The Search for Zero-Defect Code

Brian Button
Agile Solutions Group
St. Louis, MO
http://www.agilesolutionsgroup.com
bbutton@agilesolutionsgroup.com

9/6/03

Agenda

- Project Description
- Methodology
- Architecture
- Design
- Conclusions and Lessons Learned

Project Description and Overview

- March, 2003
  - Email message to local mailing list
  - Responded, selling Agile Methods and TDD
  - Those skills were differentiator
- Packaging Conveyor control system
  - Prime contractor communicated with client
  - Hardware contractor built the hardware
  - We built the software

Hardware Description

- Input Conveyor
  - Initial bar code reader
  - Product catalog inserters controlled
- Exit Conveyor
  - Cold sealer to wrap brown paper around item
  - Label Printer to affix shipping label
  - Exit bar code reader to verify correct label on correct package

More Hardware Description

- Conveyor belt hardware controller
  - PLC provided by hardware vendor
  - Communicated to via serial port
  - Serial protocol was industry standard DirectNet
- Our server
  - Linux box running Knoppix/Debian
  - Serial ports for bar code readers and PLC
  - Parallel port for label printer

What did I have to control?

- Software had to
  - Read from both bar code reader serial ports
  - Communicate to PLC via its serial port
  - Send print jobs to printer
  - Poll PLC for events

System Parameters

- Original specs had encoder on conveyor that would send event every time belt moved 1”
  - 10 Hz tick rate
  - This tick concept became key architecture concept (more later)
- Rapid processing cycle of 10 Hz led me to implement system in C++ rather than Java, Python, Ruby, etc.

Original Requirements

- Requirements agreed to by hardware vendor in April or so
- Package scanned at entry
  - DB lookup based on bar code
  - Send command to inserters to add correct catalog
  - Format shipping label
  - Queue print job
  - If anything failed, stop system

More Requirements

- PLC would tell me when package exited cold sealer
- Verify scanner would give me bar code to check against expected value. If no match, stop system.

Oh, Yeah

- Just to add a bit of excitement to the project, I would not be able to see the hardware until integration time.
  - Scared the hell out of me
  - Communicated my fear
  - No resolution
  - *Scared the hell out of me*

Initial Architecture

Initial Architecture

- Learned the basics of the system while in California
- Full of excitement, I implemented an initial architectural framework on plane ride home
- Settled on interesting metaphor for this system.
- After much thought, much consideration, after much consternation, I decided that my metaphor would be..........

My Metaphor

- A Conveyor Belt!
- Elegant architecturally
  - There were two conveyor belts
  - Defined Station for each processing element
    - ScanStation, PrintStation, VerifyStation, Terminal
  - Packages added to input conveyor when input bar code reader read a bar code
    - When created, packages knew their location (tick0), got list of all Stations.

More Metaphor

  - For each tick, each Package was told to advance
  - Package iterated through all its Stations, telling it that a new location was available.
  - Stations knew their own locations
  - If Package was in Station, Station did the right thing.
- So friggin' elegant!

Oops!

- Initial architectural framework was developed in a vacuum.
- There was no working code that proved it to be correct.
- It was close, but not quite.
- That baggage slowed me down over next couple of weeks.
- Refactored that baggage out to go faster.

Multithreading?

- It seems like a lot is going on all at once.
- Screams out for multithreading
- How to do that and keep code simple enough that I can get it right?

Separation of Concerns

- Primary architectural concern is to keep separate concerns separate in code
- Threading and business logic are two separate concerns.
  - Should be in different places
- Failure to do this mixes threading logic into business code, making both harder to test

Development Goal

- Goal was to develop code single threaded to get business logic correct and patch in threading later.
- A little fearful about this
- Worked beautifully
- Trick to make it work was Active Object pattern
  - www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf

Implementation Begins

- TDD All The Way!!!
- Began writing tests for most simple thing I could think of
  - ScanStation Operation
    - PackageProcessingAtScan
    - PackageAdvancesThroughTicks
    - PackageHasStopsAssociatedWithIt
    - SingleWidthStationsAreOK
    - etc

Implementation Continues

- Continued writing tests for base features
- After they worked, wrote tests for serial ports, bar code readers, printers, etc.
- System was 90% complete
- Then it happened...

