

CSC 7700: Scientific Computing

Module A: Basic Skills

Lecture 2: Collaboration management, Programming best practices

Dr Frank Löffler

Center for Computation and Technology
Louisiana State University, Baton Rouge, LA

Sep 06 2013



Overview



Software development, or

- Application Development
- Software Design
- Software Engineering
- Software Application Development
- Enterprise Application Development
- Platform Development

... development of a software product in a planned and structured process.



Software development involves some combination of stages:

- Market research
- Gathering requirements for the proposed business solution
- Analyzing the problem
- Devising a plan or design for the software-based solution
- Implementation (coding) of the software
- Testing the software
- Deployment
- Maintenance and bug fixing

Collection of stages: software development life-cycle (SDLC).

- Very different methodologies to combine stages exist.
- Choice of methodology should be project-dependent.



Software development in Science

Software development involves some combination of stages:

- Market research
- Gathering requirements for the proposed business solution
- Analyzing the problem
- Devising a plan or design for the software-based solution
- Implementation (coding) of the software
- Testing the software
- Deployment
- Maintenance and bug fixing

Collection of stages: software development life-cycle (SDLC).

- Very different methodologies to combine stages exist.
- Choice of methodology should be project-dependent.



Before starting implementation: create “project environment”:

- Communication channels
- Version control system
- Bug tracker and tasks list
- Documentation format
- Testing tools
- Package management



Communication channels



Dependent on team distribution, consider possibilities like:

- In-person meetings
- Conference phone calls
- Email, especially dedicated mailing lists
- Instant messaging (e.g. IRC)
- VoIP/Video-conferences (e.g. Skype/Google Hangouts/BigBlueButton)



Version control systems

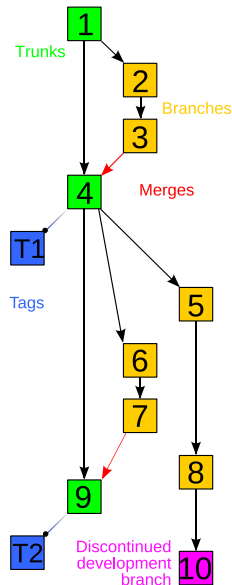


Version control systems

Also known as

- Revision control
- Source control
- Software configuration management

Definition: Management of changes to documents, programs, and other information stored as computer files



Issue tracker



- Synonyms: trouble ticket system, support ticket or incident ticket system
- More restricted: bug tracking system, bug tracker
- Database of “tickets”, describing issues/incidents/bugs

Workflow

- 1 User notices bug/issue/problem
- 2 (User tries to create small test case, presenting the problem)
- 3 User creates/opens ticket in issue tracker
- 4 Developer reproduces problem
- 5 Developer fixes problem
- 6 Developer closes ticket, notifying User



Tickets/Issues can have attached

- Type (e.g. defect/enhancement)
- Priority (e.g. minor, major, critical, blocker)
- Project component
- Target project milestone
- Version of project component
- List of people CC'ed on changes of ticket
- Owner
- Files (e.g. patches)

Benefits of issue trackers over, e.g. direct developer contact

- Issues are recorded in database, cannot be forgotten
- Users can look-up if specific problem was already reported
- Users can automatically get change notifications



- A large number of stand-alone issue tracker implementations exist
 - Trac
 - Bugzilla
 - GNATS
- Open-source hosting sites usually automatically provide issue tracking systems, e.g.
 - sourceforge
 - savannah
 - seul
 - github
 - google code



Documentation format



Depending on need, various formats possible

- Plain text
- Man pages / Help system documents
- Application-internal
- Print-oriented, e.g. \LaTeX , word processor
- Wiki
- Website as in plain HTML and typically in RCS



Summary



User- and development-friendly project environment provides:

- Information about project: e.g. website
- Communication channels for developers
- Infrastructure for shared code development
 - Project standards
 - Revision control system
- Communication channels for users, especially
 - Channel for problems/issues, directed at developers
 - Users-for-users channel



Best Coding Practices

or

How not to annoy your collaborators



Best Coding Practices - Don't just do it... do it right!

