

With Economy and Elegance

Software Engineering Reclaimed

Kevlin Henney

kevin@curbralan.com

Conflict without Cause?

engineering the branch of science and technology concerned with the design, building, and use of engines, machines, and structures.

The New Oxford Dictionary of English

Structural engineering is the science and art of designing and making, with economy and elegance, buildings, bridges, frameworks, and other similar structures so that they can safely resist the forces to which they may be subjected.

The Institution of Structural Engineers

Beyond Metaphor

- To understand something abstract and aphysical we look to the concrete and familiar
 - ◆ We use metaphors as guides... but metaphors have limits, and they provide guidance not identity
- In characterising software development a number of metaphors have been used
 - ◆ Software development as civil engineering, software development as mathematics, software development as gardening, etc.

Professionalism

- Many of the metaphors for software development appeal to professionalism
 - ◆ Competence and skill in an occupation and the relationship to a discipline of knowledge
 - ◆ For software, it is the discipline of knowledge that is in a continual state of flux and growth
- The use of one metaphor does not necessarily exclude the use of others
 - ◆ But in searching for identity we need to look beyond caricatures and preconceptions

Engineering or Craft?

- Engineering and craft have a number of things in common
 - ◆ They share the notions of making things, common bodies of practice and a professional discipline
- What differentiates engineering from craft is not maturity, headcount or industrialisation
 - ◆ Scale, complexity and nature make the difference
 - ◆ Software engineering involves craft, but it is a discipline of technology that is empirical with theoretical underpinnings

Understanding Engineering

- The mistake has not been in treating software development as a kind of engineering...
 - ◆ But in assuming that it was a form of physical engineering
 - ◆ But in assuming that *engineering* was synonymous with *plan driven*
 - ◆ But in assuming a caricature of other engineering disciplines to compare against and mimic
 - ◆ But in assuming that there was no art, craft or sense of aesthetics in engineering

Understanding Family

- Software engineering is a kind of informational engineering, not a kind of physical engineering
 - ◆ Therefore, software engineering is a cousin, not a sibling (and certainly not a clone), of the other frequently cited forms of engineering — cousins are still family, but they have less in common
- This means that much of what has passed for *software engineering* until now is pastiche
 - ◆ Choose the wrong map, and you will end up lost and heading in the wrong direction

Emerging Discipline

- Many known solutions and solution approaches form a stable base to build on
 - ◆ The classic topics of computer science
 - ◆ Using commoditised tools and libraries
 - ◆ Patterns for capturing and communicating software architecture and development experience
- Clearer thinking about how people work
 - ◆ Empirical and lean development processes
 - ◆ Focusing on interconnected systems, not just parts

Emerging Professionalism

- A clearer sense of responsibility can be found in many approaches advocated today
 - ◆ Programmers are responsible for the quality of their code, which includes testing, clean code, reasoned and reasonable choice of implementation, appropriate runtime characteristics, etc.
- It is easier to form developer communities that communicate norms, practices and experiences
 - ◆ From discussion groups to Open Source projects
 - ◆ From informal group meetings to conferences

Shared Understanding?

- The challenge is to make sure that what is already known and won is known and won
 - ◆ Software development is a diverse and fast moving field, and there is already a lot to know — and not everyone wants to know
 - ◆ Many previous attempts at capturing and communicating expertise have been too rigid and imposed in their approach
 - ◆ A mixture of advocacy, osmosis, openness, respect and leading by example is needed

Economy and Elegance

- Economy and elegance are not arbitrary or whimsical concerns
 - ◆ Effective software development involves more than just a focus on functional and operational behaviour – concern for the developmental qualities of a system supports other objectives
 - ◆ Recognising that these qualities have value is part of professionalism – look beyond surface effects
- So, what does this entail for the detail of code?

With Economy

Continuing existence or cessation of existence: those are the scenarios. Is it more empowering mentally to work towards an accommodation of the downsizings and negative outcomes of adversarial circumstance, or would it be a greater enhancement of the bottom line to move forwards to a challenge to our current difficulties, and, by making a commitment to opposition, to effect their demise?

