Principles of Software Construction: Objects, Design, and Concurrency

Part 5: Concurrency

Introduction to concurrency, part 2

Concurrency primitives, continued

Charlie Garrod Michael Hilton

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Homework 5a due 9 a.m. tomorrow
  - Framework design discussion
- 2nd midterm exam returned today
- Reading due today:
  - Java Concurrency in Practice, Sections 11.3 and 11.4

A sudden reversal: one solution?

```
public class ReversedList<T> extends AbstractList<T> {
    private final List<T> source;

    public ReversedList<T>(List<T> source) {
        this.source = source;
    }
    @Override public int size() { return source.size(); }
    @Override public T get(int index) {
        return source.get(size() – index – 1);
    }
}

public class Mysterious {
    public static <T> List<T> reverseOf(List<T> source) {
        return new ReversedList<>(source);
    }
}
```

A sudden reversal: a slightly better solution

```
class ReversedList<T> extends AbstractList<T> {
    private final List<T> source;

    ReversedList<T>(List<T> source) {
        this.source = source;
    }
    @Override public int size() { return source.size(); }
    @Override public T get(int index) {
        return source.get(size() – index – 1);
    }
}

public class Mysterious {
    public static <T> List<T> reverseOf(List<T> source) {
        return new ReversedList<>(source);
    }
}
```

A sudden reversal: a slightly even better solution

```
public class Mysterious {
    private class ReversedList<T> extends AbstractList<T> {
        private final List<T> source;

        public ReversedList<T>(List<T> source) {
            this.source = source;
        }
        @Override public int size() { return source.size(); }
        @Override public T get(int index) {
            return source.get(size() – index – 1);
        }
    }

    public static <T> List<T> reverseOf(List<T> source) {
        return new ReversedList<>(source);
    }
}
```

A sudden reversal: a better solution

```
public class Mysterious {
    public static <T> List<T> reverseOf(List<T> source) {
        return new AbstractList<T>() {
            @Override public int size() {
                return source.size();
            }
            @Override public T get(int index) {
                return source.get(size() – index – 1);
            }
        };
    }
}
```

Plane and simple design challenge

Puzzler: “Racy Little Number”

Puzzler: “Racy Little Number”

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class LittleTest {
   int number;

   @Test
   public void test() throws InterruptedException {
       number = 0;
       Thread t = new Thread(() -> {
          assertEquals(2, number);
       });
       number = 1;
       t.start();
       number++;
       t.join();
   }
}
```

How often does this test pass?

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class LittleTest {
   int number;

   @Test
   public void test() throws InterruptedException {
       number = 0;
       Thread t = new Thread(() -> {
          assertEquals(2, number);
       });
       number = 1;
       t.start();
       number++;
       t.join();
   }
}
```

(a) It always fails

(b) It sometimes passes

(c) It always passes

(d) It always hangs

How often does this test pass?

(a) It always fails

(b) It sometimes passes

(c) It always passes – but it tells us nothing

(d) It always hangs

JUnit doesn't see assertion failures in other threads

Another look

```
import org.junit.*;
import static org.junit.Assert.*;

public class LittleTest {
   int number;

   @Test
   public void test() throws InterruptedException {
       number = 0;
       Thread t = new Thread(() -> {
          assertEquals(2, number); // JUnit never sees the exception!
       });
       number = 1;
       t.start();
       number++;
       t.join();
   }
}
```

How do you fix it? (1)

```
// Keep track of assertion failures during test
volatile Exception exception;
volatile Error error;

// Triggers test case failure if any thread asserts failed
@After
public void tearDown() throws Exception {
    if (error != null)
        throw error;
    if (exception != null)
        throw exception;
}
```

How do you fix it? (2)

```
Thread t = new Thread(() -> {
    try {
        assertEquals(2, number);
    } catch(Error e) {
        error = e;
    } catch(Exception e) {
        exception = e;
    }
});
```

Now it sometimes passes*

*YMMV (It's a race condition)

The moral

- JUnit does not well-support concurrent tests
  - You might get a false sense of security
- Concurrent clients beware…

Puzzler: “Ping Pong”

```
public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread( () -> pong() );
        t.run();
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}
```

What does it print?

```
public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread( () -> pong() );
        t.run();
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}
```

(a) PingPong

(b) PongPing

(c) It varies

What does it print?

(a) `PingPong`

(b) `PongPing`

(c) It varies

Not a multithreaded program!

