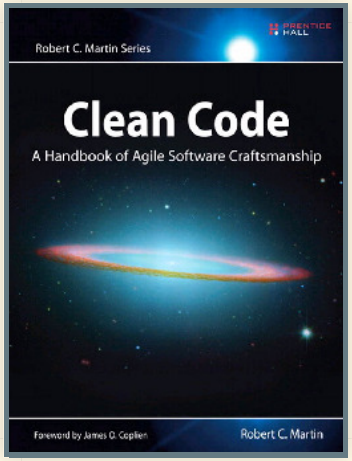
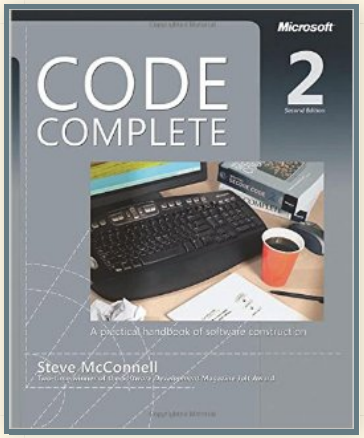
GOOD CODING PRACTICE

By Ben Keith

Quoin, Inc.

BASED ON


Robert C. Martin Series
PRENTICE HALL
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin


Microsoft
CODE COMPLETE
2
A practical handbook of software construction
Steve McConnell

WHAT IS BAD CODE?

- Difficult to read and understand
- Might be more complex than necessary
- Unexpected behaviors/surprises

WHY BAD CODE

- Developer ignorance/incompetence
- Laziness
- Schedule pressure
  - "I'll fix this later"
- Some combination of those

TECHNICAL DEBT

- Writing bad code
  - Like taking out a high interest loan
  - Immediate benefit but debt service mounts up fast
  - Not fixing the code is like making interest-only payments
  - Project failure/abandonment = Bankruptcy

SHORT/MID TERM EFFECTS

- Decreased productivity
- Increased difficulty of task estimation
- Decreased morale

RESPONSIBILITY

- Developers should take responsibility for bad code
- Managers are only responsible if they rush code contrary to the judgment of the developers and don't make time to fix it shortly

GOOD CODE

- Clean
- Concise
- Understandable
- Conforms to the programming language

NAMING

- The most powerful documentation for your code
- Good names dramatically improve understandability
  - Obviates the need for many types of comments

- Don't use cryptic or abbreviated names unless they are established convention

```
ls_set_tab: function (tab) {
  localStorage.setItem('current_tab', tab);
},
```

or

```
save_tab_in_browser: function(tab) {
  localStorage.setItem('current_tab', tab);
},
```

```
determine_current_tab: function(action) {
  var active_tab = localStorage.getItem(action + '_tab'),
      tab = $(active_tab),
      tab_nav = $('a[href="' + active_tab + '"]'),
      subgroup = tab_nav.parents('ul.sub');

  if (active_tab !== null && tab.length) {
    $(".tab-handles li").removeClass("current");
    $(".tab").hide();

    tab_nav.parents('li').addClass("current");

    if (subgroup) {
      subgroup.slideDown('normal');
    }

    tab.show();
```

BE SPECIFIC

```
...
getNextPage: function() {
    var uri = new Uri(document.location.toString());
    uri.setPath('/' + this.model.id + '/');
    uri.setAnchor('login/verify/email-changed');
    return uri.toString();
},
...
```

- Don't be too lazy to use (reasonably) long, descriptive names
  - IDEs (and even vim) support code completion
  - If a function name is too long you might be trying to do too much in it

FUNCTIONS: DO ONE THING

- What is a function/method/routine/procedure?

> *[T]he reason we write functions is to decompose a larger concept (in other words, the name of the function) into a set of steps at the next level of abstraction. -- Clean Code*

```python
def verify_recaptcha(remote_ip_addr, user_response):
    data = {
        'secret': settings.RECAPTCHA_SECRET,
        'remoteip': remote_ip_addr,
        'response': user_response,
    }
    resp = requests.post("https://www.google.com/recaptcha/api/siteverify", data=data, verify=False)

    try:
        return resp.json().get('success', False)
    except ValueError:
        logger.error("Recaptcha verification request returned malformed response: %s" %
                     (resp.text,))
        return False
```

vs.

```
def verify_recaptcha(remote_ip_addr, user_response):
    request_data = create_recaptcha_request_data(remote_ip_addr, user_response)
    resp = requests.post("https://www.google.com/recaptcha/api/siteverify", data=request_data)
    return test_recaptcha_response(resp)

def create_recaptcha_request_data(remote_ip_addr, user_response):
    return {
        'secret': settings.RECAPTCHA_SECRET,
        'remoteip': remote_ip_addr,
        'response': user_response,
    }

def test_recaptcha_response(resp):
    try:
        return resp.json().get('success', False)
    except ValueError:
        logger.error("Recaptcha verification request returned malformed response: %s" %
                     (resp.text,))
        return False
```