Tom Burton, *Long Words Bother Me*

*To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them?*

William Shakespeare, *Hamlet*

Sufficiency

- Additional options and features can lead to confusion rather than clarity
 - ◆ Overachieving, speculative design is weaker and more complex, not stronger and simpler
- Focus on what is needed, and use what you don't know to help drive the development
 - ◆ Look for ways to elicit feedback
- Clarity, ease of use and performance are often achieved by reducing both clutter and hedging

Too Much

```
interface Iterator
{
    boolean set_to_first_element();
    boolean set_to_next_element();
    boolean set_to_next_nth_element(in unsigned long n) raises(...);
    boolean retrieve_element(out any element) raises(...);
    boolean retrieve_element_set_to_next(out any element, out boolean more) raises(...);
    boolean retrieve_next_n_elements(in unsigned long n, out AnySequence result, out boolean more) raises(...);
    boolean not_equal_retrieve_element_set_to_next(in Iterator test, out any element) raises(...);
    void remove_element() raises(...);
    boolean remove_element_set_to_next() raises(...);
    boolean remove_next_n_elements(in unsigned long n, out unsigned long actual_number) raises(...);
    boolean not_equal_remove_element_set_to_next(in Iterator test) raises(...);
    void replace_element(in any element) raises(...);
    boolean replace_element_set_to_next(in any element) raises(...);
    boolean replace_next_n_elements(in AnySequence elements, out unsigned long actual_number) raises(...);
    boolean not_equal_replace_element_set_to_next(in Iterator test, in any element) raises(...);
    boolean add_element_set_iterator(in any element) raises(...);
    boolean add_n_elements_set_iterator(in AnySequence elements, out unsigned long actual_number) raises(...);
    void invalidate();
    boolean is_valid();
    boolean is_in_between();
    boolean is_for(in Collection collector);
    boolean is_const();
    boolean is_equal(in Iterator test) raises(...);
    Iterator clone();
    void assign(in Iterator from_where) raises(...);
    void destroy();
};
```

Enough

```
interface BindingIterator
{
    boolean next_one(out Binding result);
    boolean next_n(in unsigned long how_many, out BindingList result);
    void destroy();
};
```

Essential versus Accidental Complexity

- Complexity comes from many sources, not all of which are necessary
 - ◆ Sometimes the problem domain is complex, which may dictate complexity in the solution
 - ◆ Sometimes it is misunderstanding or insufficient skill that creates complexity
 - ◆ Sometimes it is cleverness that creates the complexity — code by über-programmers is often based on speculative generality and gratuitous use of advanced techniques and language features

Insufficient Skill

```
public class RecentlyUsedList
{
    public RecentlyUsedList()
    {
        list = new List<string>();
    }
    public void Add(string newItem)
    {
        if (list.Contains(newItem))
        {
            int position = list.IndexOf(newItem);
            string existingItem = list[position];
            list.RemoveAt(position);
            list.Insert(0, existingItem);
        }
        else
        {
            list.Insert(0, newItem);
        }
    }
    ...
}
```

```
...
public int Count
{
    get
    {
        int size = list.Count;
        return size;
    }
}
public string this[int index]
{
    get
    {
        int position = 0;
        foreach (string value in list)
        {
            if (position == index)
                return value;
            ++position;
        }
        throw new ArgumentOutOfRangeException();
    }
}
private List<string> list;
}
```

Sufficient Skill

```
public class RecentlyUsedList
{
    public void Add(string newItem)
    {
        list.Remove(newItem);
        list.Add(newItem);
    }
    public int Count
    {
        get
        {
            return list.Count;
        }
    }
    public string this[int index]
    {
        get
        {
            return list[Count - index - 1];
        }
    }
    private List<string> list = new List<string>();
}
```


Decremental Development

- Don't include the unused or repeat yourself like a broken record (or CD... whatever)
 - ◆ *Eliminate Waste* (Lean Development), *Don't Repeat Yourself* (Pragmatic Programming), *Once And Once Only* (Extreme Programming), *Omit Needless Code* and *Unify Duplicate Code* (Programmer's Dozen)
- Refactoring, encapsulation, conversation and libraries help in the search for the minimum
 - ◆ These are hill-descending techniques