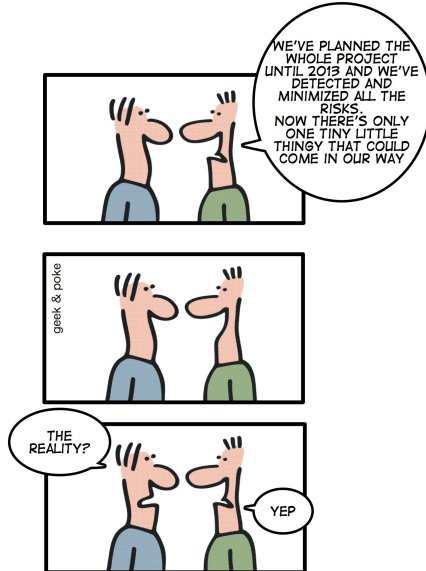
- Project planning
 - First of all: have a plan!
- Programming styles and conventions
 - Improve readability (to others and yourself)
 - Reduce the probability of (you) introducing errors
 - Make contributions by others more likely



Project Planning



Some day on Geek & Poke



<http://geekandpoke.typepad.com/>

"PLAN THE WORK, WORK THE PLAN"



Plan ahead!

- Define goals
- Define sub-goals
- Define road-map
- Bad plan often is better than having none
- The complete team must understand plan before start
- Do not deviate without reason

Design pitfalls

- Over-designing: 'Don't bite off more than you can chew'
- Two generally good principles
 - "Keep it Simple, Stupid!" - KISS
 - Utilize information hiding



KISS is acronym for

- Keep it simple, Stupid!
- Keep it short and simple

Key points:

- Simplicity should be a key goal in design
- Unnecessary complexity should be avoided

Related concepts:

- Occam's razor (We should tend towards simpler theories)
- Einstein: *"Everything should be made as simple as possible, but no simpler."*
- Antoine de Saint Exupéry: *"It seems that perfection is reached not when there is nothing left to add, but when there is nothing left to take away."*



Code review / Peer review:

- Look at other peoples work. Learn from it.
- Solutions for problems often available - use them.
- Let others see your code and learn from their knowledge.
- Sometimes: program together (walk-through, pair programming)



Testing



- Should not be an afterthought
- Integral part of software development
- Needs to be planned, and done proactively
- Developed while the application is being designed and coded



Functional testing

- Verify specific action or function of code
- Usually found in code requirements documentation
- “Can the user do this”

Non-functional testing

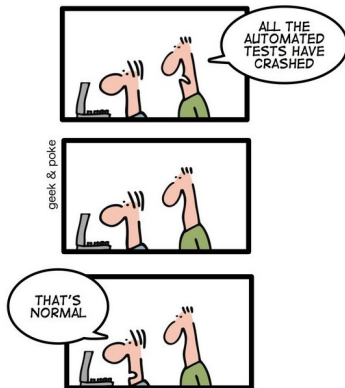
- Not related to specific action or function, e.g.
 - Scalability
 - Testability
 - Maintainability
 - Usability
 - Performance
 - Security



Today on Geek & Poke

GEEK & POKE'S LIST OF BEST PRACTICES

*TODAY: CONTINUOUS INTEGRATION
GIVES YOU THE COMFORTING
FEELING TO KNOW THAT
EVERYTHING IS NORMAL*



<http://geekandpoke.typepad.com/>



Source specific coding styles

Identifier naming



Naming conventions

Reasons:

- to reduce the effort needed to read and understand source code
- to enhance source code appearance
(for example, by disallowing overly long names or abbreviations)
- to enhance clarity in cases of potential ambiguity
- to help avoid "naming collisions" that might occur when the work product of different organizations is combined



Considerations:

- shorter identifiers may be preferred because they are easier to type
- extremely short identifiers are very difficult to uniquely distinguish using automated search and replace tools
- longer identifiers may be preferred because short identifiers cannot encode enough information or appear too cryptic
- longer identifiers may be disfavored because of visual clutter



Programmers generally tended to use short identifiers, in part because of

- programming languages with length limitations
- early linkers which required variable names to be restricted to 6 characters to save memory
- early source code editors lacking auto-complete
- early low-resolution monitors with limited line length (e.g. only 80 characters)
- much of computer science originating from mathematics, where variable names are often only a single letter



Identifier length example

Compare

```
get a b c
```

```
if a < 24 and b < 60 and c < 60
```

```
    return true
```

```
else
```

```
    return false
```

to

```
get hours minutes seconds
```

```
if hours < 24 and minutes < 60 and seconds < 60
```

```
    return true
```

```
else
```

```
    return false
```



Naming Conventions

A set of rules for choosing identifiers

- Hungarian Notation
 - embed information (e.g. type) into name
 - lower case mnemonics
 - examples: `sName`, `strName`, `iMax`, `intMax`, `i_max`
 - popular primarily in Microsoft environments
- Underscore style
 - underscore “_” between compound words
 - might be confused with minus sign
 - underscore inconvenient on some keyboard layouts
- CamelCase
 - compound words, joined without spaces, capitalized words
 - uses less characters than underscore notation
 - inappropriate for case-insensitive languages



