Simple Made Easy

Rich Hickey

Simplicity is prerequisite for reliability

Edsger W. Dijkstra

Word Origins

- Simple

  sim- plex

  one fold/braid

  vs complex

- Easy

  ease < aise < adjacens

  lie near

  vs hard

Simple

- One fold/braid
  - One role
  - One task
  - One concept
  - One dimension

- But not
  - One instance
  - One operation
- About lack of interleaving, not cardinality
- *Objective*

Easy

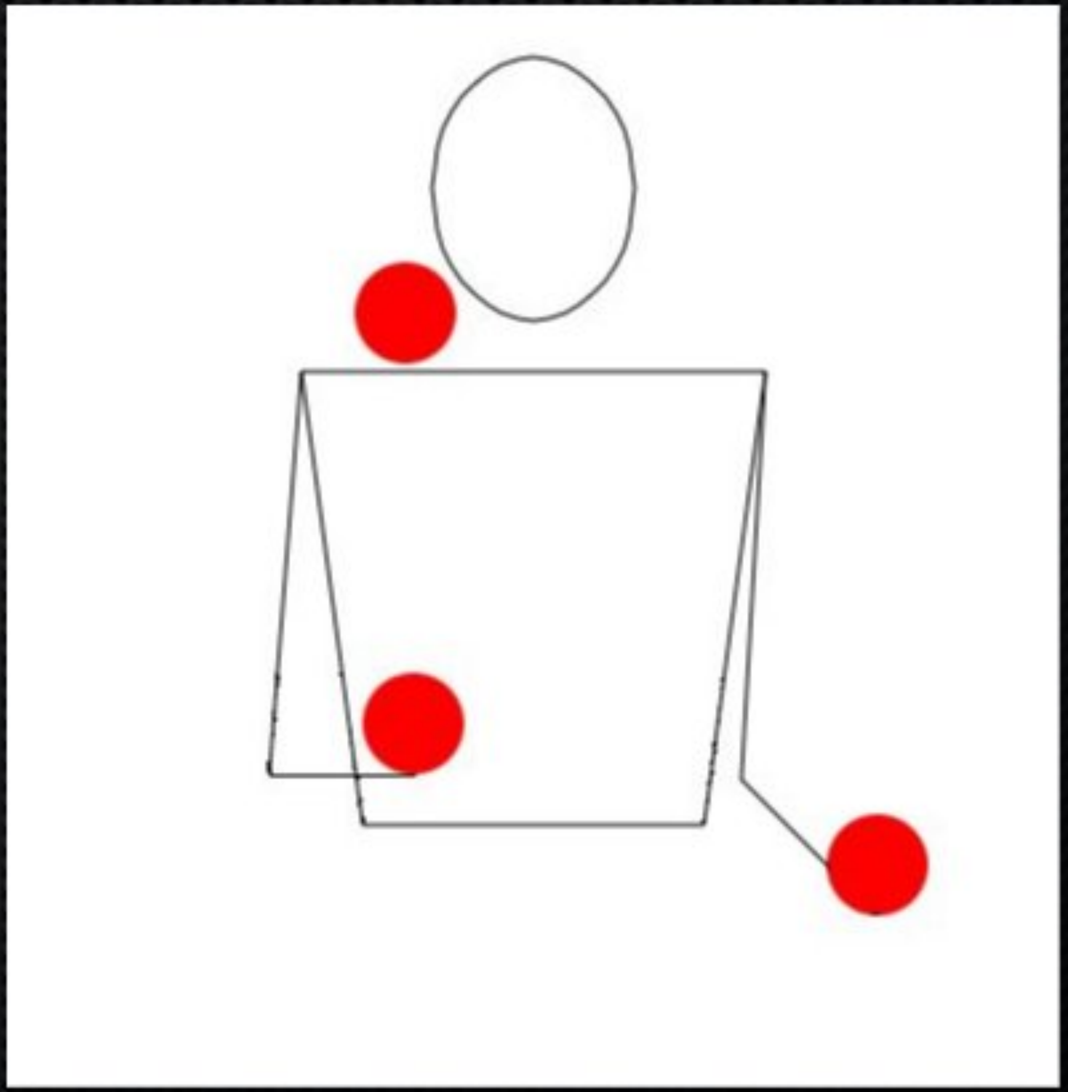
- Near, at hand
  - on our hard drive, in our tool set, IDE, apt get, gem install...
- Near to our understanding/skill set
  - familiar
- Near our capabilities
- Easy is *relative*

