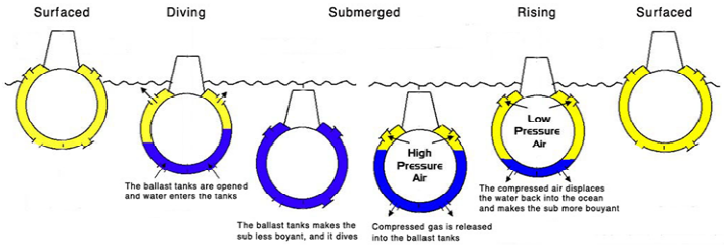
Malina SOFTWARE CORP.

Theory

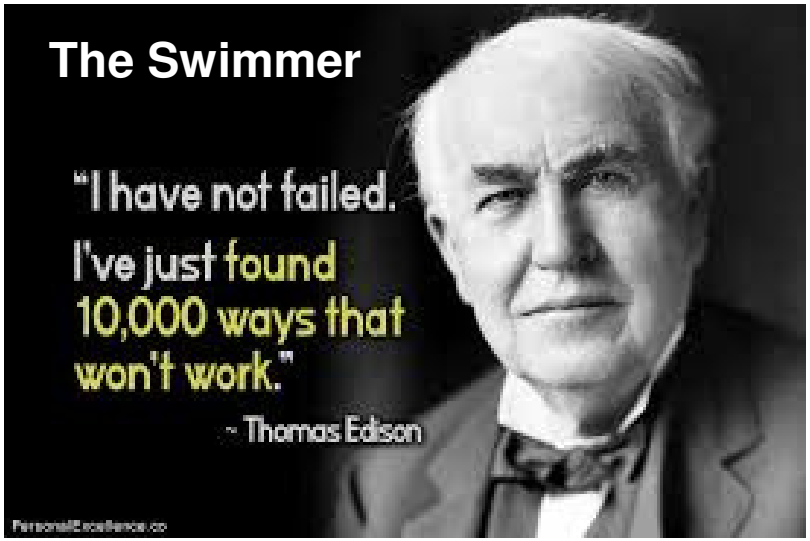


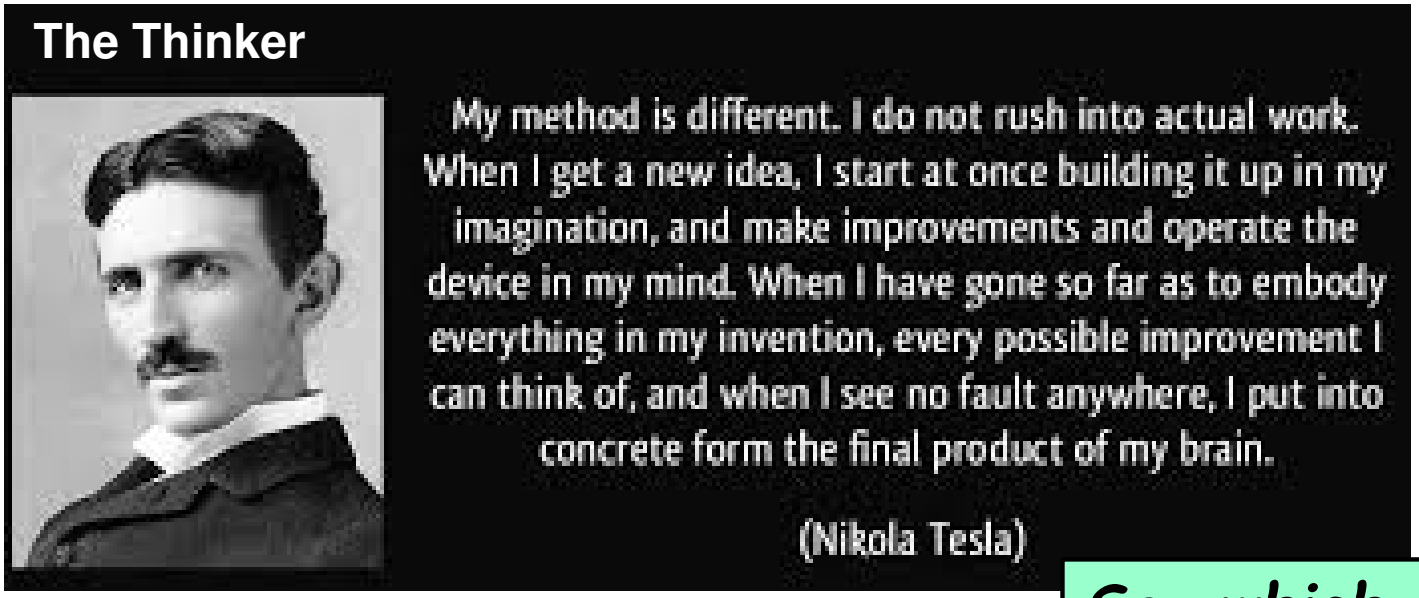
Think or Swim? Ramblings On Engineering Methodology