Requirements Changes!!!

- Remember that tick that became part of the architecture?
- Hardware vendor unilaterally changed their mind.
  - No encoder, no ticks, no location information
  - Major architectural change
    - Changed from location-based to event-based architecture

Results of Requirements Change

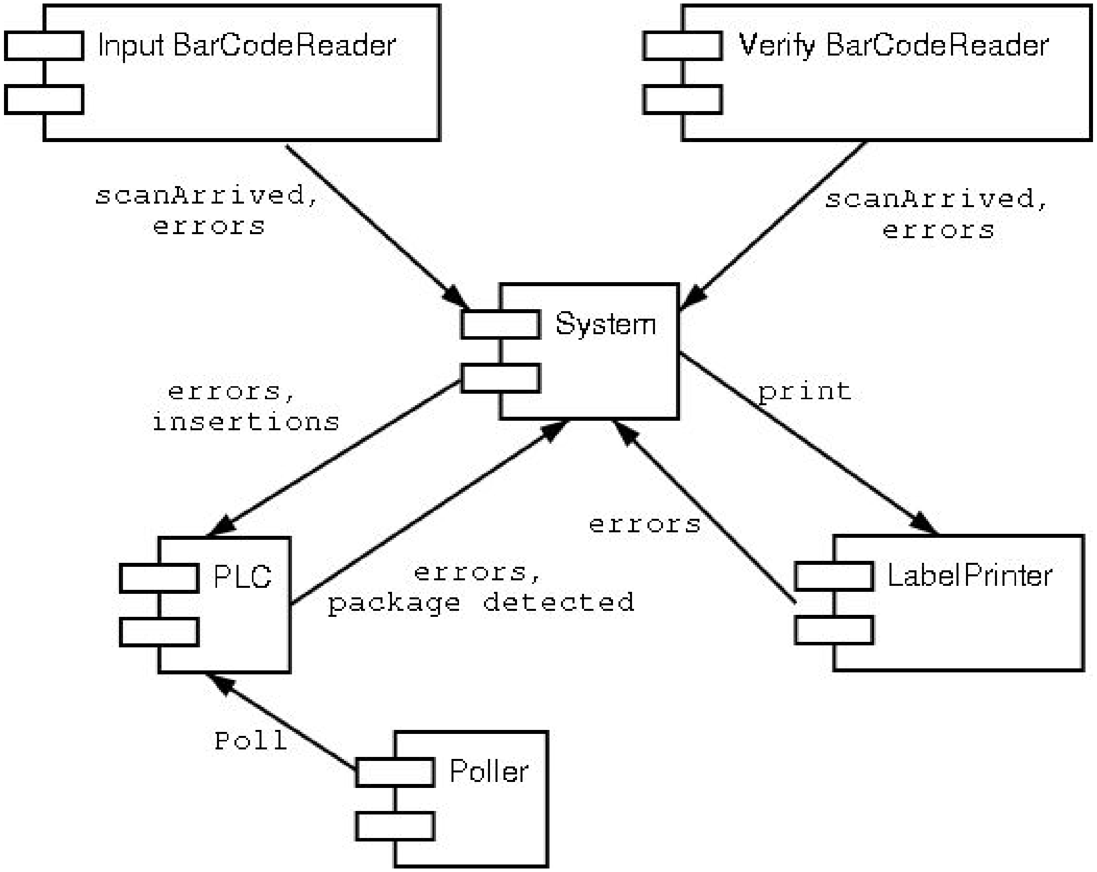
- No problem!
- TDD worked!
- System was loosely coupled
- Tore out heart of application and started over
- Reimplemented core of system
  - Brought over extra classes as they were needed
  - 5 days to reimplement whole core

Detailed Look at Code

- Enough of this talking
- Let's see some tests and code!
  - In order of interest to me, not implementation order

Subsystem Diagram

- Independent Subsystems in MPS



How do subsystems communicate?

