Practice, Practice, Practices

Christopher R. Genovese

Department of Statistics & Data Science

Tue 23 Sep 2025
Session #9

Plan

Style Practices and Principles

Plan

Style Practices and Principles

Revision and Redesign: A Case Study

Plan

Style Practices and Principles

Revision and Redesign: A Case Study

Building State Machines

Plan

Style Practices and Principles

Revision and Redesign: A Case Study

Building State Machines

Practice Activity: Representing Disjoint Sets

Announcements

- **Homework**:
  - No migit questions???
  - **laser-tag** assignment due Tue 30 Sep. Available on github problem bank.
  - Push mini-assignment **union-find** when done

Goals for Today

- Style and Best Practices
- Revision and Redesign: State Machine Upgrades as a Case Study
- Activity: Representing Disjoint Sets
- Debugging (if time allows)

Plan

Style Practices and Principles

Revision and Redesign: A Case Study

Building State Machines

Practice Activity: Representing Disjoint Sets

Best Practices

Programming is a form of communication to *two* audiences: the computer and human readers (including future you).

As long as your code is syntactically correct, the computer will run it, for better or worse. But your code will be checked, studied, tested, documented, debugged, used, modified, generalized, and reused by humans. To get the most value from your time spent programming, you need to pay attention to how *humans* process your code.

Indeed, the features that characterize good code are very similar in spirit to the features that characterize good writing.

There are many details to manage in a complex piece of code, and consequently there are many detailed choices to consider in practice. These include matters of

- style
- *naming*
- documentation
- organization
- design
- dependence
- error handling
- tooling

See our rubric or e.g., the book *Code Complete* by Steve McConnell. These are worth reading and studying.

But for our purposes today, we can cover a lot of ground with only **a few basic principles**.

Write Code to be Read

Code *communicates ideas* and *describes abstractions*, often complicated ones. Its execution is like an unfolding story, with characters traveling along their own narrative arcs.

Try to maximize the **ease** and **clarity** with which the reader can process the code. Help them to

- understand the ideas/abstractions behind the code
- identify the characters/entities involved, and
- follow the story.

A good principle for writing/presentation: **prepare your reader for the information you are about to give**.

Here are a few implications of this principle:

- Format your code to make it easy to read
- Use meaningful, concrete, and descriptive *names*
- Arrange your code to bring out the central idea in each chunk
- Make critical relationships salient
- Structure your interfaces to present a clean and consistent abstraction
- Avoid hidden side effects and obscure features
- Use documentation to supplement code not mimic it

Write Code to be Read: Documentation

Give readers an entry point for understanding how the code is used, how data flows, et cetera. Examples and description can help, even if brief. Section labels and pointers to entry points can be helpful. Tests are a form of documentation too.

Use docstrings/structured comments for nontrivial functions.

The code can impart meaning on details – on the how – but not as easily on the why or when. Comments help there.

If you write a *clever* piece of code, first ask if you *need to be clever* and if so, consider documenting the goals of the code, constraints, reasons, etc.

Task for sharpening: Go to an open-source repository that interests you and **read the code**. What works for you? What doesn't? What *makes you work*?

Be consistent

A *foolish* consistency may be the hobgoblin of little minds, but for programming, a practical consistency is helpful to in many ways.

Here are a few implications of this principle.

- Use consistent *formatting, spacing, and style*
- Use consistent *naming schemes* for variables, functions, classes, and files (`CamelCase`, `kebab-case`, `snake_case`, `ALL_CAPS`)
- Use consistent documentation formatting, style, and scope
- Use consistent *interfaces* to functions and classes
- Use consistent error handling

Many conventions for naming, formatting, spacing, etc. are included in *style guides* used by projects or programming languages.

For example, PEP 8 describes naming and formatting conventions for Python code, and *your code will be expected to follow it*. (PEP 8 is unusual because nearly every major Python project uses it.) R has a lot of historical cruft that means nobody uses the exact same style, but the tidyverse style guide is a good reference. Read these guides!

Don't Repeat Yourself

Seriously, don't repeat yourself. It's inefficient to repeat yourself. So don't do it. Really.

Keep your code DRY! (Not WET – wasting everyone's time!)