Doesn't have to be reused to be split out

```
        def dispatch(self, request, *args, **kwargs):
            self.request = request
            self.args = args
            self.kwargs = kwargs
            if request.customer.is_authenticated:
                self.user = request.customer.username
            else:
                self.user = self.kwargs['token'].split(":", 1)[0]

                signer = Signer()

                try:
                    original = signer.unsign(self.kwargs['token'])
                except signing.BadSignature:
                    return self.invalid()

            return super(ChangePasswordView, self).dispatch(request, *args, **kwargs)
```

> *[A] way to know that a function is doing more than "one thing" is if you can extract another function from it with a name that is not merely a restatement of its implementation. -- Clean Code*
>
> *You've extracted too much when your code itself is more clear than the method name.*

- Everything a function does should be at the same level of abstraction

DRY

```
...
var dfd = new $.Deferred();
profanityPromise1 = Profanity.checkText(this.formData.caption).fail(function(){
        self.error.formErrors = 'Please fix the fields in red';
        self.error.fieldErrors['caption'] = "This field contain a profanity";
    });
profanityPromise2 = Profanity.checkText(this.formData.credit).fail(function(){
        self.error.formErrors = 'Please fix the fields in red';
        self.error.fieldErrors['credit'] = "This field contain a profanity";
    });
...
```

```
class ApplicationController < ActionController::Base
  .....
  def check_user_license
    if current_user.user_licenses.present?
      #check all user licenses to see if ANY are active or review period of time
      if current_user.user_licenses.count == 1
        #The first clause is used because there's just one user license related in this scenario.
        if current_user.user_licenses.first.review_expiration_date.present? and current_user.user_licenses.
          user_any_active = current_user.user_licenses.where("user_licenses.active = true and (user_license
        else
          user_any_active = current_user.user_licenses.where("user_licenses.active = true and user_licenses
        end
      else
        user_any_active = current_user.user_licenses.where("user_licenses.active = true and user_licenses.e
      end

      if user_any_active
```

VS.

```ruby
class User < ActiveRecord::Base
  .....
  def has_active_license
    self.user_licenses.any? do |lic|
      valid_exp_date = lic.expiration_date > Time.now
      valid_review_exp_date = lic.review_expiration_date.present? and lic.review_expiration_date > Time.now

      lic.active and (valid_exp_date or valid_review_exp_date)
    end
  end
  .....
```