Bran Selić

Practice

Thinkers and Swimmers





So, which is better?

A Note on Edison's Approach

- **Thomas Edison [1914]:**

  "When I want to discover something, I begin by reading up everything that has been done along that line in the past — that's what all these books in the library are for. I see what has been accomplished at great labor and expense in the past. I gather data of many thousands of experiments as a starting point, and then I make thousands more."

An aside:
***A Culture of Persistent Reinvention* –**
Software practitioners rarely look at prior work beyond their immediate environment

"The [engineer] should be equipped with knowledge of many branches of study and varied kinds of learning, for it is by his judgment that all work done by the other arts is put to test. This knowledge is the child of practice and theory."

- Vitruvius

On Architecture, Book I (1st Century BC)

"The only genuine knowledge is the result of direct experience"

- Mao Zedong

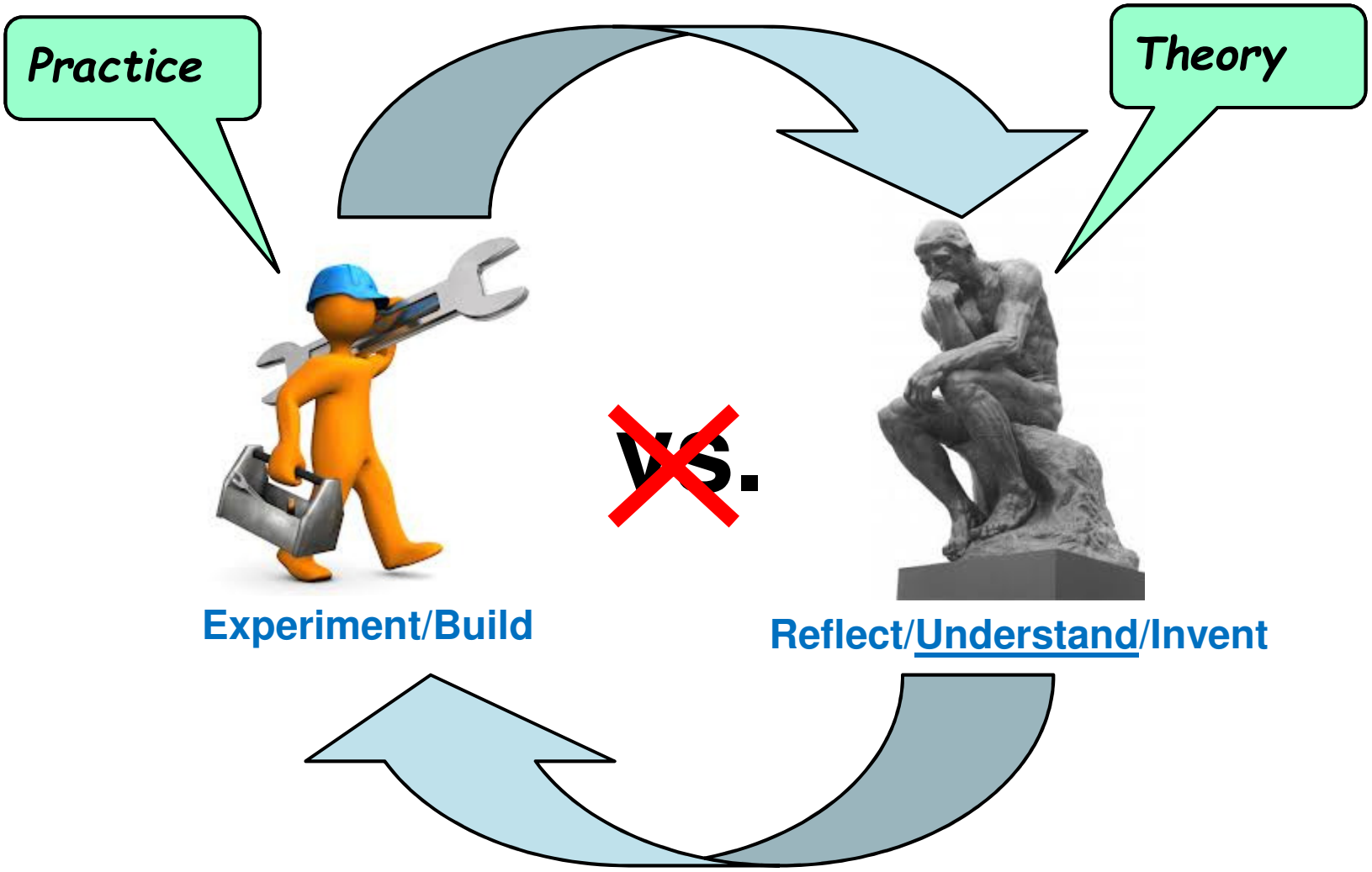
Think OR Swim?