Source specific coding styles

Source code formatting



Source code formatting *or* Programming style

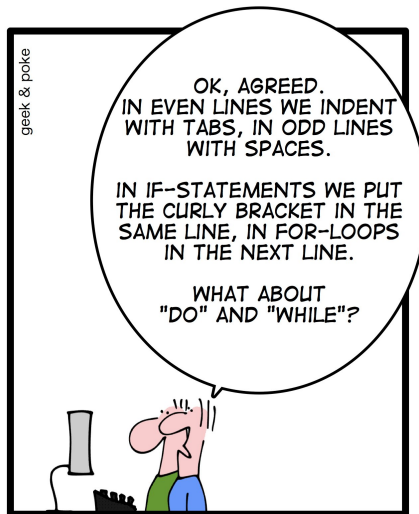
- Often designed for a specific programming language
- Large projects or companies usually define style

Common elements

- Layout of source code, including indentation
- Use of white space around operators and keywords
- Naming Conventions
- Use and style of comments
- Use or avoidance of particular programming constructs



SIMPLY EXPLAINED



PAIR PROGRAMMING

<http://geekandpoke.typepad.com/>



Indent style

- Assists in identifying control flow and blocks of code
- Mandatory in some programming languages

Compare

```
if (hours < 24 && minutes < 60 && seconds < 60)
{
    return true;
}
else
{
    return false;
}
```

or

```
if (hours < 24 && minutes < 60 && seconds < 60) {
    return true;
} else {
    return false;
}
```

to

```
if (    hours<
24  && minutes<
60  && seconds<
60  )
{return    true
;}        else
{return    false
;}        ;}
```



Vertical alignment

Vertical alignment is often helpful to arrange similar elements.

Compare

```
$search = array('a', 'b', 'c', 'd', 'e');  
$replacement = array('foo', 'bar', 'baz', 'quux');
```

Another example:

```
$value = 0;  
$anothervalue = 1;  
$yetanothervalue = 2;
```

to

```
$search      = array('a', 'b', 'c', 'd', 'e');  
$replacement = array('foo', 'bar', 'baz', 'quux');
```

Another example:

```
        $value = 0;  
    $anothervalue = 1;  
$yetanothervalue = 2;
```



- Most free-format languages unconcerned about amount of allowed whitespace
- Generally matter of taste
- Good practice: be consistent

```
int i;  
for (i=0;i<10;++i){  
    printf("%d",i*i+i);  
}
```

```
int i;  
for (i=0; i<10; ++i) {  
    printf("%d", i*i+i);  
}
```

```
int i;  
for (i = 0; i < 10; ++i) {  
    printf ("%d", i * i + i);  
}
```

```
int i;  
for( i = 0; i < 10; ++i ) {  
    printf( "%d", i * i + i );  
}
```



Tabs versus Spaces: An Eternal Holy War

People care about a few different things

- ① Amount of screen columns code is indented
 - a lot of different views (mainly 2, 4 or 8 spaces)
 - might depend on context
- ② How TAB characters in files are displayed on screen
 - historic: move to the right until the current column is a multiple of 8
 - many Microsoft Windows and Mac editors: same as above, but multiple of 4
 - many editors configurable
 - alternative: indent to the next tab stop (where tab stop is file-dependent)
- ③ What happens when the TAB key is pressed
 - possibility 1: Insert TAB character as is
 - possibility 2: Indent this line
(cause the first non-whitespace character on this line to occur at specific column)



Tabs versus Spaces: An Eternal Holy War

People care about a few different things

- 1 Amount of screen columns code is indented
Core issue - matter of taste
- 2 How TAB characters in files are displayed on screen
Technical issue, interoperability
- 3 What happens when the TAB key is pressed
Technical issue, interoperability

Solutions:

- Agreement within project
- Avoid TAB characters in files or, at least:
Avoid TABS for alignment, use only for indentation



Source specific coding styles

General programming practices



Left-hand comparisons

Remove possible errors by using left-hand comparisons:

Comparison:

```
// A right-hand comparison checking if $a equals 42.  
if ( $a == 42 ) { ... }  
// Recast, using the left-hand comparison style.  
if ( 42 == $a ) { ... }
```

Assignment:

```
// Inadvertent assignment which is often hard to debug  
if ( $a = 42 ) { ... }  
// Compile time error indicates source of problem  
if ( 42 = $a ) { ... }
```