Construct vs Artifact

- We focus on experience of use of construct
    - programmer convenience
    - programmer replaceability
- Rather than the long term results of use
    - software quality, correctness
    - maintenance, change
- We must assess constructs by their artifacts

Limits

- We can only hope to make reliable those things we can understand
- We can only consider a few things at a time
- Intertwined things must be considered together
- Complexity undermines understanding



Change

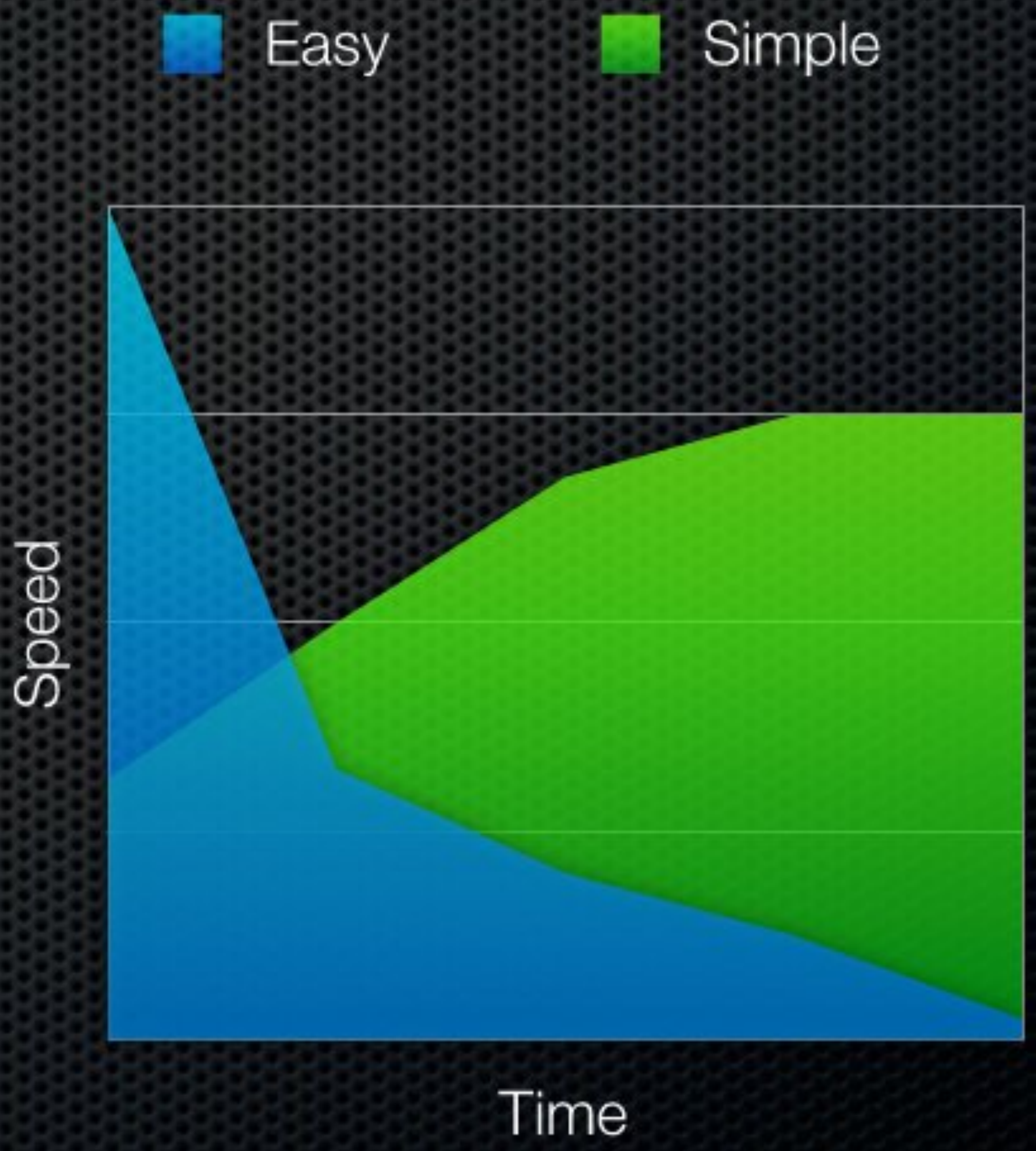
- Changes to software require analysis and decisions
- What will be impacted?
- Where do changes need to be made?
- Your ability to reason about your program is critical to changing it without fear
    - Not talking about proof, just informal reasoning