Another look

```
public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread( () -> pong() );
        t.run();  // An easy typo!
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}
```

How do you fix it?

```
public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread( () -> pong() );
        t.start();
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}
```

Now prints PingPong

The moral

- Invoke `Thread.start`, not `Thread.run`
- `java.lang.Thread` should not have implemented `Runnable`

Key concepts from last Thursday

An easy fix:

```
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }
    static synchronized void transferFrom(BankAccount source,
                                          BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance   += amount;
    }
    public synchronized long balance() {
        return balance;
    }
}
```

Concurrency control with Java's *intrinsic* locks

- `synchronized (lock) { … }`
  - Synchronizes entire block on object `lock`; cannot forget to unlock
  - Intrinsic locks are *exclusive*: One thread at a time holds the lock
  - Intrinsic locks are *reentrant*: A thread can repeatedly get same lock
- `synchronized` on an instance method
  - Equivalent to `synchronized (this) { … }` for entire method
- `synchronized` on a static method in class Foo
  - Equivalent to `synchronized (Foo.class) { … }` for entire method

Some simple actions are not atomic

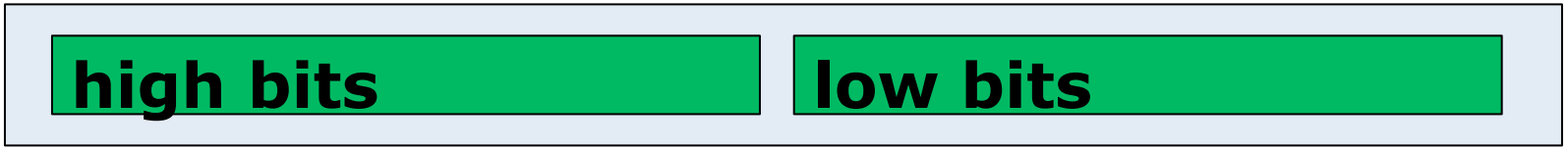
- Increment is not atomic

```
i++;
```

is actually

1. Load data from variable `i`
2. Increment data by `1`
3. Store data to variable `i`

- Also, Java Language Spec allows word-tearing on 64 bit values



Yet another example: cooperative thread termination

```
public class StopThread {
    private static boolean stopRequested;

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested)
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(42);
        stopRequested = true;
    }
}
```

How do you fix it?

```
public class StopThread {
    private static boolean stopRequested;
    private static synchronized void requestStop() {
        stopRequested = true;
    }
    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested())
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(42);
        requestStop();
    }
}
```

A better(?) solution

```
public class StopThread {
    private static volatile boolean stopRequested;

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested)
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(42);
        stopRequested = true;
    }
}
```

Today: Concurrency, motivation and primitives

- Midterm exam 2 recap
- Racy puzzlers
- More basic concurrency in Java
  - Some challenges of concurrency
  - Concurrency primitives for coordination
- Still coming soon:
  - Higher-level abstractions for concurrency
  - Program structure for concurrency
  - Frameworks for concurrent computation

A liveness problem: poor performance

```
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }
    static synchronized void transferFrom(BankAccount source,
                                          BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance   += amount;
    }
    public synchronized long balance() {
        return balance;
    }
}
```

A liveness problem: poor performance

```
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }
    static void transferFrom(BankAccount source,
                             BankAccount dest, long amount) {
        synchronized(BankAccount.class) {
            source.balance -= amount;
            dest.balance   += amount;
        }
    }
    public synchronized long balance() {
        return balance;
    }
}
```

A proposed fix?: *lock splitting*

```
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }
    static void transferFrom(BankAccount source,
                             BankAccount dest, long amount) {
        synchronized(source) {
            synchronized(dest) {
                source.balance -= amount;
                dest.balance   += amount;
            }
        }
    }
    …
}
```

A liveness problem: deadlock

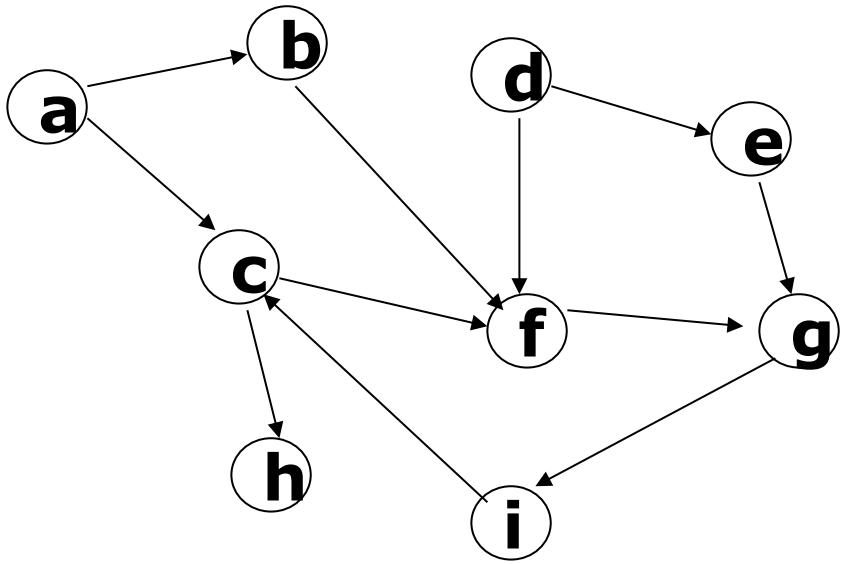
- A possible interleaving of operations:
  - `bugsThread` locks the `daffy` account
  - `daffyThread` locks the `bugs` account
  - `bugsThread` waits to lock the `bugs` account…
  - `daffyThread` waits to lock the `daffy` account…