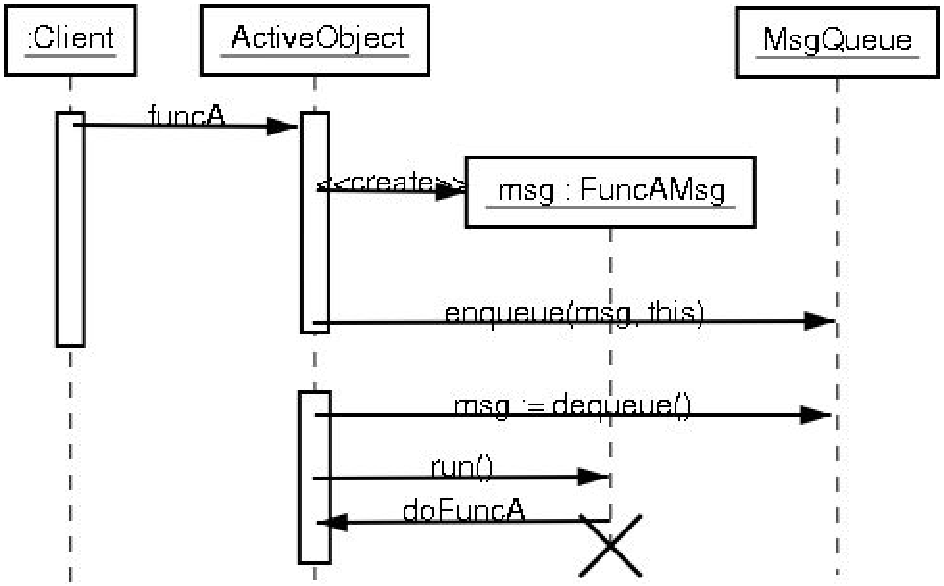
- Each subsystem represents an independent activity
- Any of them could be active at any time
- Implies multithreading and all its associated problems.
- Active Object pattern designed to solve this.

Active Object

- Active Object pattern separates act of invoking a method from method execution
  - Caller invokes method and returns
  - Receiver executes method in its own thread and calls back results in same thread
  - Any results that cross to another Active Object have to return them using the same mechanism
- Result is that each Active Object is really single threaded within itself

Active Object Sequence Diagram

- Client calls funcA() in his thread, msg created and queued.
- ActiveObject runs in its own thread, dequeues the msg, and executes it



Producer/Consumer Queue

- Main architectural class of entire project
- Accepts msgs queued in thread of caller
- Returns them to Active Objects in AO's own thread.
- This class has **got** to work, or nothing else will.

Producer/Consumer Queue Tests

- First unit test
  - Producer and Consumer are defined in test case
  - Producer adds one int to queue
  - Consumer pulls it off in different thread
  - Queue should be empty at end

```
TEST(putOneOnTakeOneOff, PCQ)
{
  ProducerConsumerQueue<int> queue;

  Producer p(queue);
  Consumer c(queue);

  boost::thread  consumerThread(c);
  boost::thread  producerThread(p);

  producerThread.join();
  consumerThread.join();

  CHECK(queue.isEmpty());
}
```

Producer/Consumer Queue Tests (cont)

- Second test – stress test
  - CountingConsumer like Consumer, but it also counts number of ints removed from queues
  - CountingProducer adds an int whose value increases monotonically
  - Test adds 600000 ints through 5 CountingProducers and confirms that they are all pulled off successfully
  - Just to give confidence that queue works

Producer/Consumer Queue Stress Test Code

```
TEST(stressTest, PCQ)
{
  ProducerConsumerQueue<int>  queue;
  CountingProducer p1(queue, 100000);
  CountingProducer p2(queue, 120000);
  CountingProducer p3(queue, 110000);
  CountingProducer p4(queue, 140000);
  CountingProducer p5(queue, 130000);
  CountingConsumer c1(queue);

  boost::thread c(c1);

  boost::thread t1(p1);
  boost::thread t2(p2);
  boost::thread t3(p3);
  boost::thread t4(p4);
  boost::thread t5(p5);

  t5.join();
  t4.join();
  t3.join();
  t2.join();
  t1.join();

  for(int i = 0; i < 10000 && (queue.getDepth() > 0); i++)
  {
    boost::thread::yield();
  }

  LONGS_EQUAL(0, queue.getDepth());
  LONGS_EQUAL(600000, c1.getCount());

  c1.stop();

  // Stop Consumer thread by forcing it through its loop one more time after I
  // set stop to true.
  CountingProducer terminator(queue, 1);
  boost::thread tthread(terminator);
  tthread.join();

  c.join();
}
```