- A *false dilemma*. These are NOT mutually exclusive approaches ⇒ Think AND Swim
- One of the most effective means of learning (and *understanding*!) is doing (i.e., practice)
  - Trial and error
- But, also: *"You can always find a better solution to a problem if you think about it long enough"* [Ernst Munter, Bell-Northern Research Engineer]
  - Reflecting on experience to *build understanding*

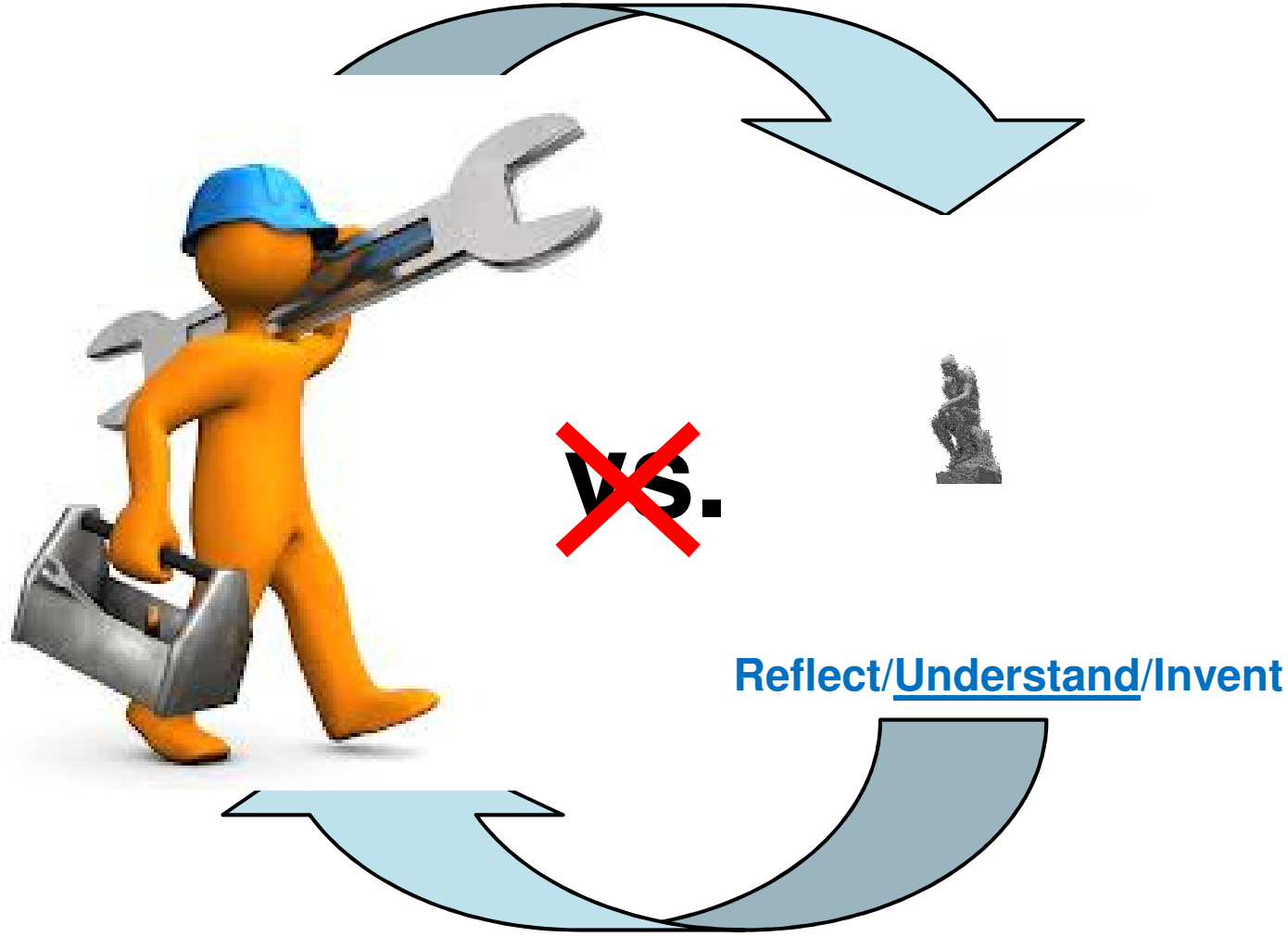
"The world if filled with Knowledge; it is almost empty of Understanding. For let me tell you, Knowledge is of the head, Understanding of the heart."