A liveness problem: deadlock

```
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }
    static void transferFrom(BankAccount source,
                             BankAccount dest, long amount) {
        synchronized(source) {
            synchronized(dest) {
                source.balance -= amount;
                dest.balance   += amount;
            }
        }
    }
    …
}
```

Avoiding deadlock

- The *waits-for graph* represents dependencies between threads
  - Each node in the graph represents a thread
  - An edge T1->T2 represents that thread T1 is waiting for a lock T2 owns
- Deadlock has occurred iff the waits-for graph contains a cycle
- One way to avoid deadlock: locking protocols that avoid cycles



Avoiding deadlock by ordering lock acquisition

```
public class BankAccount {
  private long balance;
  private final long id = SerialNumber.generateSerialNumber();

  public BankAccount(long balance) {
    this.balance = balance;
  }

  static void transferFrom(BankAccount source,
                           BankAccount dest, long amount) {
    BankAccount first = source.id < dest.id ? source : dest;
    BankAccount second = first == source ? dest : source;
    synchronized (first) {
        synchronized (second) {
            source.balance -= amount;
            dest.balance += amount;
        }
    }
  } …
```

A subtle problem: The lock object is exposed

```
public class BankAccount {
  private long balance;
  private final long id = SerialNumber.generateSerialNumber();

  public BankAccount(long balance) {
    this.balance = balance;
  }

  static void transferFrom(BankAccount source,
                           BankAccount dest, long amount) {
    BankAccount first = source.id < dest.id ? source : dest;
    BankAccount second = first == source ? dest : source;
    synchronized (first) {
        synchronized (second) {
            source.balance -= amount;
            dest.balance += amount;
        }
    }
  } …
```