ProducerConsumerQueue<> code

```
template<class T>
class ProducerConsumerQueue
{
  public:
    ProducerConsumerQueue() {}
    ~ProducerConsumerQueue() {}

    void  enqueue(T msg)
    {
      boost::mutex::scoped_lock lock(guard);
      messageQueue.push_front(msg);
      messagePending.notify_one();
    }

    bool isEmpty() const
    {
      return messageQueue.empty();
    }
```

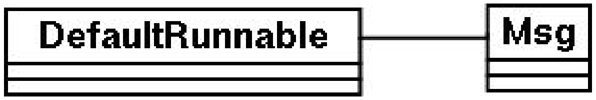
```
    int getDepth() const
    {
      return messageQueue.size();
    }

    T     dequeue()
    {
      boost::mutex::scoped_lock lock(guard);
      while(messageQueue.empty())
      {
        messagePending.wait(lock);
      }

      T msgToReturn = messageQueue.back();
      messageQueue.pop_back();

      return msgToReturn;
    }

  private:
    boost::mutex      guard;
    boost::condition  messagePending;
    std::deque<T>     messageQueue;
};
```

DefaultRunnable tests

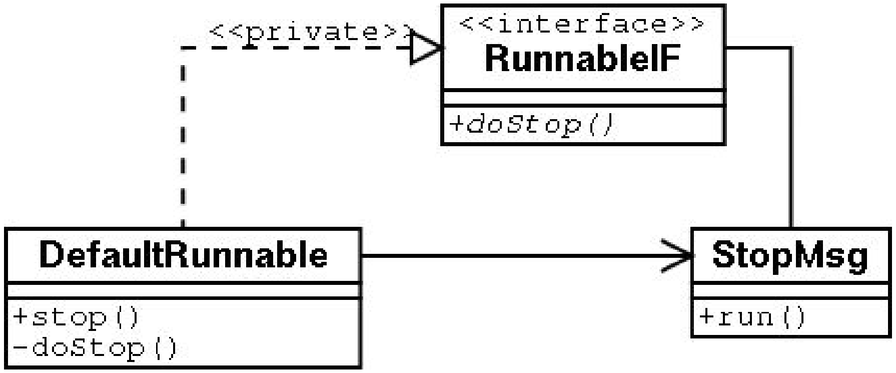
- ProducerConsumerQueue enables messages to pass between threads.
- DefaultRunnable is the base class for all ActiveObjects in system
- Problem exists in how callbacks work
  - ActiveObject queues msg, giving msg a pointer back to ActiveObject for callback
  - ActiveObject depends on Msg class, and Msg class depends on ActiveObject

Stupid C++ Tricks

- Private Interface Callback pattern



  - Dependency cycle needs to be fixed



Private Interface Callback Pattern

- DefaultRunnable has public stop() method
- DefaultRunnable has *private* doStop() method
- DefaultRunnable has *private* base class
- Clients invoke stop()
- Classes calling back get DefaultRunnable pointer as its private base, RunnableIF, and call its *public* doStop() method
- Dependency cycle is broken

DefaultRunnable Test support code

```
class Runnable
{
  public:
    Runnable() : keepGoingFlag(new bool(true)) {}
    Runnable(const Runnable & other)
      : keepGoingFlag(other.keepGoingFlag) {}
    virtual ~Runnable();
    virtual void operator()() = 0;
    virtual void start() = 0;
    virtual void stop() { *keepGoingFlag = false; }

  protected:
    virtual bool keepGoing() const
    {
       return *keepGoingFlag;
    }

  private:
    boost::shared_ptr<bool> keepGoingFlag;
};
```

```
class ThreadedClass : public Runnable
{
  public:
    ThreadedClass() : counter(new int(0)),
           processMessages(new bool(false)) {}
    ThreadedClass(const ThreadedClass & other)
      : Runnable(other),
        counter(other.counter),
        processMessages(other.processMessages)
    {}

    void operator()()
    {
      while(keepGoing())
      {
        if(*processMessages)
        {
          (*counter)++;
        }
      }
    }

    void start() { *processMessages = true; }

    int getCounter() const { return *counter; }

  private:
    boost::shared_ptr<int> counter;
    boost::shared_ptr<bool> processMessages;
};
```