Each piece of knowledge embodied in the code should have one unambiguous and authoritative representation.

Here are a few implications of this principle.

- If you find yourself repeating a piece of code, put it in a function.
- If you find yourself using a number or other literal, make it a *named constant*. (Besides a few basic cases such as 0, 1.)
- Documentation should not merely repeat what the code does but should add value. For instance: why, who, when?

It's easier to chew small pieces

Any stretch of code focuses on a few key ideas. Organizing your code to bring out one idea at a time, rearranging as needed.

- Organize your code modularly (paragraphs, functions, files)
- Prefer functions that do *one* thing well
- Prefer orthogonality (decoupling)
- Prefer functions/classes/modules with a distinct purpose and identity

Coupling: Consider how a change in your code/design/interface/... will cascade through your whole codebase.

Keep the contract clear

Each function or class has an explicit contract behind it. *"I give you this, you give me that."*

Make that contract salient in your code, your *names*, your tests, and your documentation.

An implication: **separate calculations, actions, and data** (Referential transparency is a good goal in any paradigm.)

An idea we will discuss: using language features to enforce this contract (from types to assertions to explicit pre/post conditions).

Keep information on a need to know basis

Each function, class, and module in your code needs some information to do its job.

Give it the information it needs but no more.

Giving too much information couples parts of the code that should be independent, making them harder to test, debug, and reason about.

Objects in particular should "**encapsulate**" the information they contain quite jealously.

Make it run, make it right, make it fast – in that order

Only optimize the bottlenecks! This is a *data-driven* process.

> *The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.*
>
> *– Donald Knuth, The Art of Computer Programming*

Plan

Style Practices and Principles

Revision and Redesign: A Case Study

Building State Machines

Practice Activity: Representing Disjoint Sets

Recall the Types in Our Initial Design

```
data StateMachine =
  record StateMachine where
    currentState : State
    states       : List State
    transitions  : Dict TransitionId Transition
    events       : Dict EventType (Set TransitionId)
    stateActions : Dict (StateActionTiming, State) StateAction
    transActions : Dict (TransActionTiming, TransitionId) TransitionAction
```

Recall the Types in Our Initial Design

```
State = String
TransitionId = String
EventType = String

data Transition = record Transition where
                    source : State
                    target : State
                    name   : TransitionId     -- discretionary

data Event = record Event where
               type : EventType
               payload : payloadType type     -- payload type varies with event type

data StateActionTiming = Exit | Enter
data TransActionTiming = Before | During | After

StateAction = Event -> State -> StateMachine -> ()
TransitionAction = Event -> Transition -> StateMachine -> ()
```

Applications and Extensions

1. Pattern Matching

   A simple config file format

```
[section1]
key1 = value1
key2 = value2    # end of line comment
key3 = value3

[section2]
key4 = value4    # and so forth. Note blank line between sections
```

Applications and Extensions

1. Pattern Matching
   What are the states? Transitions? Actions? What are the payloads in the events?

Applications and Extensions

1. Pattern Matching

State	Transitions to...
Top Level	Section Start, Missing Section
Section Start	Awaiting Key, Missing Key
Awaiting Key	Awaiting Sep, Missing Sep
Awaiting Sep	Awaiting Val, Missing Val
Awaiting Val	Comment, Awaiting Key, Missing Val, Section End
...	...

Think of the input as a sequence of tokens. Events are dispatched with type based on the token with the token as payload.
We can handle this with the design as is, but notice what is less convenient here. An extension would help: .

Applications and Extensions

1. Pattern Matching

State	Transitions to...
Top Level	Section Start, Missing Section
Section Start	Awaiting Key, Missing Key
Awaiting Key	Awaiting Sep, Missing Sep
Awaiting Sep	Awaiting Val, Missing Val
Awaiting Val	Comment, Awaiting Key, Missing Val, Section End
...	...

Think of the input as a sequence of tokens. Events are dispatched with type based on the token with the token as payload.
We can handle this with the design as is, but notice what is less convenient here. An extension would help: **transition data**.

