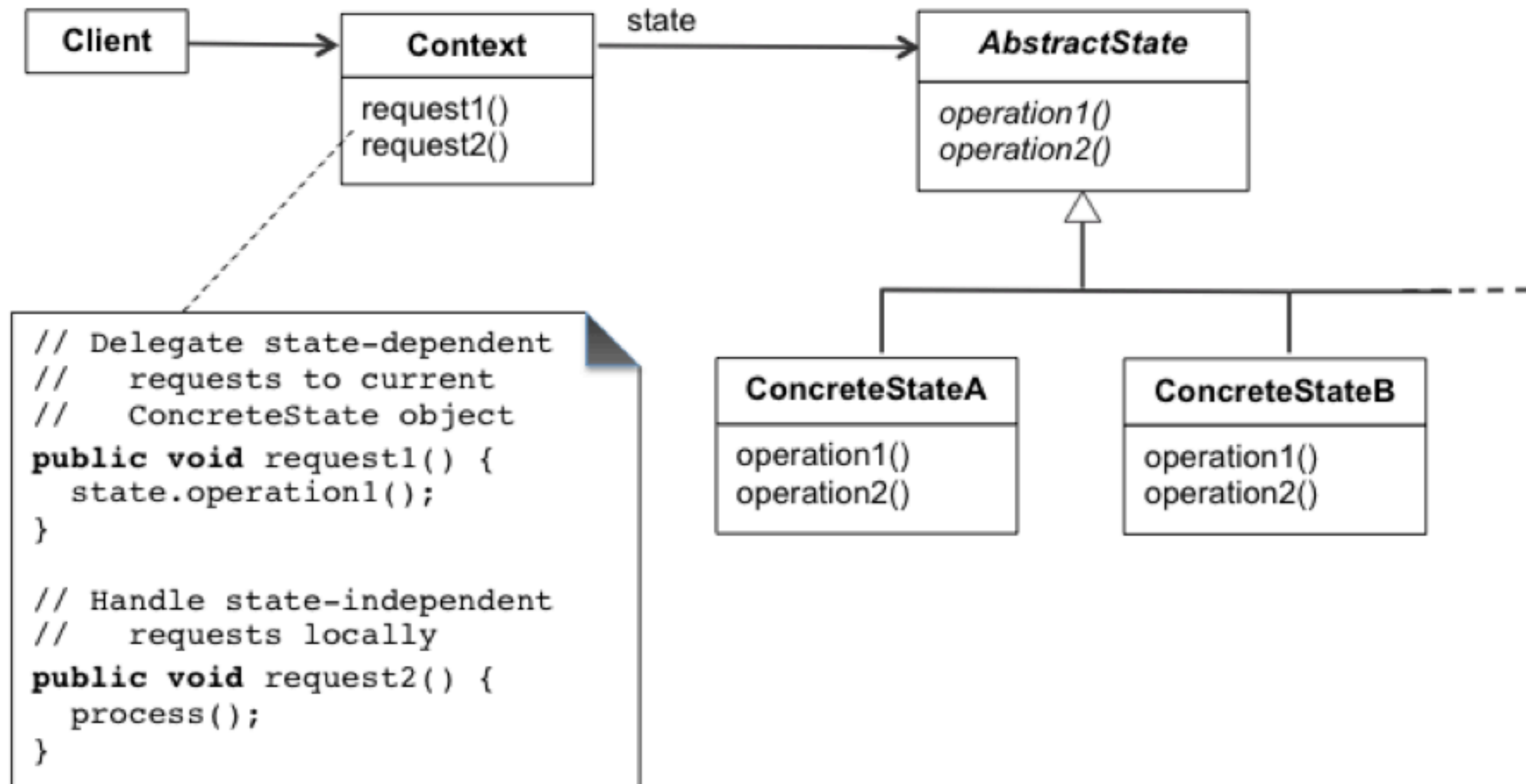
State Pattern Solution (adapted from Burris)

- Super class or interface specifies all events handled by a state
- The concrete event handlers are implemented in concrete states

  Different states can handle the same event in different ways
- *Context* code
  - holds a current state object
  - *delegates* event handling to the current state object
  - decides on change of state
- Alternatively: event handlers decide on change of state

See `StatePattern.zip`

State Pattern Class Diagram (from Burris)



SOLID Object-Oriented Design Principles

- **S**ingle Responsibility Principle (SRP, see Lecture 2)
- **O**pen Closed Principle (OCP, see Lecture 10)
- **L**iskov Substitution Principle (LSP, see Lecture 4)
- **I**nterface Segregation Principle (ISP, treated in this lecture)
- **D**ependency Inversion Principle (DIP, see Lecture 8)

Interface Segregation Principle: The Issue

```
class Service {
    void methodA1() { . . . }
    void methodA2() { . . . }
    void methodB1() { . . . }
    void methodB2() { . . . }
}

class ClientA { /* uses Service.methodAx */ }

class ClientB { /* uses Service.methodBx */ }
```

- When interface for `ClientB` changes, also `ClientA` 'suffers'

Interface Segregation Principle

(More advanced principle)

- When designing a class for several clients with different needs,
- rather than loading the class with all methods that clients need
- and making each client depend on the complete interface,
- create specific interfaces for each kind of client,
- make each client depend only on 'its' interface, and
- implement all interfaces in the class.

Interface Segregation Principle: Example

```
class InterfaceA {
    void methodA1();
    void methodA2();
}
class InterfaceB {
    void methodB1();
    void methodB2();
}

class Service implements InterfaceA, InterfaceB {
    . . . /* implement all methods */
}

class ClientA { /* uses InterfaceA.methodAx */ }
class ClientB { /* uses InterfaceB.methodBx */ }
```

Performance Profiling in NetBeans

- **Profile** > **Advanced Commands** > **Run Profiler Calibration**
- **Profile** > **Profile Main Project...**
  - CPU: Analyze Performance
  - **Advanced (instrumented)**, customize to exclude gui package
  - **Profile only project classes**
  - Do not **Use defined Profiling Points**
  - Run, Live Results
- Illustrated with a small demo
- Shows how often methods are called and how much time they run

Summary

- Concurrency via Java threads

  `SwingWorker`

- State design pattern
- Interface Segregation Principle (ISP)
- Performance profiling