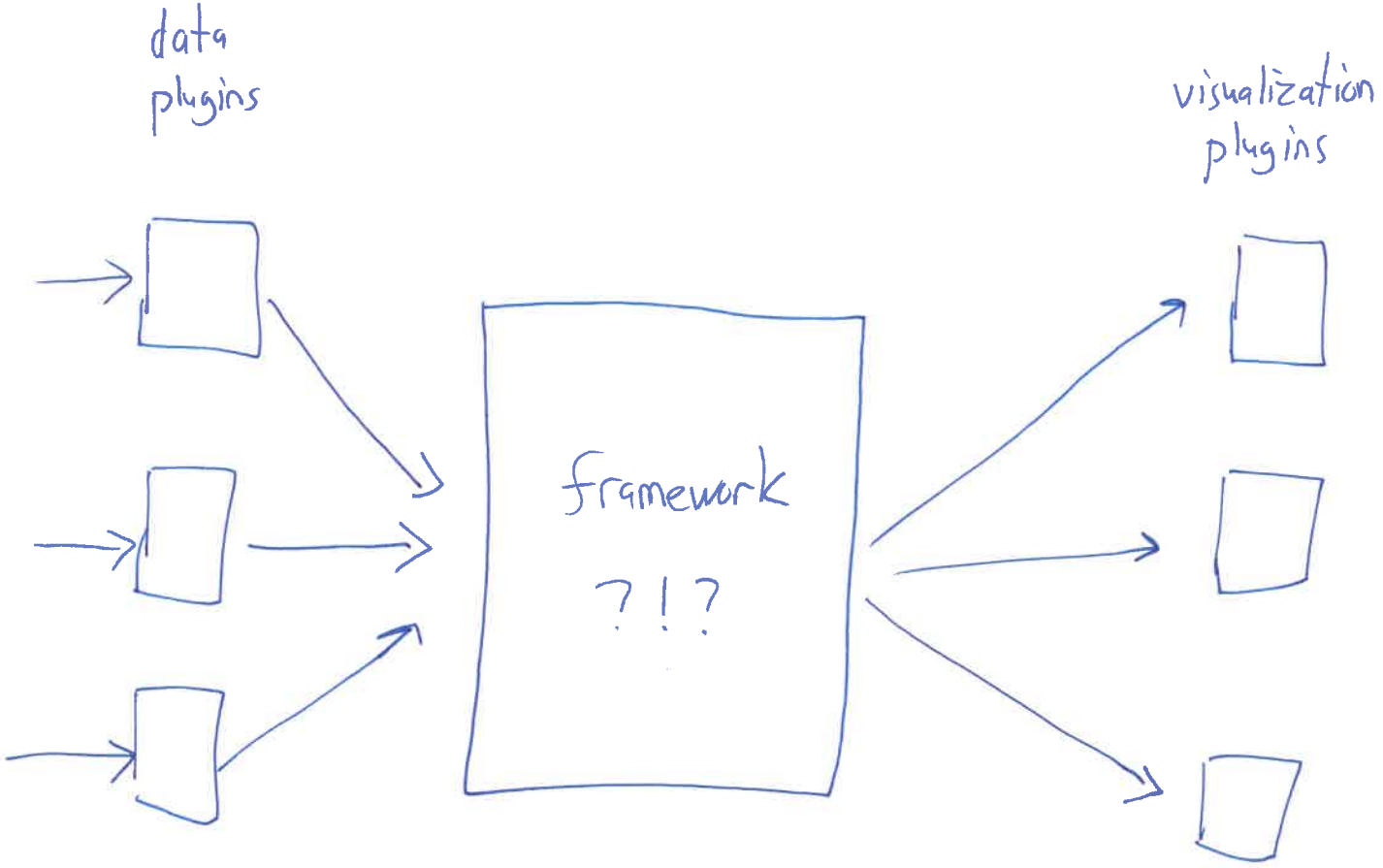
An easy fix: Use a private lock

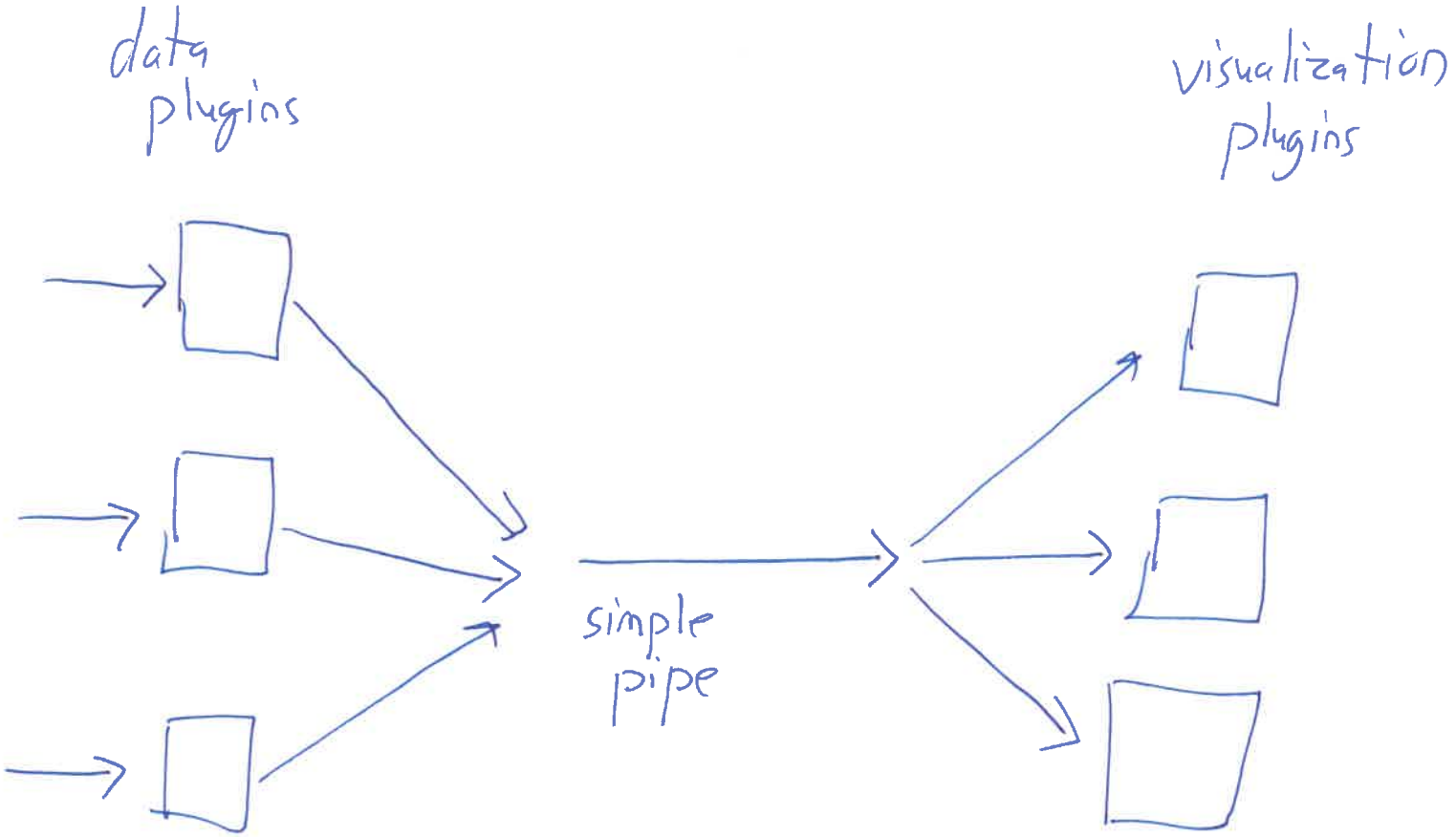
```
public class BankAccount {
  private long balance;
  private final long id = SerialNumber.generateSerialNumber();
  private final Object lock = new Object();

  public BankAccount(long balance) {
    this.balance = balance;
  }

  static void transferFrom(BankAccount source,
                           BankAccount dest, long amount) {
    BankAccount first = source.id < dest.id ? source : dest;
    BankAccount second = first == source ? dest : source;
    synchronized (first.lock) {
        synchronized (second.lock) {
            source.balance -= amount;
            dest.balance += amount;
        }
    }
  } …
```

