

# Performance recording & tuning

- Identify areas to tune
- “Never” tune details during development
- Good performance data will let you:
  - Identify when your application slows down
  - Rule out “I think it’s slower now...” complaints
  - Keep as much detail about the slow call as possible

# How to record performance data?

```
@Aspect
public class PerformanceCollectingAspect {
    private static final long THRESHOLD = 500L;

    @Pointcut("execution(* com.companyname.services..*(..))")
    private void serviceExecution() { }

    @Around("serviceExecution()")
    public Object completeProfiling(ProceedingJoinPoint pjp)
        throws Throwable {

        long startedAt = System.currentTimeMillis();
        try {
            return pjp.proceed();
        } finally {
            long elapsed = System.currentTimeMillis() - startedAt;
            if (elapsed > THRESHOLD) writePerformanceLogEntry(pjp);
        }
    }
}
```

# Developing for performance

- Write @Timed integration tests
- Use the performance collecting aspect
- Your application satisfies the performance requirements
- You can ship the performance logs to the production environment

# Identifying poor performance

- Write performance monitoring aspect that:
  - Measures the invocation times
  - If the time exceeds the recorded average by  $x$  percent, raise an event

# Top 3 Hibernate tips

- Hibernate is powerful, which means that it is easy to use it “incorrectly”
- Session management problems (particularly when combining Hibernate and plain JDBC)
- Problems with large result sets
- Subtle issues with lazily-loaded objects

# Hibernate and JDBC code

- When you are using Hibernate calls and plain JDBC calls in one transaction

```
public class SomeHibernateDao extends
    HibernateDaoSupport implements SomeDao {

    public void save(Some s) {
        getHibernateTemplate().saveOrUpdate(s);
        getHibernateTemplate().flush();
    }

}
```

# Use AOP

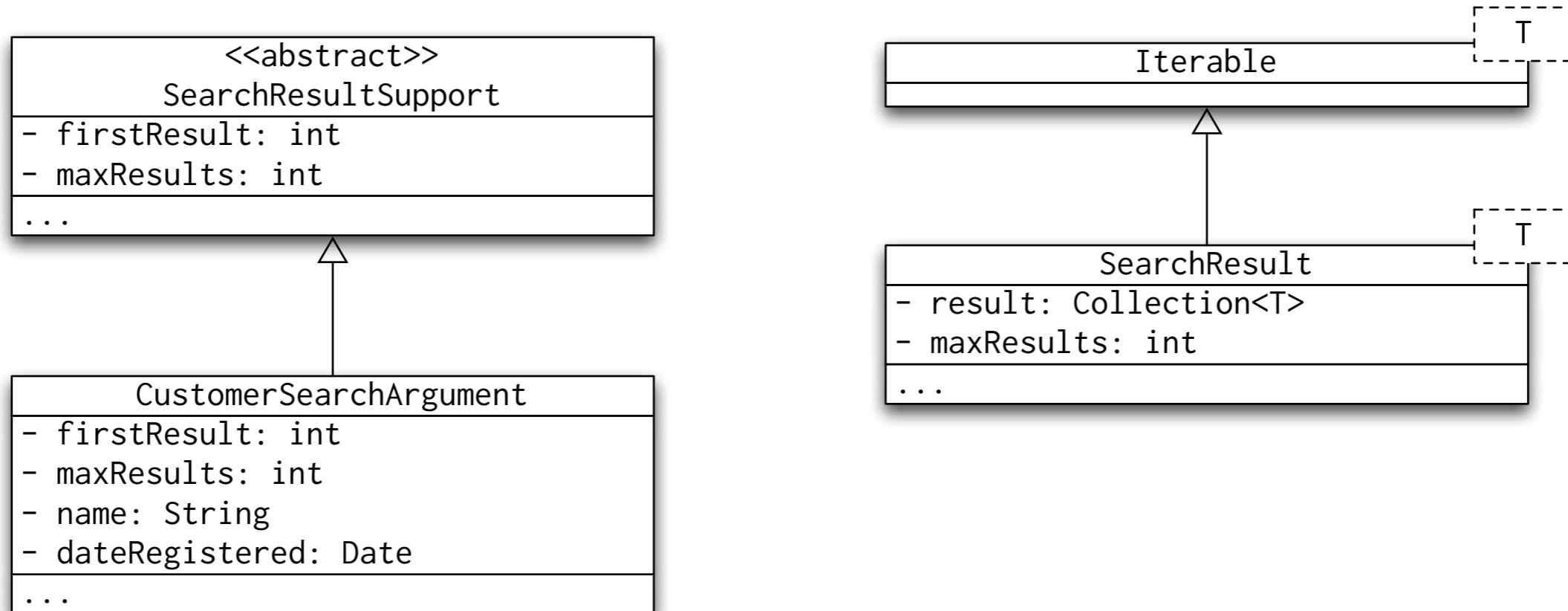
- Create per-control flow aspect

```
public percfow aspect FlushHibernateDao {
    private HibernateTemplate hibernateTemplate;

    pointcut needsFlush: plainJdbcDaoCall
        preceded by hibernateDaoCall;

    before needsFlush() {
        hibernateTemplate.flush();
    }
}
```

# Search argument & result



# Search argument & result

```
public SearchResult<Customer>
  find(final CustomerSearchArgument a) {
  return getHibernateTemplate().execute(
    new HibernateCallback() {
      public Object doInHibernate(Session s) {
        Criteria select =
          session.createCriteria(Customer.class);
        select.setFirstResult(a.getFirstResult());
        select.setMaxResults(a.getMaxResults());
        Criteria count =
          session.createCriteria(Customer.class);
        count.setProjection(Projections.rowCount());
        return new SearchResult<Customer>(
          select.list(),
          count.uniqueResult()
        );
      }
    }
  );
}
```

# Lazy-loading

- Tricky to pass lazily-loaded objects to the clients (over RMI, perhaps)
- If any of the object's properties is lazily-loaded, calling it will throw session closed exception
- There is no magic bullet, but you can write a small aspect that checks that any "service" method does not return CGLIB-enhanced objects