- Same logic repeated with slightly different values

```
var quoteShowOnFSFlag = document.getElementsByName("quoteShowOnFSFlag");
var chkQuoteShowOnFSFlag = document.getElementsByName("chkQuoteShowOnFSFlag");
for (var i=0; i<chkQuoteShowOnFSFlag.length; i++)
{
  if (chkQuoteShowOnFSFlag[i].checked == true)
  {
    quoteShowOnFSFlag[i].value = "Y";
  }
  else
  {
    quoteShowOnFSFlag[i].value = "N";
  }
}

//set the quote show in onix list
var quoteShowInOnixFlag = document.getElementsByName("quoteShowInOnixFlag");
var chkQuoteShowInOnixFlag = document.getElementsByName("chkQuoteShowInOnixFlag");
for (var i=0; i<chkQuoteShowInOnixFlag.length; i++)
{
  if (chkQuoteShowInOnixFlag[i].checked == true)
  {
    quoteShowInOnixFlag[i].value = "Y";
  }
  else
  {
    quoteShowInOnixFlag[i].value = "N";
  }
}

// AWARD ----------------------------------------------------------------------------------------------------
//set the quote show on FS list : AWARD
//alert("save() : awardShowOnFSFlag");
var awardShowOnFSFlag = document.getElementsByName("awardShowOnFSFlag");
var chkAwardShowOnFSFlag = document.getElementsByName("chkAwardShowOnFSFlag");
for (var i=0; i<chkAwardShowOnFSFlag.length; i++)  {
  if (chkAwardShowOnFSFlag[i].checked == true)    {
    awardShowOnFSFlag[i].value = "Y";
  }  else  {
    awardShowOnFSFlag[i].value = "N";
  }
}
//set the quote show on InOnixt : AWARD
//alert("save() : awardShowInOnixFlag");
var awardShowInOnixFlag = document.getElementsByName("awardShowInOnixFlag");
var chkAwardShowInOnixFlag = document.getElementsByName("chkAwardShowInOnixFlag");
```

```
  }  else  {
    awardShowInOnixFlag[i].value = "N";
  }
}
//set the quote show on OnWebSite : AWARD
//alert("save() : awardShowOnFSFlag");
var awardShowOnWebSiteFlag = document.getElementsByName("awardShowOnWebSiteFlag");
var chkAwardShowOnWebSiteFlag = document.getElementsByName("chkAwardShowOnWebSiteFlag");
for (var i=0; i<chkAwardShowOnWebSiteFlag.length; i++)  {
  if (chkAwardShowOnWebSiteFlag[i].checked == true)    {
    awardShowOnWebSiteFlag[i].value = "Y";
  }  else  {
    awardShowOnWebSiteFlag[i].value = "N";
  }
}
// AWARD ------------------------------------------------------------------------------------------------------------

//set the prepack list
var prepackShowOnFS = document.getElementsByName("chkDisplayShowOnFS");
var displayShowOnFS = document.getElementsByName("displayShowOnFS");
for (var i=0; i<prepackShowOnFS.length; i++)
{
  if (prepackShowOnFS[i].checked == true)
  {
    displayShowOnFS[i].value = "Y";
  }
  else
  {
    displayShowOnFS[i].value = "N";
  }
}

//set show on fact sheet flag for author history
var compShowOnFS = document.getElementsByName("compShowOnFS");
var compTitleShowOnFS = document.getElementsByName("compTitleShowOnFS");
for (var i=0; i<compShowOnFS.length; i++)
{
  if (compShowOnFS[i].checked == true)
  {
    compTitleShowOnFS[i].value = "Y";
  }
  else
  {
    compTitleShowOnFS[i].value = "N";
  }
}

if (document.FactSheetForm.disCompsP2.checked == true)
```

```
else
{
  document.FactSheetForm.displayCompsP2.value = "N";
}

var authShowOnFS = document.getElementsByName("authShowOnFS");
var authHistShowOnFS = document.getElementsByName("authHistShowOnFS");
for (var i=0; i<authShowOnFS.length; i++)
{
  if (authShowOnFS[i].checked == true)
  {
    authHistShowOnFS[i].value = "Y";
  }
  else
  {
    authHistShowOnFS[i].value = "N";
  }
```

vs.

```
function setHiddenInputsFromCheckBoxState(hidden_name, checkbox_name) {
  var checkboxes = document.getElementsByName(checkbox_name);
  var hiddenInputs = document.getElementsByName(hidden_name);
  for (var i = 0; i < checkboxes.length; i++) {
    hiddenInputs[i] = checkboxes[i].checked ? "Y" : "N";
  }
}

var hiddenInputNameToCheckboxName = {
  quoteShowOnFSFlag: "chkQuoteShowOnFSFlag",
  quoteShowInOnixFlag: "chkQuoteShowInOnixFlag",
  awardShowOnFSFlag: "chkAwardShowOnFSFlag",
  awardShowInOnixFlag: "chkAwardShowInOnixFlag",
  awardShowOnWebSiteFlag: "chkAwardShowOnWebSiteFlag",
  chkDisplayShowOnFS: "displayShowOnFS",
  compShowOnFS: "compTitleShowOnFS",
  authShowOnFS: "authHistShowOnFS",
```

ANONYMOUS CALLBACK FUNCTIONS IN JS (OR RUBY)

- Example of both mixed abstraction levels and way too much nesting

```
submit: function(e) {
  .....
  $.when(user.save(formData, {wait: true}))
    .done(function() {
      $.when(user.fetch()).done(function() {
        self.error = null;
        self.resetFormData();
        self.trigger("complete");
        //self.$("span.error").remove();
        //self.$("div.errorTop").remove();
        $.when(self.render()).done(function() {
          if( !self.$('.save_top').hasClass('show_notice') ) {
              self.$('.save_top').addClass('show_notice');
          }

          $("body,html").animate({ scrollTop: 0 }, "slow");
          setTimeout(function(){
              self.$('.save_top').removeClass('show_notice')
          }, 5000);
```