Applications and Extensions

1. Pattern Matching

```
State = String
TransitionId = String
EventType = String

data Transition a = record Transition where
                      source : State
                      target : State
                      name   : TransitionId
                      info   : a

SimpleTransition = Transition ()

TransitionAction = Event -> Transition a -> StateMachine -> ()
```

(Adjusting the code)

Applications and Extensions

2. Finite State Markov Chains

   As we saw last time, Markov chain machines benefit from both transition data and *selectors*. How should the selectors be configured? Stored? For instance,

```
data StateMachine =
  record StateMachine where
    currentState : State
    states       : List State
    transitions  : Dict TransitionId Transition
    events       : Dict EventType (Set TransitionId)
    selectors    : Dict EventType Selector
    stateActions : Dict (StateActionTiming, State) StateAction
    transActions : Dict (TransActionTiming, TransitionId) TransitionAction

Selector = EventType -> Set (Transition a) -> Maybe (Transition a)  -- not pure
```

   Note how we've implicitly handled guards here. Is that a good idea?

   How does our code change?

Applications and Extensions

2. Finite State Markov Chains
   As we saw last time, Markov chain machines benefit from both transition data and *selectors*. How should the selectors be configured? Stored?

Applications and Extensions

3. User Interface Control

   Consider a very simple registration form with an email field, a password field, and a submit button. Code to manage this in an *ad hoc* manner quickly becomes complex. We have a submit button (enabled/disabled), one or more error labels (invisible/visible), and a success label (invisible/visible).

   A state machine allows us to represent much of this **code as data**!

   What are the states, transitions, and actions? Events are derived from the user, what do they look like? How are *guards* used here?

Code Smell

We use **code smell** to mean features of code, concrete or impressionistic, that hints at or suggests weakness of design, susceptibility to complexity, and vulnerability to bugs. These warnings are worth taking seriously as they can indicate deeper problems.

An example include code that is not DRY. What might be code smell in our definition of the `transitions` table?

Code Smell

Notice the repetition of the id. It rankles but is useful. Yet this is code smell. Do we need to look up transitions by id? When? Why? Having written dispatch, how should we organize these?

And notice that changing the organization also allows us to support a more flexible design: implicit or functionally defined transitions?

Code Smell

Notice the repetition of the id. It rankles but is useful. Yet this is code smell. Do we need to look up transitions by id? When? Why? Having written dispatch, how should we organize these?

And notice that changing the organization also allows us to support a more flexible design: implicit or functionally defined transitions?

For example: checking for balanced (and possibly interleaved) delimiters, e.g., LaTeX source.

What are the states and transitions that we want in this case? Can they be defined *a priori*? Should state be `Either String Int`? Pros and cons?

Plan

Style Practices and Principles

Revision and Redesign: A Case Study

Building State Machines

Practice Activity: Representing Disjoint Sets

The Builder Pattern

What should our interface be for the `StateMachineBuilder`?

The Builder Pattern

What should our interface be for the `StateMachineBuilder`?

One desirable feature would be a **fluent** interface, where each builder method (except `build`) returns the builder so that we can chain actions. We specify a state machine by defining states, transitions, events, actions, along with guards and selectors if appropriate.

```
machine = (
    StateMachineBuilder()
    .add_states('a', 'b', 'c', 'd')
    .add_transition('a', 'b')       # Name defaults to 'a -> b'
    .add_transition('c', 'd', name='foo')
    .add_action('enter', 'c', on_c_func)
        #...
    .build()
)
```

or in R

```
machine = StateMachineBuilder() |>
            add_states('a', 'b', 'c', 'd') |>
            add_transition('a', 'b', name='ok') |>       # Name defaults to 'a -> b
            add_transition('c', 'd', name='foo') |>
            add_action('enter', 'c', on_c_func)  |>
                #... |>
            build()
```

The `build` method ***validates the specification*** and ***creates the `StateMachine`*** object.

Plan

Style Practices and Principles

Revision and Redesign: A Case Study

Building State Machines

Practice Activity: Representing Disjoint Sets

The Union-Find Algorithm

In this activity, we consider an algorithm that represents and manipulates disjoint sets, called the **union-find algorithm**.

Here, we are considering a set of objects; these can be anything such as computers in a network, pages on the internet, pixels in an image, or locations in a maze. We are looking at two operations on these objects:

1. A "union" operation: connect two objects together (and consequently any objects they are each connected to).
2. A "find" query: is there a path connecting two specified objects?