Refactoring is Gradual, ...

```
class access_control
{
public:
    bool is_locked(const std::basic_string<char> &key) const
    {
        std::list<std::basic_string<char> >::const_iterator found = std::find(locked.begin(), locked.end(), key);
        return found != locked.end();
    }
    bool lock(const std::basic_string<char> &key)
    {
        std::list<std::basic_string<char> >::iterator found = std::find(locked.begin(), locked.end(), key);
        if(found == locked.end())
        {
            locked.insert(locked.end(), key);
            return true;
        }
        return false;
    }
    bool unlock(const std::basic_string<char> &key)
    {
        std::list<std::basic_string<char> >::iterator found = std::find(locked.begin(), locked.end(), key);
        if(found != locked.end())
        {
            locked.erase(found);
            return true;
        }
        return false;
    }
    ...
private:
    std::list<std::basic_string<char> > locked;
    ...
};
```

Stable...

```
class access_control
{
public:
    bool is_locked(const std::string &key) const
    {
        return std::count(locked.begin(), locked.end(), key) != 0;
    }
    bool lock(const std::string &key)
    {
        if(is_locked(key))
        {
            return false;
        }
        else
        {
            locked.push_back(key);
            return true;
        }
    }
    bool unlock(const std::string &key)
    {
        const std::size_t old_size = locked.size();
        locked.remove(key);
        return locked.size() != old_size;
    }
    ...
private:
    std::list<std::string> locked;
    ...
};
```

And Goal Oriented

```
class access_control
{
public:
    bool is_locked(const std::string &key) const
    {
        return locked.count(key) != 0;
    }
    bool lock(const std::string &key)
    {
        return locked.insert(key).second;
    }
    bool unlock(const std::string &key)
    {
        return locked.erase(key);
    }
    ...
private:
    std::set<std::string> locked;
    ...
};
```

The Value of Economy

- Economy represents care, far-sightedness and elimination of waste
 - ◆ In code, clear as you go and keep it streamlined
 - ◆ In process, include what is needed and keep it light
- It is not a principle on its own, but a consideration with many consequent benefits
 - ◆ If you want fast code, keep it clean
 - ◆ If you want extensible code, keep it clean
 - ◆ If you want testable code, keep it clean

And Elegance

People in high tech take pride in their work. They are individuals who see the details of the things they produce in the light of the trials and triumphs they experience while creating products. In the courage of creation, they find a place to hang their individuality. Programmers and techno types appreciate elegant, spare code and the occasional well-turned architectural hack.

Rick Levine, *The Cluetrain Manifesto*

Read your own compositions, and when you meet with a passage which you think is particularly fine, strike it out.

Samuel Johnson

Aesthetics

- Yes, aesthetics matter
 - ◆ In particular, elegance relates to grace, style, ingenuity and simplicity, and the sense that there is pleasure to be derived from this
 - ◆ Elegance offers guidance in economy
- Taste may be personal, but common ground and inter-subjective consensus often exists
 - ◆ Taste derives from a mix of culture, consideration and experience

grep

```
int grep(char *regexp, FILE *f, char *name)
{
    int n, nmatch;
    char buf[BUFSIZ];

    nmatch = 0;
    while (fgets(buf, sizeof buf, f) != NULL) {
        n = strlen(buf);
        if (n > 0 && buf[n-1] == '\n')
            buf[n-1] = '\0';
        if (match(regexp, buf)) {
            nmatch++;
            if (name != NULL)
                printf("%s:", name);
            printf("%s\n", buf);
        }
    }
    return nmatch;
}
```

```
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
        return matchhere(regexp+1, text+1);
    return 0;
}

int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}

int matchstar(int c, char *regexp, char *text)
{
    do {
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}
```

eval

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                          (lambda-body exp)
                          env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                 (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Take a Different Point of View

- It can be easy to stick (or get stuck) with a design for the wrong reasons
 - ◆ We can keep a design out of conservatism against change — assuming that it has to be that way because it always has, notionally preserving the status quo but actually compounding the problem
 - ◆ We may fall in love with a design because it seemed particularly fine and rather clever — this is not quite the same as the refinement and aesthetics associated with elegance

Conflict without Cause

There have always been fairly severe size constraints on the Unix operating system and its software. Given the partially antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy but a certain elegance of design.

Dennis Ritchie and Ken Thompson

In Conclusion

- Software engineering needs to be reclaimed in terms of the reality of successful practice
 - ◆ There is emerging coherence and professionalism
 - ◆ But there is an inevitable tension between the old and new views of software engineering
- Economy and elegance are not opposites
 - ◆ Economy ensures that what is added has value and that what is removed preserves or improves value
 - ◆ Elegance places humans in the picture, encouraging comprehensibility, care and creativity