Debugging

- What’s true of every bug in the field?
- It has passed the type checker
  - and all the tests
- Your ability to reason about your program is critical to debugging

Development Speed

- Emphasizing ease gives early speed
- Ignoring complexity will slow you down over the long haul
- On throwaway or trivial projects, nothing much matters



Easy Yet Complex?

- Many complicating constructs are
  - Succinctly described
  - Familiar
  - Available
  - Easy to use
- What matters is the complexity they *yield*
  - Any such complexity is *incidental*

Benefits of Simplicity

- Ease understanding
- Ease of change
- Easier debugging
- Flexibility
  - policy
  - location etc

Making Things Easy

- Bring to hand by installing
  - getting approved for use
- Become familiar by learning, trying
- But mental capability?
  - not going to move very far
  - make things near by simplifying them

Parens are Hard!

- Not at hand for most
- Nor familiar
- But are they simple?
- Not in CL/Scheme
  - overloaded for calls and grouping
  - for those that bothered trying, this is a valid complexity complaint

- Adding a data structure for grouping, e.g. vectors, makes each simpler
  - minimal effort can then make them easy too

LISP programmers know the value of everything and the cost of nothing.

Alan Perlis

What’s in *your* Toolkit?

Complexity	Simplicity
State, Objects	Values
Methods	Functions, Namespaces
vars	Managed refs
Inheritance, switch, matching	Polymorphism a la carte
Syntax	Data
Imperative loops, fold	Set functions
Actors	Queues
ORM	Declarative data manipulation
Conditionals	Rules
Inconsistency	Consistency

Complect

- To interleave, entwine, braid
  - archaic
- Don’t do it!
  - Complecting things is the source of complexity
- Best to avoid in the first place