The Union-Find Algorithm: Examples

We have a collection of *atoms* $a, b, c, \ldots$. These can be variables, expressions, types, or other terms. We posit a sequence of equalities, e.g., $a = b$ and $b = c$, and we query whether other terms are equal, e.g., is $a = c$?

A posited equality is a "union" operation. It connects the two atoms in the equality and by transitivity all else A query is a "find" operation. Are two atoms equal?

The Union-Find Algorithm: Examples

Consider an undirected graph or an image. Two nodes in the graph are connected if there is an edge between them; two pixels in the image are connected if they are adjacent and both have value bigger than some specified threshold.

What do the union and find operations mean in terms of the connected components of the graph and image?

The Union-Find Algorithm: Examples

```
0 1 2 3 4 5 6 7 8 9                 objects
{0 1} 2 {4 5} 6 7 {3 8 9}           disjoint sets of objects
{3 8 9}                             find query: are 3 and 9 "connected"?
{0 1 3 8 9} 2 {4 5} 6 7             union command: connect 0 and 3
```

The Idea

Whatever our objects it will be convenient to label them from 0 to n-1, where n is the total number of objects.

Our goal is to find a data structure that lets us efficiently run union commands and find queries, with very large sets of objects and very large numbers of commands/queries.

We will look at implementations of several union-find algorithms. In each case, we will build a **forest** of trees.

The Interface

We will create a class that implementsaninterface for disjoint sets (union-find). The interface looks like

```
-- Create a collection of disjoint sets
makeSets       : List object -> DisjointSets object

-- Given an object in the sets, return its representative
representative : DisjointSets object -> object -> object

-- Are two objects connected, i.e., in the same component?
connected?     : DisjointSets object -> object -> object -> Bool

-- Join two objects, and all the objects connected to them.
union          : DisjointSets object -> object -> object -> DisjointSets object
```

Traditionally `connected?` is called `find`, which seems meh as a name.

Start by constructing a few tests for the `union` command and the `connected` query. You will use these below.

The Algorithms: A Sketch

Eager. The files EagerObjectSets.* implement a simple data structure that is fast for find queries but slow for union commands.

We represent our forest with an array `root` of size `n`. Each object represents a node in the forest, and each node is either by itself or part of a tree with one root node and one or more children.

So this means that `root[obj_i] == root[obj_j]` if object `obj_i` and `obj_j` are "connected." We take the value of this to be the root of the tree to which the objects belong.

Consider the following series of commands and the associated forests. What does the root array look like at each stage? If we continue connecting nodes until everything is connected, what will the result look like?

The Algorithms: A Sketch

```
0 1 2 3 4 5 6 7 8 9
```

```
0 1 2 4 5 6 7 8 9          union(3,4)      root[3] = 4 = root[4]
      |
      3
```

```
0 1 2 5 6 7 8  9           union(4,9)      root[3] = root[4] = root[9] = 9
              / \
              4  3
```

```
7 1 2 5 6 8  9             union(0,7)
|           / \
0           4  3
```

```
7 1 2 6 8  5               union(3,5)
|         /|\
0        4 9 3
```

```
                           connected(9,5) => true
```