DefaultRunnable Test support code #2

```
class ChildCallbackIF
{
  public:
    virtual ~ChildCallbackIF();
    virtual void callMe() = 0;
};

ChildCallbackIF::~ChildCallbackIF() {}

class ChildMsg : public RunnableMsg
{
  public:
    ChildMsg(ChildCallbackIF & child_)
      : child(child_) {}
    void run() { child.callMe(); }

    ChildCallbackIF & child;
};
```

```
class Child1 : public DefaultRunnable, private ChildCallbackIF
  {
    public:
      Child1() : counter(new int(0)) {}
      Child1(const Child1 &  other)
        : DefaultRunnable(other), counter(other.counter) {}
      ~Child1() {}

      void incrementCounter()
      {
        boost::shared_ptr<RunnableMsg> msg(new ChildMsg(*this));
        queue->enqueue(msg);
      }

      int getCounter() const { return *counter; }

    private:
      boost::shared_ptr<int> counter;

      void callMe() { (*counter)++; }
  };
```

DefaultRunnable test code

```
TEST(testStopMsg, RunnableTest)
{
  boost::shared_ptr<Child1> child1(new Child1);

  ThreadManager mgr;
  mgr.addThread(child1);

  mgr.stopAll();

  CHECK(true);
}

TEST(counterIncremented, RunnableTest)
{
  boost::shared_ptr<Child1> child1(new Child1);

  ThreadManager mgr;
  mgr.addThread(child1);

  child1->incrementCounter();
  mgr.stopAll();

  LONGS_EQUAL(1, child1->getCounter());
}

TEST(nothingPushedUntilStartIsCalled, RunnableTest)
{
  boost::shared_ptr<ThreadedClass>
        threadedClass(new ThreadedClass);

  ThreadManager mgr;
  mgr.addThread(threadedClass);

  LONGS_EQUAL(0, threadedClass->getCounter());
}
```

```
TEST(somethingIsPushedAfterStartIsCalled, RunnableTest)
{
  boost::shared_ptr<ThreadedClass> threadedClass(new ThreadedClass);
  boost::thread ourThread(*threadedClass);

  threadedClass->start();

  ThreadManager::wait();

  CHECK(threadedClass->getCounter() > 0);
}

TEST(threadsCanBeJoinedAfterStopCalled, RunnableTest)
{
  boost::shared_ptr<ThreadedClass> threadedClass(new ThreadedClass);
  boost::thread ourThread(*threadedClass);

  threadedClass->stop();
  ourThread.join();
}

TEST(threadsCanBeCollectedAndStopped, RunnableTest)
{
  boost::shared_ptr<ThreadedClass> threadedClass1(new ThreadedClass);
  boost::shared_ptr<ThreadedClass2> threadedClass2(new ThreadedClass2);
  boost::shared_ptr<ThreadedClass> threadedClass3(new ThreadedClass);
  boost::shared_ptr<ThreadedClass2> threadedClass4(new ThreadedClass2);

  {
    ThreadManager mgr;
    mgr.addThread(threadedClass1);
    mgr.addThread(threadedClass2);
    mgr.addThread(threadedClass3);
    mgr.addThread(threadedClass4);
  }

  CHECK(true);
}
```