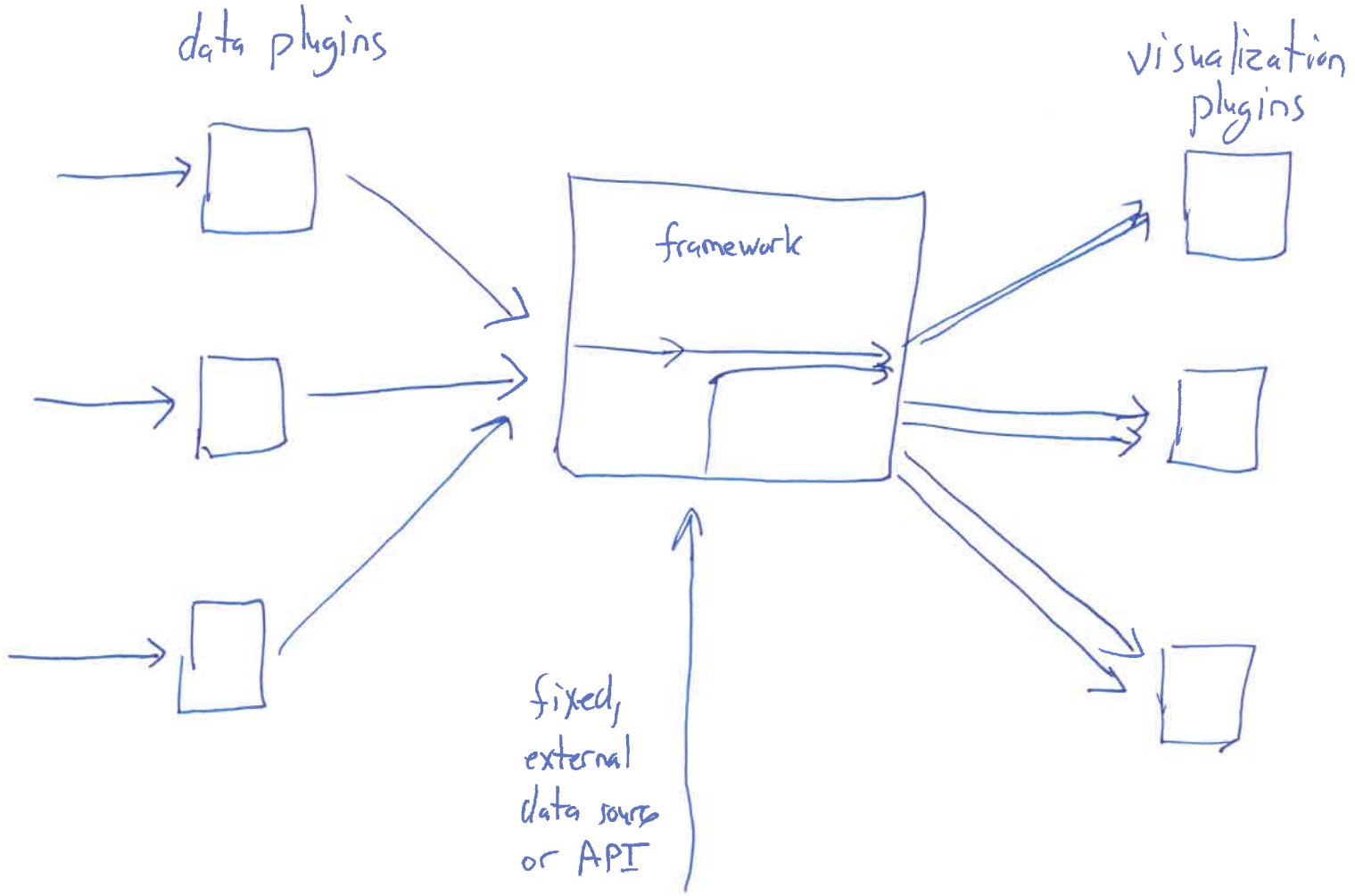
Concurrency and information hiding

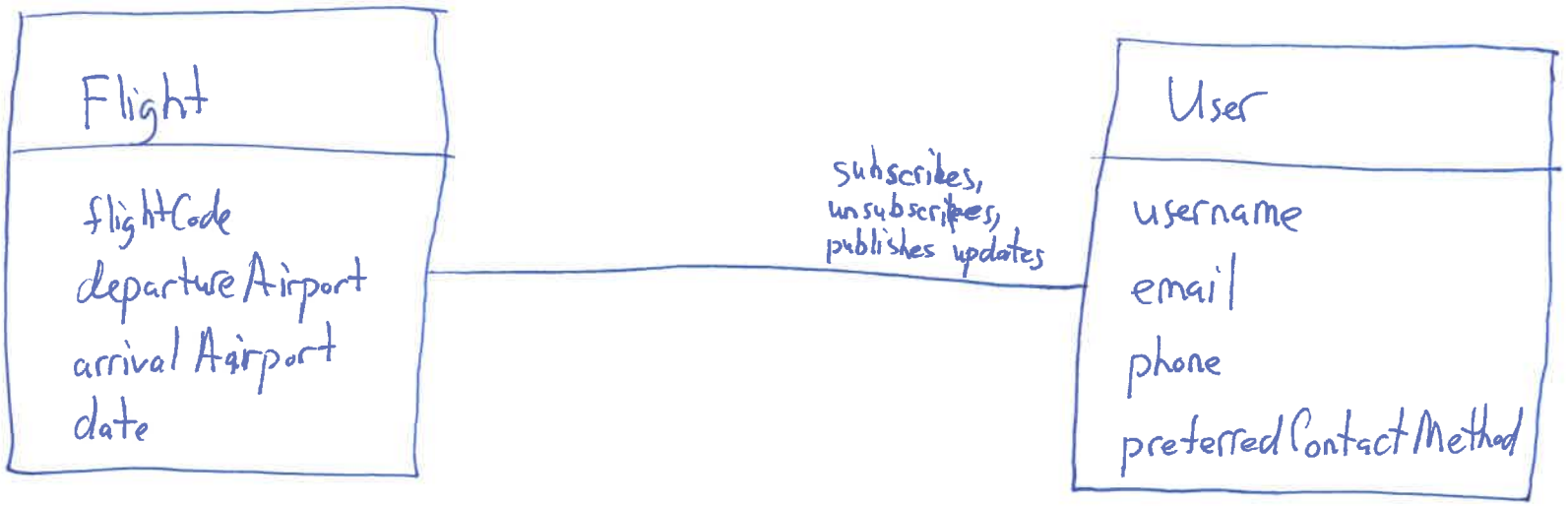
- Encapsulate an object's state: Easier to implement invariants
  - Encapsulate synchronization: Easier to implement synchronization policy