-Louis Sullivan, Architect

The Technical Development Process

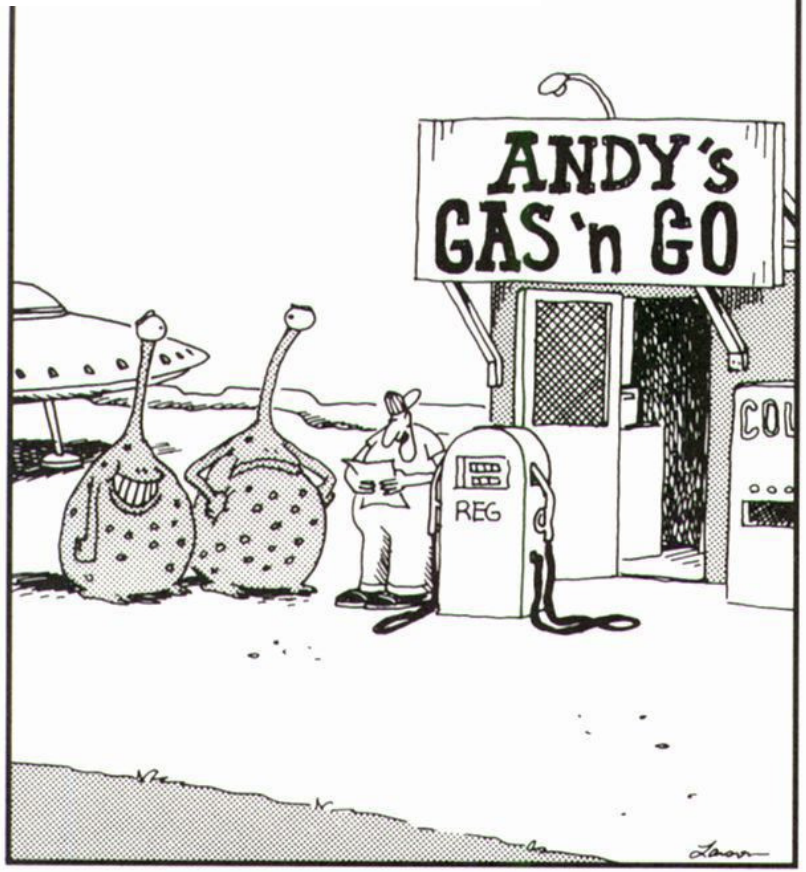


An Unfortunate Misinterpretation of "Agile"



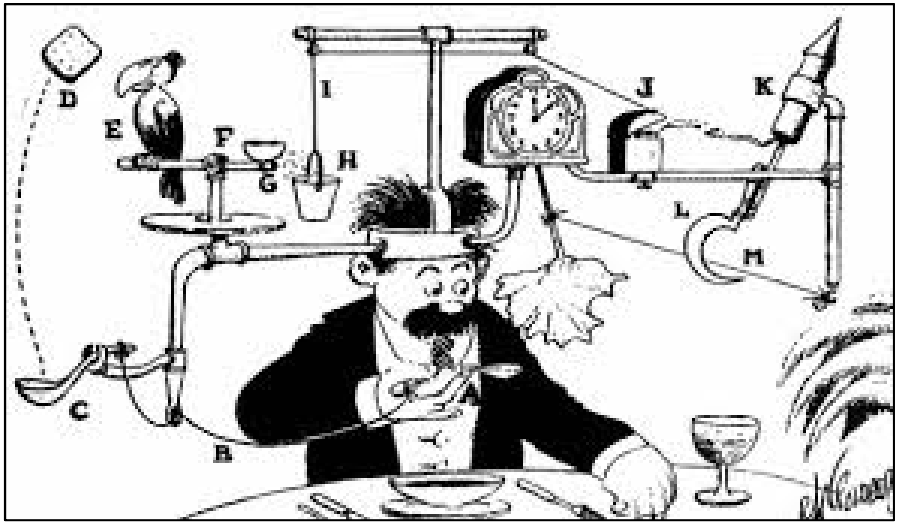
Unpleasant Outcomes of Mismanaged Trial and Error