DefaultRunnable code

```
class RunnableMsg
{
  public:
    virtual ~RunnableMsg();
    virtual void run() = 0;
};

class RunnableIF
{
  public:
    virtual ~RunnableIF();
    virtual void doStop() = 0;
};

class DefaultRunnable : public Runnable, protected RunnableIF
{
  public:
    DefaultRunnable();
    DefaultRunnable(const DefaultRunnable & other);
    ~DefaultRunnable() {}

    void start() {}
    void stop();
    void operator()();

  protected:
    virtual void  runNextCommand();

    boost::shared_ptr<ProducerConsumerQueue<boost::shared_ptr<RunnableMsg> > > queue;

  private:
    void doStop() { Runnable::stop(); }
};
```

DefaultRunnable code #2

```
namespace
{
  class StopMsg : public RunnableMsg
  {
    public:
      StopMsg(RunnableIF & callback_) : callback(callback_) {}
      void run() { callback.doStop(); }

      RunnableIF & callback;
  };
}

DefaultRunnable::DefaultRunnable()
  : Runnable(),
    queue(new ProducerConsumerQueue<boost::shared_ptr<RunnableMsg> >)
{
}

DefaultRunnable::DefaultRunnable(const DefaultRunnable & other)
  : Runnable(other),
    RunnableIF(other),
    queue(other.queue)
{
}
```

```
void DefaultRunnable::stop()
{
  boost::shared_ptr<RunnableMsg> msg(new StopMsg(*this));
  queue->enqueue(msg);
}

void DefaultRunnable::operator()()
{
  while(keepGoing())
  {
    runNextCommand();
  }
}

void DefaultRunnable::runNextCommand()
{
  boost::shared_ptr<RunnableMsg> msg = queue->dequeue();
  msg->run();
}
```

ThreadManager

- Needed a class to collect Runnables
  - Add to collection
  - Stop all
  - Wait for all to stop
- Similar to boost::thread_group
  - But did extra stuff, so I had to write my own
- Tested along with DefaultRunnable

ThreadManager code

```
class ThreadManager
{
  public:
    ThreadManager();
    ThreadManager(const ThreadManager &);
    ~ThreadManager();

    template<class RunnableType> void addThread(boost::shared_ptr<RunnableType> runnable)
    {
      runnables->push_back(runnable);

      boost::thread * t = new boost::thread(*runnable);
      threads->add_thread(t);
    }

    void stopAll()
    {
      for(vector<boost::shared_ptr<Runnable> >::iterator iter = runnables->begin();
          iter != runnables->end();
          iter++)
      {
        boost::shared_ptr<Runnable> runnable = *iter;
        runnable->stop();
      }
      threads->join_all();
    }

    void waitForAllThreadsToExit() { threads->join_all(); }
    static void wait(int yields = 100) { for(int i = 0; i < yields; i++) boost::thread::yield(); }

  private:
    boost::shared_ptr<std::vector<boost::shared_ptr<Runnable> > > runnables;
    boost::shared_ptr<boost::thread_group>                        threads;
};
```

Conclusions

- I started developing code in a vacuum. That code caused me trouble. Don't do that.
- I felt pressure on site during integration to make changes without updating/creating tests, and succumbed to it for a while.
  - After a short time (couple hours), I began to be afraid to change my code
  - I updated all tests and avoided that temptation the rest of the trip. I was much happier.

Final Result

- ***Zero*** bugs in installed system
  - Zero Defect Software!!
- At integration, I had a little problem for about 3 hours with a communication protocol misunderstanding. Once fixed, it worked immediately.
- Rest of system has worked flawlessly
- Not me, it was the process.

Future Projects

- Articles coming every week or so on other features, interesting concepts, lessons learned during this project.
- Will be posted to web each week
  - http://www.agilesolutionsgroup.com
- Another project is possible right now based on this codebase
- Changes in that project will drive further abstraction and refactoring. I'll report back on that later.

Future Articles

- Implementing Communications Protocol using Test Driven Development Without Access to Hardware
- Using Decorator Pattern to Add Logging to System
- Multithreaded Unit Testing with Active Objects
- Evolution of Label Printing and Formatting using Boost Regexp Library

Feedback, please!!!

- This presentation created in a vacuum.
- You are my customers
- What questions did I leave unanswered?
- What did I explain badly or not at all?
- What else should we talk about?
- Respond on mailing list
  - http://groups.yahoo.com/group/xpstl