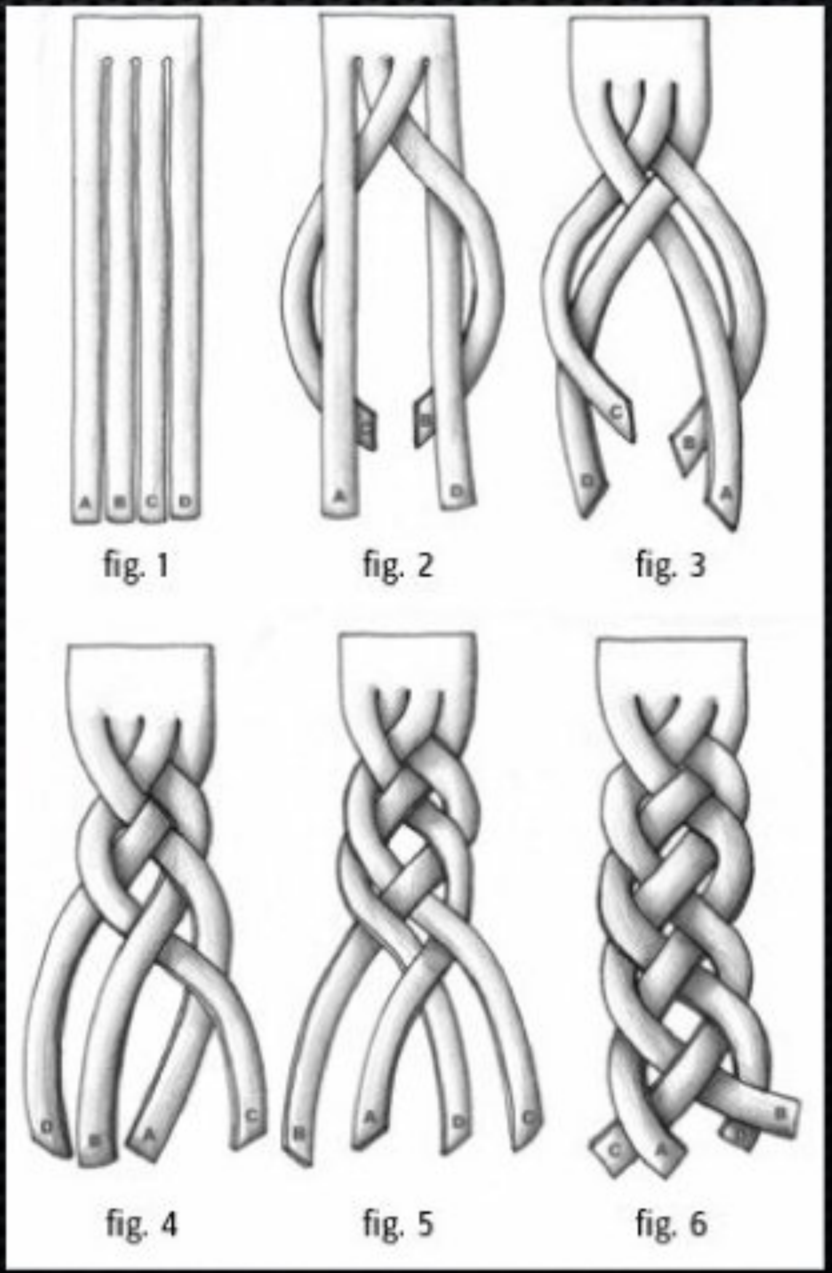


fig. 1 fig. 2 fig. 3

fig. 4 fig. 5 fig. 6

Compose

- To place together
- Composing simple components is the key to robust software

Modularity and Simplicity

Modularity and Simplicity

Modularity and Simplicity

- Partitioning and stratification don't imply simplicity
  - but *are* enabled by it
- Don’t be fooled by code organization

State is Never Simple

- Complects value and time
- It *is* easy, in the at-hand and familiar senses
- Interweaves everything that touches it, directly or indirectly
  - Not mitigated by modules, encapsulation
- Note - this has nothing to do with asynchrony

Not all refs/vars are Equal

- None make state simple
- All warn of state, help reduce it
- Clojure and Haskell refs *compose* value and time
  - Allow you to extract a simple value
  - Provide abstractions of time
- Does your var do that?

The Complexity Toolkit

Construct	Complects
State	Everything that touches it
Objects	State, identity, value
Methods	Function and state, namespaces
Syntax	Meaning, order
Inheritance	Types
Switch/matching	Multiple who/what pairs
var(iable)s	Value, time
Imperative loops, fold	what/how
Actors	what/who
ORM	OMG
Conditionals	Why, rest of program

The Simplicity Toolkit

Construct	Get it via...
Values	final, persistent collections
Functions	a.k.a. stateless methods
Namespaces	language support
Data	Maps, arrays, sets, XML, JSON etc
Polymorphism a la carte	Protocols, type classes
Managed refs	Clojure/Haskell refs
Set functions	Libraries
Queues	Libraries
Declarative data manipulation	SQL/LINQ/Datalog
Rules	Libraries, Prolog
Consistency	Transactions, values

Environmental Complexity

- Resources, e.g. memory, CPU
- *Inherent* complexity in implementation space
  - All components contend for them
- Segmentation
  - waste
- Individual policies don't compose
  - just make things more complex

Programming, when stripped of all its circumstantial irrelevancies, boils down to no more and no less than very effective thinking so as to avoid unmastered complexity, to very vigorous separation of your many different concerns.