Wrong System

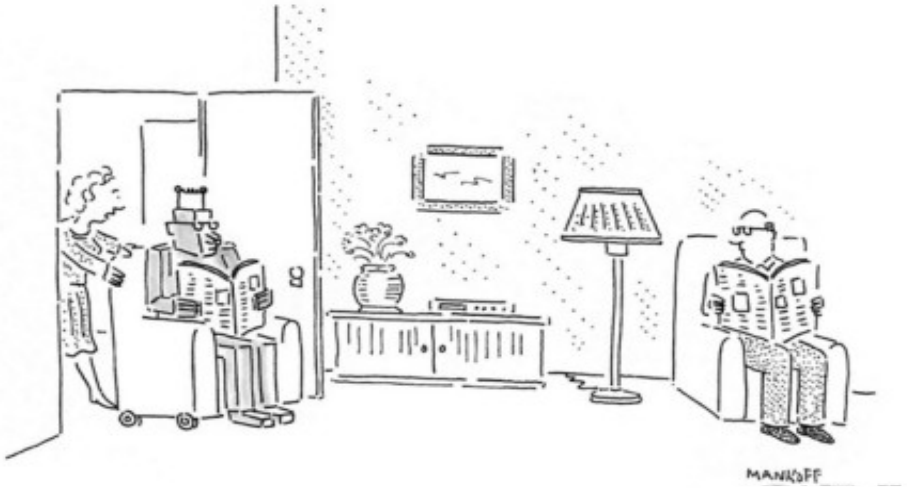


"Shoot! You not only got the wrong planet, you got the wrong *solar* system. ... I mean, a wrong planet I can understand—but a whole solar system?"