data flow

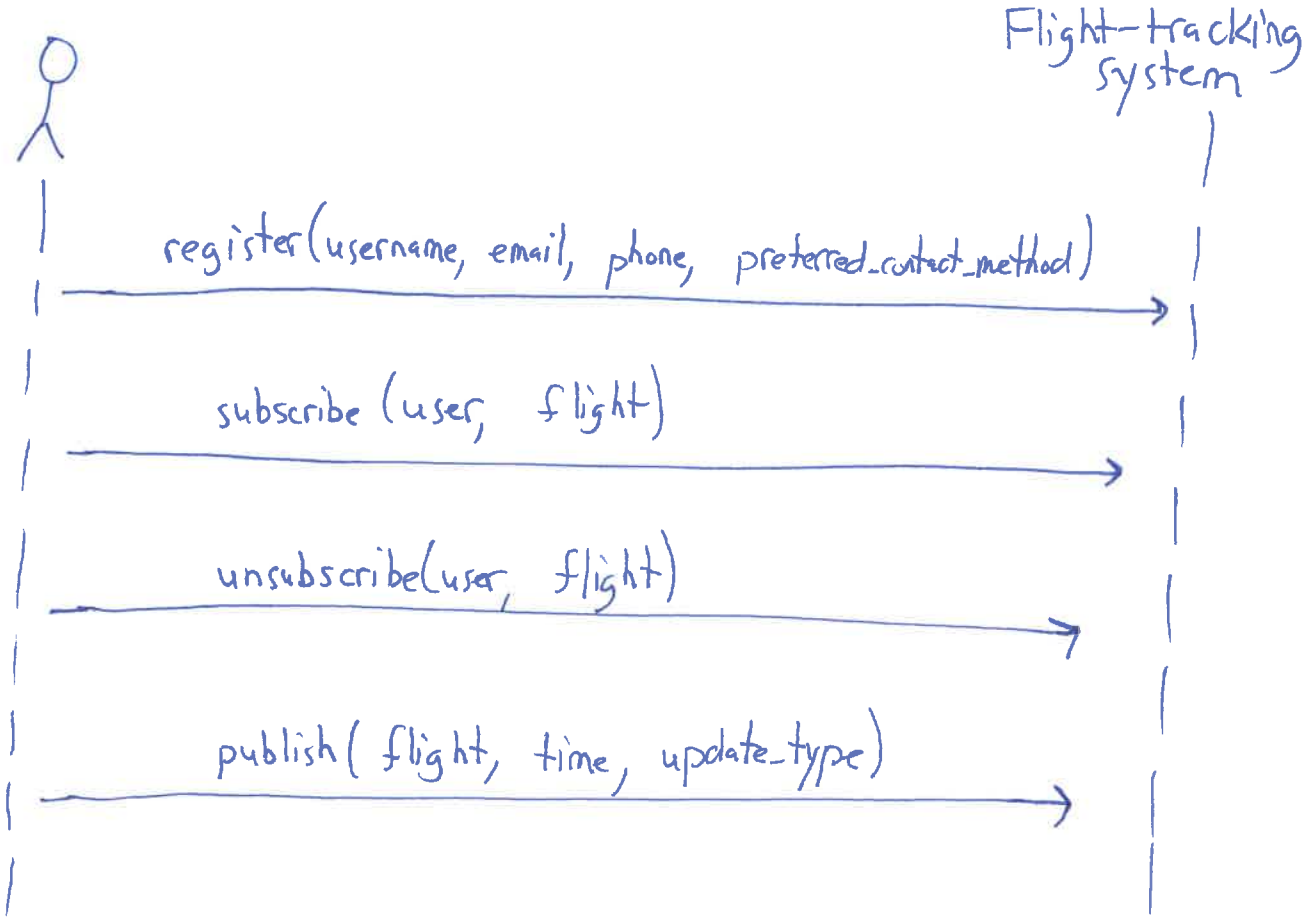


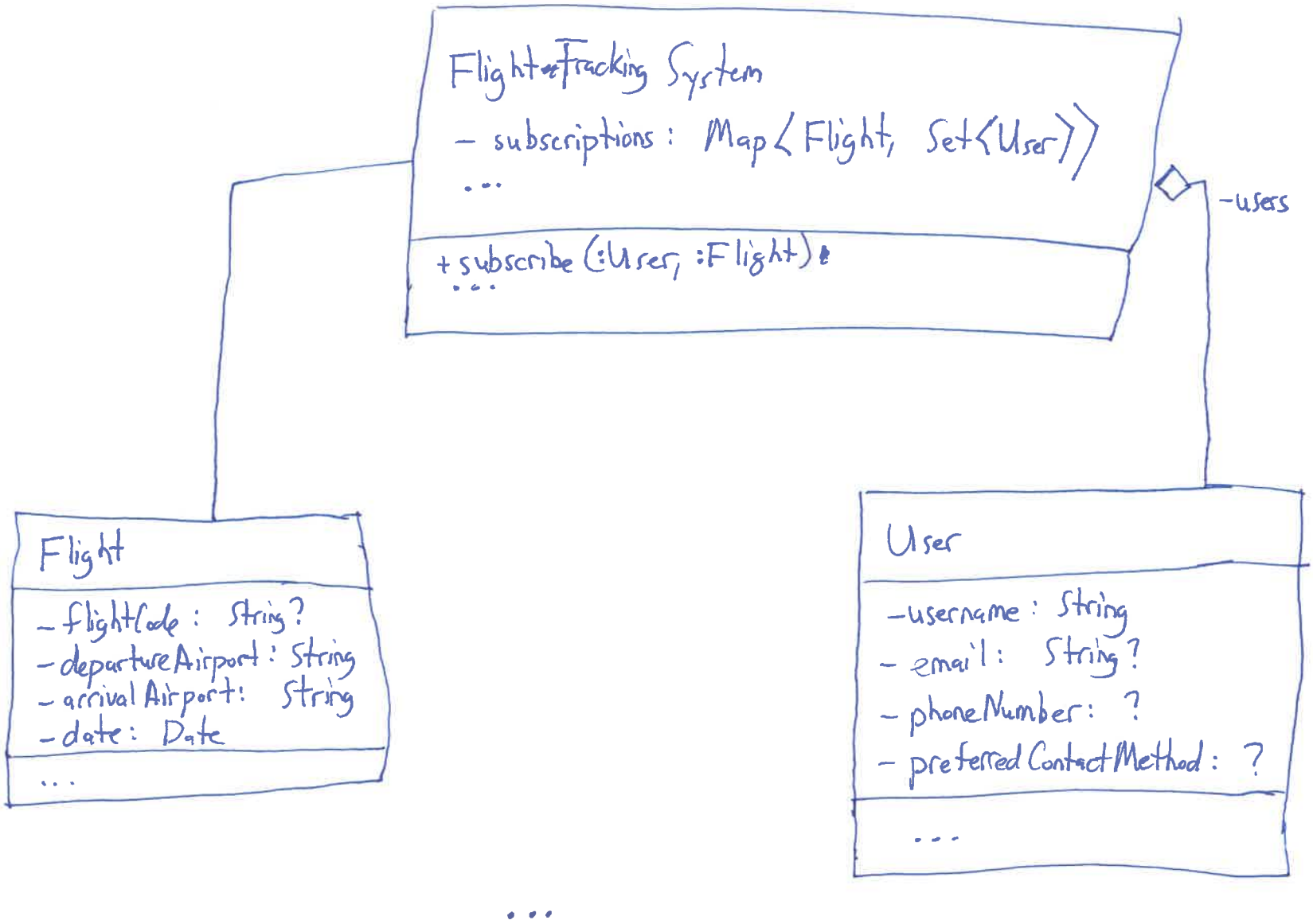

data
plugins
visualization
plugins
simple
pipe

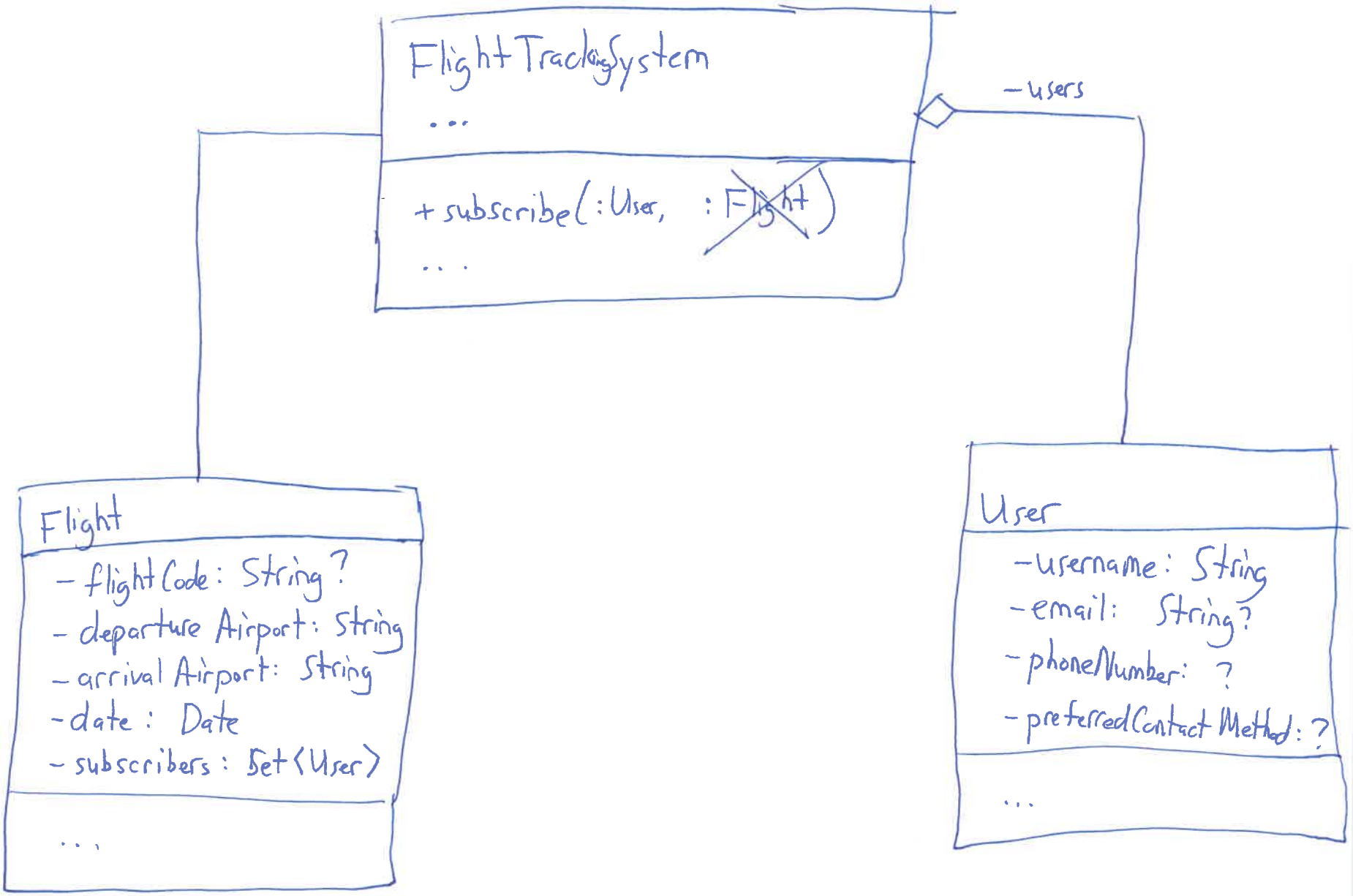

data plugins
visualization plugins
framework
fixed, external data source or API


Flight
flightCode
departureAirport
arrivalAirport
date
subscribes,
unsubscribes,
publishes updates
User
username
email
phone
preferredContactMethod

...


Flight-tracking system
register(username, email, phone, preferred_contact_method)
subscribe (user, flight)
unsubscribe(user, flight)
publish(flight, time, update_type)


FlightTracking System
- subscriptions : Map<Flight, Set<User>>
...
+subscribe(:User, :Flight)
...
-users
Flight
-flightCode : String?
-departureAirport : String
-arrivalAirport : String
-date : Date
...
User
-username : String
-email : String?
-phoneNumber : ?
-preferredContactMethod : ?
...
...


FlightTrackingSystem
...
+subscribe(:User, :Flight)
...
-users
Flight
-flightCode: String?
-departureAirport: String
-arrivalAirport: String
-date: Date
-subscribers: Set<User>
...
User
-username: String
-email: String?
-phoneNumber: ?
-preferredContactMethod: ?
...