Looping and control structures

Use the “right” loop structure, for example:

```
i = 0
while i < 5
    print i * 2
    i = i + 1
end while
print "Ended loop"
```

vs.

```
for i = 0, i < 5, i=i+1
    print i * 2
print "Ended loop"
```



Curly brackets and loops

Use curly brackets even when not necessary (depends on language), e.g.:

```
/* The incorrect indentation hides the fact that this  
line is not part of the loop body. */
```

```
    for (i = 0; i < 5; ++i);  
/* → */    printf("%d\n", i*2);  
            printf("Ended loop");
```

or

```
/* The incorrect indentation hides the fact that this  
line is not part of the loop body. */
```

```
    for (i = 0; i < 5; ++i)  
        fprintf(logfile, "loop reached %d\n", i);  
/* → */    printf("%d\n", i*2);  
            printf("Ended loop");
```



List separators

Add list separator after final element in list (where supported):

```
const char *array[] = {  
    "item1",  
    "item2",  
    "item3", /* still has the comma after it */  
};
```

Benefit: Prevents syntax errors and subtle string-concatenation bugs after re-ordering



Language specific convention examples

C, C++

- Keywords and standard library identifiers mostly lowercase
- Macro names only in upper case with underscores
- Names beginning with double underscores or underscore and capital letter are reserved for internals of implementation (standard library, compiler)

Perl

- Locally scoped variables and subroutine names are lowercase with underscores
- Subroutines and variables meant to be treated as private are prefixed with an underscore
- Declared constants are all caps
- Package names are camel case, except pragmas (e.g. `use strict;`)



Python

- UpperCamelCase for class names
- lowercase_separated_by_underscores for other names

Java

- Class names should be nouns in CamelCase.
- Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized
- Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign \$ characters, even though both are allowed.



- Think about documentation before you start writing
- Update documentation regularly
- Comment often, explain what is done

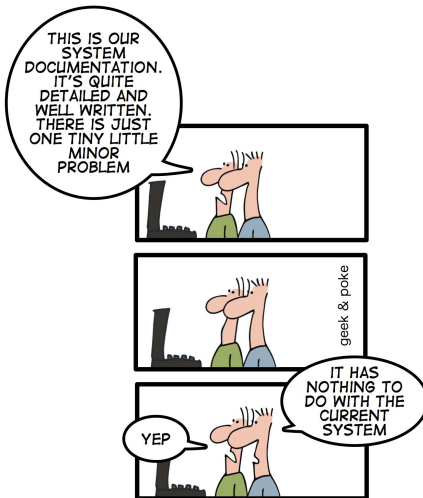
```
/* compute mass from integral over rho  
   as in paper xyz */  
double M = 0.0;  
for (int i=0; i<N; i++)  
{  
    M += rho[i] * volume[i];  
}
```

- Don't comment the obvious

```
/* print user name */  
print "$username\n";
```



SIMPLY EXPLAINED: TAUTOOLOGY



*EXAMPLE 1:
OUTDATED SYSTEM
DOCUMENTATION*

- Usually the opposite of good coding style
- Intellectual property protection
- Reduced security exposure
- Size reduction
- At best, merely makes it time-consuming, but not impossible, to reverse engineer a program
- Often depends on the particular characteristics of the platform and compiler, making ports difficult

→ Don't do it



- Usually the opposite of good coding style
- Intellectual property protection
- Reduced security exposure
- Size reduction
- At best, merely makes it time-consuming, but not impossible, to reverse engineer a program
- Often depends on the particular characteristics of the platform and compiler, making ports difficult

→ Don't do it - Except for fun



Obfuscation Example

Print prime numbers less than 100:

```
void primes(int cap) {  
    int i, j, composite;  
    for(i = 2; i < cap; ++i) {  
        composite = 0;  
        for(j = 2; j * j <= i; ++j)  
            composite += !(i % j);  
        if(!composite)  
            printf("%d\t", i);  
    }  
}  
  
int main(void) {  
    primes(100);  
}
```



Obfuscation Example

Rewrite for as while. Use special values.

```
void primes(int cap) {  
    int i, j, composite, t = 0;  
    while(t < cap * cap) {  
        i = t / cap;  
        j = t++ % cap;  
        if(i <= 1);  
        else if(!j)  
            composite = j;  
        else if(j == i && !composite)  
            printf("%d\\t", i);  
        else if(j > 1 && j < i)  
            composite += !(i % j);  
    }  
}
```



```
int main(void) {  
    primes(100);  
}
```