Chewing-gum-and-duct tape System



Problems with Trial and Error

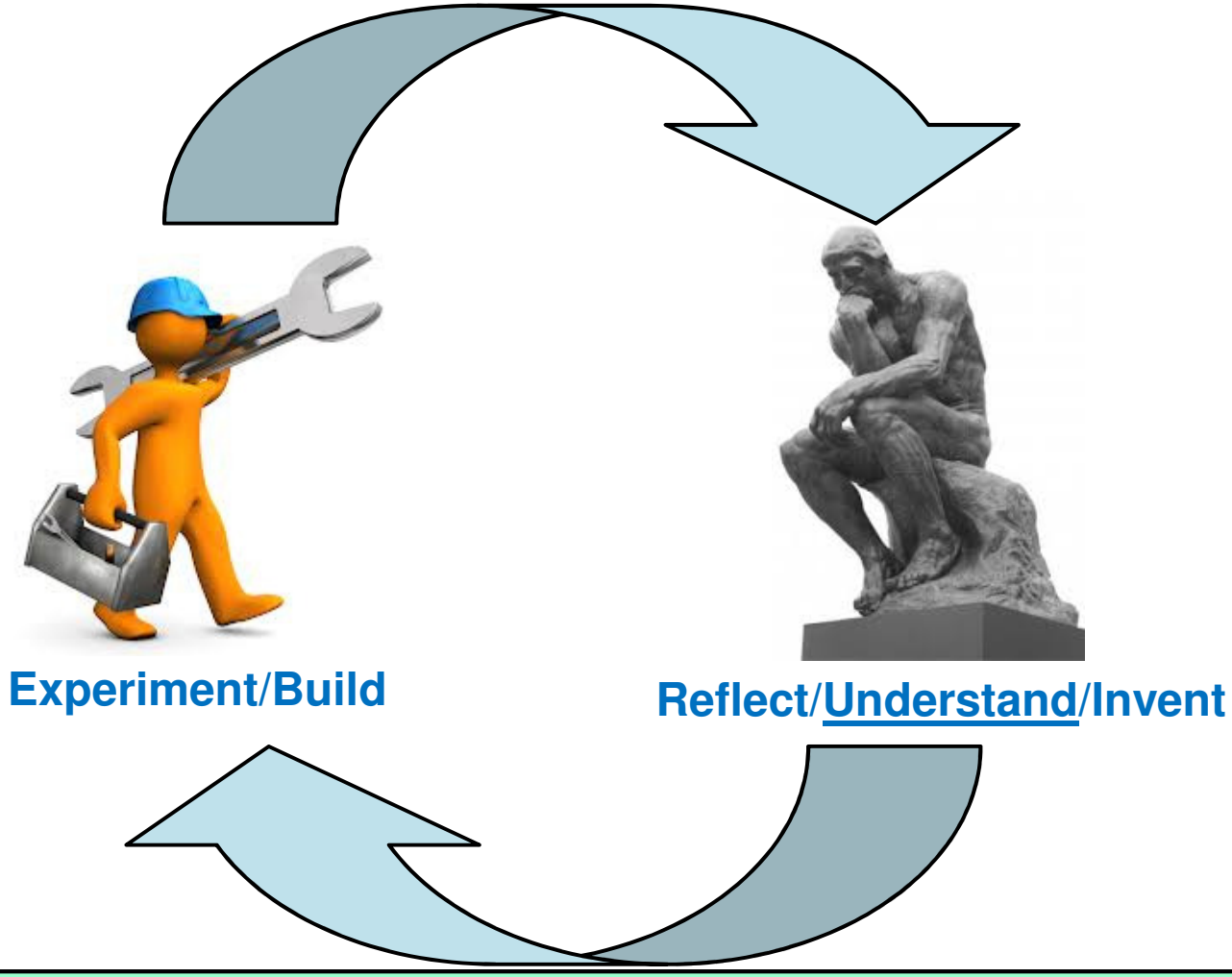


"Don't panic. It's only a prototype."