- Avoid more than one level of callback nesting in a single function
- Use of anonymous functions avoids "onDone"/"handle.." functions
- Body of anonymous callback should follow previous rules
- Process the event arg to the callback and then call well named functions
  - Don't mix abstraction levels in callbacks
  - example

SINGLE RESPONSIBILITY PRINCIPLE

- A class has only one reason to change (one responsibility)
- Classes have high cohesion

```
class AbstractParser(object):
    .....
    def __init__(self, paper_id, notice_date, subcategory_id):
        .....

    def authenticate(self, username, password):
        .....
        try:
            return models.Paper.objects.get(username=username.strip(), password=password.strip()).pk
        except models.Paper.DoesNotExist:
            raise AuthenticationError(username)

    def parse(self, content):
        .....

    def to_unix(self, content):
        """
```

- SRP is prone to being violated in controller methods in web frameworks
- If you follow SRP you will go a long way to writing loosely coupled code
- Testing is easier

User license example from earlier

FORMATTING

- If you should not logically split up a function at least add vertical spacing
- Think of it like paragraphs and sentences

```ruby
def all_exportable_fields_by_form(primero_modules, record_type, user, types, reports=false)
  #Custom export does not have "violation" just reports.
  parent_form = record_type == "violation" ? "incident" : record_type
  #hide_on_view_page will filter fields for readonly users.
  readonly_user = user.readonly?(parent_form)
  custom_exportable = {}
  if primero_modules.present?
    primero_modules.each do |primero_module|
      if record_type == 'violation'
        #Custom export does not have violation type, just reporting.
        #Copied this code from the old reporting method.
        forms = FormSection.get_permitted_form_sections(primero_module, parent_form, user)
        violation_forms = FormSection.violation_forms
        forms = forms.select{|f| violation_forms.include?(f) || !f.is_nested?}
      else
        if reports
          #For reporting show all forms, not just the visible.
          forms = FormSection.get_permitted_form_sections(primero_module, parent_form, user)
          #For reporting avoid subforms.
          forms = forms.select{|f| !f.is_nested?}
        else
          #For custom export shows only visible forms.
          forms = FormSection.get_allowed_visible_forms_sections(primero_module, parent_form, user)
          #Need a plain structure.
          forms = forms.map{|key, forms_sections| forms_sections}.flatten
        end
      end
      if reports
        #For reporting just filter by type.
        include_field = lambda do |f|
          types.include?(f.type)
        end
      else
        #For custom export filter by type and visible.
        include_field = lambda do |f|
          types.include?(f.type) && f.visible?
        end
      end
      #Collect the information as: [[form name, fields list], ...].
      #fields list got the format: [field name, display name, type].
      #fields list for subforms got the format: [subform name:field name, display name, type]
      #Subforms will appears as another section because there is no way
      #to manage nested optgroup in choosen or select.
      forms_and_fields = []
      forms.sort_by{|f| [f.order_form_group, f.order]}.each do |form|
```

```
      if f.subform_section.present?
        #Collect subforms fields to build the section.
        subform_fields = f.subform_section.fields.select{|sf| types.include?(sf.type) && sf.visible
        subform_fields = subform_fields.map do |sf|
          ["#{f.name}:#{sf.name}", sf.display_name, sf.type] if !readonly_user || (readonly_user &&
        end
        subforms << ["#{form.name}:#{f.display_name}", subform_fields.compact]
      end
    end
  else
    #Not subforms fields.
    fields << [f.name, f.display_name, f.type] if !readonly_user || (readonly_user && !f.hide_on_vi
  end
end
#Add the section for the current form and the not subforms fields.
forms_and_fields << [form.name, fields]
#For every subform add the section as well.
```

That function should have been split up though

PROGRAMMING IS ITERATIVE DESIGN

- You will most likely not get the code right the first time
- Don't settle for "just working"
  - An book author rereads his writing and makes many changes before it is "done"

Example of not doing that

TESTING

- Thorough automated test suite is essential for:
  - Confident refactoring
  - Maintaining productivity
- No single developer can fully understand a non-trivial application
- Your tests should be written to the same standard as the application

THE END