Cross-Paradigm Programming

Larry Jones

Software Architect, Designer,

Lead Developer

Why are we here?

Topic	Slide
The Setup: Clojure	3
Everyone Wants to be a Physicist	5
Let's Go Back in Time (with apologies to Huey Lewis and the News)	6
I Was Told There Would be Objects	7
"Object-Oriented" Clojure	10
Can We Make It? (with apologies to the "Six Million Dollar Man")	12
And Can We Make It Better	13
Houston, We Have Results	17
Can We Make it "More" Object-Oriented?	19
Does Clojure Do Objects?	23

I thought someone said "beer" ☺

A long time ago in a job far away…

- I attended a class entitled “Complex Problem Solving.”
- It used Scheme to demonstrate how we solve problems.
- I was amazed at how such a simple language could solve such complicated problems.
- That was 25 years ago…
- I’ve been looking to repeat that feeling for 25 years.

And then I found Clojure.

Clojure purports to be better.

The Benefits

- A simpler language.
- Immutable data.
- Interoperability.
- Expressive.

The Concerns

- “Lots of Insignificant Silly Parentheses.”
- Structured programming redux?
- Harder to “grok” than objects?

As a physicist, I experiment.

This experiment has several goals.

- Determine how to implement a solution in Clojure for a problem exhibiting inheritance.
- Determine how to map standard design patterns to Clojure.
- Compare and contrast refactoring in Clojure with refactoring in other object-oriented languages.

Let's experiment with a time series.

- From Wikipedia: "… a **time series** is a sequence of data points, measured typically at successive times spaced at uniform time intervals."
- A time series is:
  - A sequence of measurements
  - In chronological order
  - In which each measurement has an associated time stamp.

We use an interface as our model.

```
public interface TimeSeries {
    public String getName();
    public Calendar getStartTime();
    public TimePeriod getPeriod();
    public ArrayList<DataPoint> getSamples();
    public void AddPoint(DataPoint aNewPoint);
    public DataPoint getPoint(int k);
};
```

Where's the object? (with apologies to Wendy's)

But how has technology changed?

Then

- Data points **must** be perfectly, regularly spaced.
  - Time stamps “sanitized.”
- Every sample **must** have a floating point value.
  - Sentinel value for “bad data.”
- Lab data not well handled.

And New

- Algorithms can take advantage of actual time stamp.
- Avoid “magic” values (which must be known/passed to all functions).
- Can we handle lab data?

Combining these problems suggests a class hierarchy.

- TimeSeries (client interface)
  - AbstractTimeSeries (common implementation)
    - RegularTimeSeries (sanitized, legacy data).
    - AlmostRegularTimeSeries (data collected today).
    - IrregularTimeSeries (sparse, irregular data).

public abstract class AbstractTimeSeries…

We begin by modeling the data.

- Texts recommend using maps as “objects”

```
{   :name “phaetra”,
    :start <GregorianCalendar…>,
    :period 60, ; in seconds
    :bad-data -9999.,
    :measurements [1025.731, …] }
```

Algorithms + Data Structures = Programs?

Abstraction barriers hide the details.

The Problem

- Unencapsulated data

The Solution

- Abstraction Barriers
  - From The Structure and Interpretation of Computer Programs
  - Constructors
    - Create our map.
  - Selectors
    - Query or change the map details.

We create an instance to test our ideas.

```
(defn make-time-series
    [name start period bad-data measurements]
    “Make a time series.”
    { :name name,
      :start start,
      : bad-data bad-data
      : measurements measurements})
```

Use the source! (more apologies)

Now that we are “green,” we refactor.

- Use the “red-green-refactor” cycle
  - Write a unit test. It fails. (Red)
  - Change the code until the test passes. (Green)
  - Refactor.
  - Repeat until we run out of time or out of money.

Disciplined way to improve the code without breaking any of it.

Refactoring improves our code.

Before

- Duplicate map initialization.
- Duplicate implementation of samples function.
- No sign of hierarchy.

After

- Common initialization of common fields.
- Hierarchy present – but not very obvious.
- Elimination of samples duplication (except I missed it!).

Its beginning to look a lot like objects….

We then add points to our time series.

- The function add-point:
  - Uses arity overloads for different concrete types.
  - But must also include asserts to protect against mistakes by callers.
  - It could also have been a multimethod.
- The function get-point:
  - Is a multimethod.
  - With two **identical** methods.

Even I'm not perfect. (Ask my children.)

Passing unit tests reduce the risk of refactoring breaking existing code.

- Regular and irregular time series have much in common.
  - Capture commonality using an ad-hoc hierarchy.
  - But now, tags must be qualified.
  - A common parent need not actually exist.

I still have not noticed the duplication in samples.

We can now draw some conclusions.

- Abstraction barriers provide similar benefits to private and public class members.
- Namespaces encapsulate implementation details.
- Multimethods allow one to model implementation inheritance hierarchies.
- Ad hoc hierarchies capture other inheritance relationships.

But we also have some additional questions.

- What about datatypes and protocols?
  - Introduced in version 1.2.
- How might we more effectively refactor Clojure code?
  - Extract method fairly obvious.
  - Others – not so much.
  - Refactoring an expression oriented language.
- What about higher order methods?

That's what I love about science: It's never settled.

Let's explore some of those questions.

- How effectively do datatypes and protocols support object-oriented solutions?
- How might we more effectively refactor Clojure code?

Back to the first step in the scientific method!

Datatypes and protocols support abstraction and performance.

Datatypes

- Functionality of maps.
- But faster than maps (and defstructs).
- Similar performance to Java instance member access.
- Support Java interfaces.
- And Clojure protocols.

Protocols

- Allows one to specify behavior.
- Without specifying an implementation.
- “Automagically” supports polymorphic dispatch based on type of first argument.

A rose by any other name…. You know the drill.

Since we have a working solution, we refactor.

- Replace our maps with a datatype
  - Supports the same abstract interface as maps.
  - Provides faster performance (I'm told).
- Protocols replace our multimethod hierarchy.
  - Without creating a hierarchy.

Protocols support design by "duck typing."

This change provides additional “data points” for our experiment.

Before Clojure 1.2

- Maps + Functions
  - Class implementations are idiomatic not structural.
- Multimethods
  - Provide obvious hierarchies.

In Clojure 1.2

- Datatype
  - More clearly expresses our “class” intent.
- Protocols
  - Do **not** support hierarchies.
  - But do provide more modularity for “classes.”

Data + Functions = Classes?

Is Clojure the right tool?

- Maps
  - Model class state implicity.
  - Must use idioms to provide modularity.
- Multimethods
  - Model obvious hierarchies.
  - Easily support other hierarchies.
- Datatypes
  - Model class state more explicity.
  - Protocols increase modularity.
- Protocols
  - Support “duck typing.”
  - Do **not** support inheritance.

But this problem involves no mutable state…
To be continued…

Resources.

- My email: mrwizard82d1@gmail.com.
- The source code: git@github.com:mrwizard82d1/chug-time-series.git.
- Programming Clojure. Stuart Halloway. Pragmatic Programmers.
- Practical Clojure. Luke VanderHart and Stuart Sierra. Apress.