- Only practical if the effort invested in the "trial" part is small enough that it can be easily discarded
  - …otherwise there is a tendency to patch up an unsatisfactory solution ("hacking away")

⇒ *The ability of tools to support "lightweight" prototyping (design space exploration) is crucial*

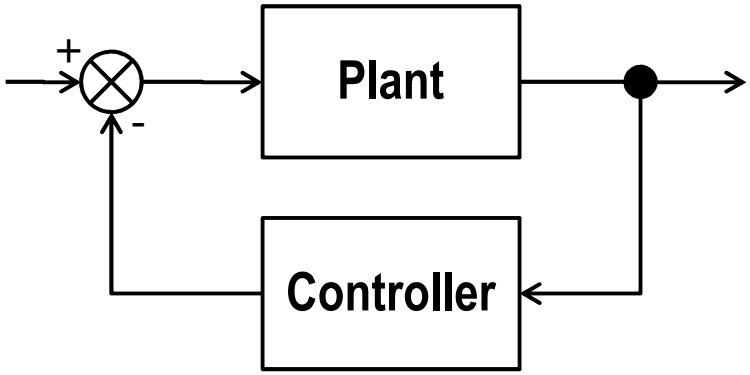
How Do We Achieve Understanding?



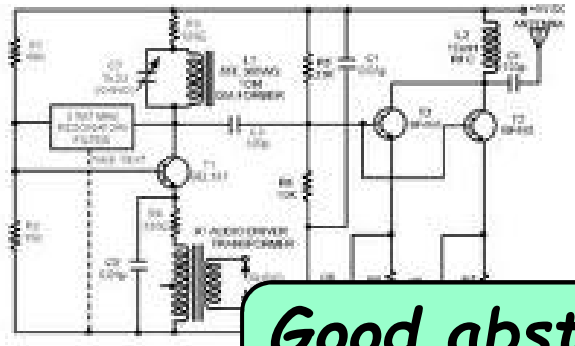
***Claim*: Understanding invariably starts with Experience followed by Abstraction**

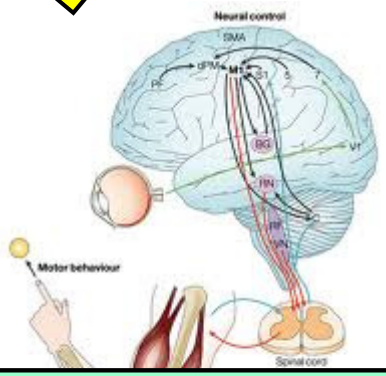
Abstraction: The Path to Understanding

- The negative feedback control pattern:



A good abstraction makes the implicit explicit [Bergson]





Good abstractions are the product of a fertile imagination [Selic]

The Big Problem with Abstraction

- **Alfred North Whitehead [*Modes of Thought*]:**

  "The topic of every science is an abstraction from the full concrete happenings of nature. But every abstraction neglects the influx of the factors omitted into the factors retained."

Yikes!! I forgot about the wind and resonance

Fortunately, in engineering, such events are infrequent since the models used in these disciplines are fairly accurate representations of reality

But, When It Comes to Software…

- The day the phones stopped…
  - January 1990; AT&T Long Distance Network crash

```
. . .;
switch (...) {
      case a:...;
           break;
      case b:...;
           break;
      case m:...;
      case n:...;
           . . .
};
```

Recovery time: 1 day

Cost: 100 millions of $'s

"The devil is in the details":

- *Potentially <u>any line of code can have an enormous impact</u>*
- *And, there are lots and lots of lines of code*

#@$!! I forgot the "break" statement…

So, How Should We Abstract Software?

- Verifying software at the level of individual programming language statements rarely scales
- The only practical way to verify complex software systems is to verify abstract representations (i.e., models) of that software
- This requires accurate models of software
- But, how can we create abstract representations of systems in which even the minutest of details can have profound consequences?

We need a new generation of modeling languages that are:

- *Less susceptible to "minor" flaws in logic*
- *Based on proven formalisms that facilitate computer-based analysis*