Edsger W. Dijkstra

Abstraction for Simplicity

- Abstract
  - drawn away
- vs Abstraction as complexity *hiding*
- Who, What, When, Where, Why and How
- I don’t know, I don’t want to know

What

- Operations
- Form abstractions from related sets of functions
  - *Small* sets
- Represent with polymorphism constructs
- Specify inputs, outputs, semantics
  - Use only values and other abstractions
- Don’t complect with:
  - How

Who

- Entities implementing abstractions
- Build from subcomponents direct-injection style
  - Pursue many subcomponents
    - e.g. policy
- Don’t complect with:
  - component details
  - other entities

How

- Implementing logic
- Connect to abstractions and entities via polymorphism constructs
- Prefer abstractions that don’t dictate *how*
  - Declarative tools
- Don’t complect with:
  - anything

When, Where

- Strenuously avoid complecting these with anything in the design
- Can seep in via directly connected objects
  - Use queues

Why

- The policy and rules of the application
- Often strewn everywhere
  - in conditionals
  - complected with control flow etc
- Explore rules and declarative logic systems

Information *is* Simple

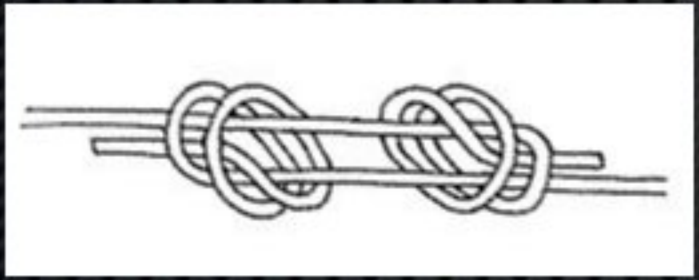
- Don’t ruin it
- By hiding it behind a micro-language
    - i.e. a class with information-specific methods
    - thwarts generic data composition
    - ties logic to representation du jour
- Represent data as data

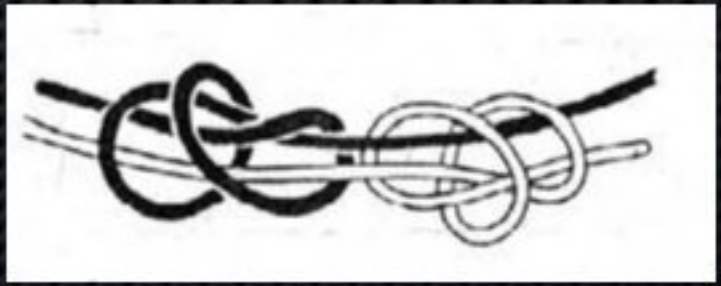
Simplicity is not an objective in art, but one achieves simplicity despite one's self by entering into the real sense of things

Constantin Brancusi

Simplifying

- Identifying individual threads/roles/dimensions
- Following through the user story/code
- Disentangling





Simplicity is a Choice

- Requires vigilance, sensibilities and care
- Your sensibilities equating simplicity with ease and familiarity are wrong
  - Develop sensibilities around entanglement
- Your 'reliability' tools (testing, refactoring, type systems) don't care
  - and are quite peripheral to producing good software

Simplicity Made Easy

- Choose simple constructs over complexity-generating constructs
  - It’s the artifacts, not the authoring
- Create abstractions with simplicity as a basis
- Simplify the problem space before you start
- Simplicity often means making more things, not fewer
- Reap the benefits!

Simplicity is the ultimate sophistication.

Leonardo da Vinci