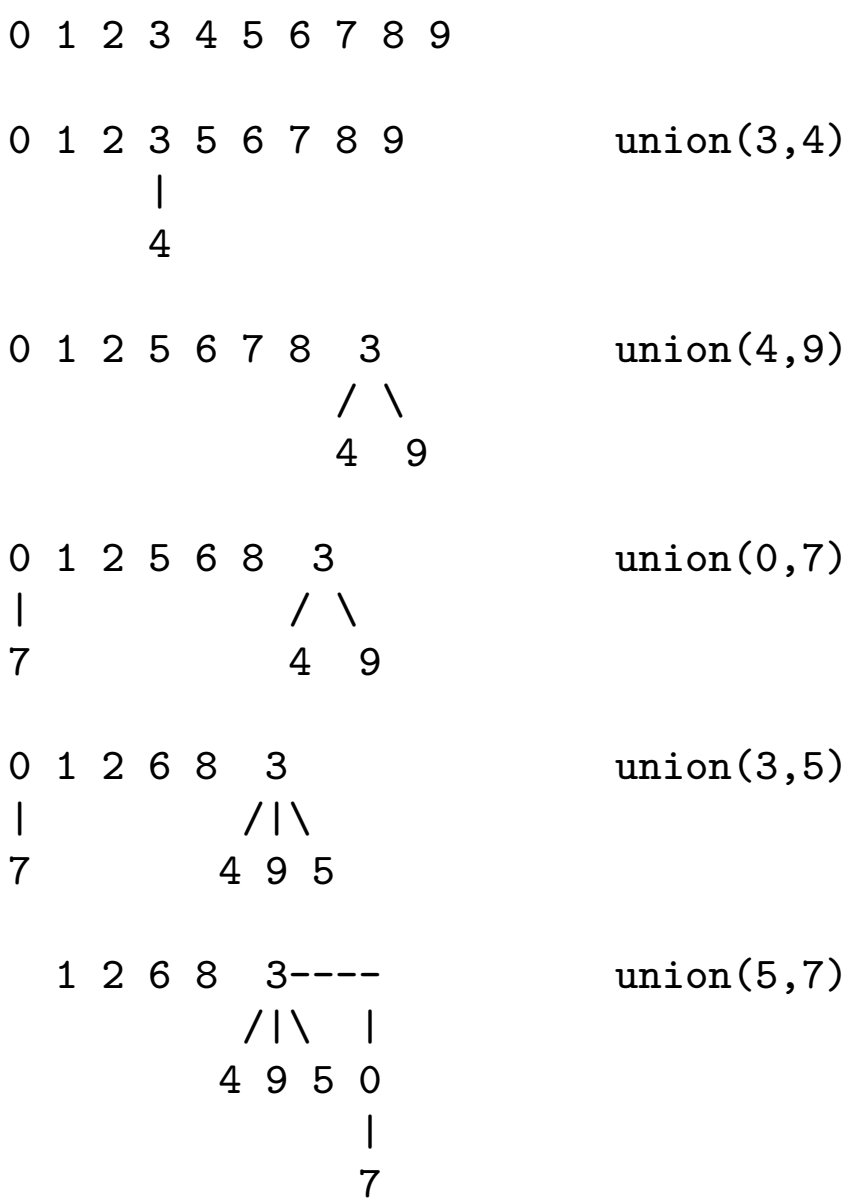
The Algorithms: A Sketch

Lazy. The eager find method leads tqo short flat trees, where each object is connected to the root node of the tree. The lazy union approach, illustrated partially in the files LazyObjectSets.* take a different approach.

Instead of keeping an array `root`, we keep an array `parent` where `parent[obj_i]` is the parent of the tree in which object `obj_i` belongs. Implement the `union` method and the utility method `representative`. Run your tests.

This is still slow. If the trees get too deep, find gets quite poor performance.

The Algorithms: A Sketch

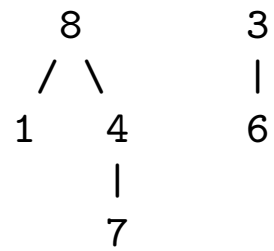


The Algorithms: A Sketch

Fast. We will improve lazy union in two ways:

1. Weighted unions. Each tree keeps track of its size. When joining two trees, connect the smaller tree below the larger one.

```
union(3, 7)  :  without weighting, connect 8 to 3
             :  with weighting, connect 3 to 8
```



   This avoid deep trees.

2. Path compression. After computing `root(obj_i)` in lazy union, set the parent of each node examined in the process to `root(obj_i)`.

```
root(9) examines objects 9, 6, 3, 1, 0

connect nodes 9, 6, 3, and 1 (and their subtrees)
directly to node 0.
```

The Algorithms: A Sketch

With these two improvements, this becomes quite efficient. With m union and find operation on n object, performance is O(n + m lg* n), where lg* n is the number of times lg must be applied to n before the value is 1 or less.

Implement union and connected in the file FastObjectSets.*. Run your tests.

Hint: A fast one pass approach modifies the `representative` function to make every other node in the path to point to its grandparent.

The Task

Implement the missing parts of each file. The Eager files are complete for your reference. Python and R versions available.

The End