Obfuscation Example

Change iteration into recursion:

```
void primes(int cap, int t, int composite) {
    int i, j;
    i = t / cap;
    j = t % cap;
    if(i <= 1)
        primes(cap, t+1, composite);
    else if(!j)
        primes(cap, t+1, j);
    else if(j == i && !composite)
        (printf("%d\t", i), primes(cap, t+1, composite));
    else if(j > 1 && j < i)
        primes(cap, t+1, composite + !(i % j));
    else if(t < cap * cap)
        primes(cap, t+1, composite);
}

int main(void) {
    primes(100, 0, 0);
}
```



Obfuscation Example

Obfuscate constructs and use meaningless variable names

```
void primes(int m, int t, int c) {  
    int i,j;  
    i = t / m;  
    j = t % m;  
    (i <= 1) ? primes(m,t+1,c) : (!j) ? primes(m,t+1,j) : (j == i && !c) ?  
    (printf("%d\t",i), primes(m,t+1,c)) : (j > 1 && j < i) ?  
    primes(m,t+1,c + !(i % j)) : (t < m * m) ? primes(m,t+1,c) : 0;  
}  
  
int main(void) {  
    primes(100,0,0);  
}
```



Obfuscation Example

Remove intermediate variables and literals

```
void primes(int m, int t, int c) {
    ((t / m) <= 1) ? primes(m,t+1,c) : !(t % m) ? primes(m,t+1, t % m) :
    ((t % m)==(t / m) && !c) ? (printf("%d\t", (t / m)), primes(m,t+1,c)) :
    ((t % m)> 1 && (t % m) < (t / m)) ? primes(m,t+1,c + !((t / m) % (t % m))) :
    (t < m * m) ? primes(m,t+1,c) : 0;
}

int main(void) {
    primes(100,0,0);
}
```

Obfuscate names again

```
void _(int __, int ____, int ____){
    ((__ / __) <= 1) ? _(__,____+1,____) : !(__ % __) ? _(__,____+1,____ % __) :
    ((__ % __)==(__ / __) && !____) ? (printf("%d\t",(__ / __)),
    _(__,____+1,____)) : ((__ % __) > 1 && (__ % __) < (__ / __)) ?
    _(__,____+1,____ + !((__ / __) % (__ % __))) : (__ < __ * __) ?
    _(__,____+1,____) : 0;
}

int main(void) {
    _(100,0,0);
}
```



Obfuscation Example

Remove literals

```
void _ (int __, int ____, int ____ , int _____) {
    ((___ / __) <= _____) ? _ (__, ___+_____, _____) : ! (___ % __) ? _ (__, ___+_____, _____%
    __, _____) : ((___ % __)= (___ / __) && !_____) ? (printf ("%d\t", (___ / __)),
    _ (__, ___+_____, _____)) : ((___ % __) > _____ && (___ % __) < (___ / __)) ?
    _ (__, ___+_____, _____ + !((___ / __) % (___ % __)), _____) : (___ < _ * __) ?
    _ (__, ___+_____, _____, _____) : 0;
}

int main(void) {
    _ (100, 0, 0, 1);
}
```

Remove redundant text

```
_ (__, ____, _____) { ___ / __ <= _____ ? _ (__, ___+_____, _____) : ! (___ % __) ? _ (__, ___+_____,
    _____% __, _____) : ___ % __ = ___ / __ && !_____ ? (printf ("%d\t", ___ / __), _ (__, ___+_____, _____,
    _____)) :
    (___ % __ > _____ && ___ % __ < ___ / __) ? _ (__, ___+_____, _____ + ! (___ / __ % (___ % __)), _____) : ___ < _ * __ ?
    _ (__, ___+_____, _____, _____) : 0; } main(void) { _ (100, 0, 0, 1); }
```



Recreational obfuscation

```

#include <math.h>
#include <sys/time.h>
#include <X11/Xlib.h>
#include <X11/keysym.h>
double L, o, P,
, _=dt, T, Z, D=1, d,
s[999], E, h= 8, l,
J, K, w[999], M, m, O,
, n[999], j=33e-3, i=
1E3, r, t, u, v, W, S=
74.5, l=221, X=7.26,
a, B, A=32.2, c, F, H;
int N, q, C, y, p, U;
Window z; char f[52]
; GC k; main(){ Display*=
XOpenDisplay( 0); z=RootWindow(e,0); for (XSetForeground(e,k=XCreateGC (e,z,0,0),BlackPixel(e,0))
; scanf("%lf%lf%lf",y +n,w+y, y+s)+1; y ++); XSelectInput(e,z= XCreateSimpleWindow(e,z,0,0,400,400,
0,0,WhitePixel(e,0) ),KeyPressMask); for(XMapWindow(e,z); ; T=sin(O)){ struct timeval G={ 0,dt*1e6}
; K= cos(j); N=1e4; M+= H*_; Z=D*K; F+=_*P; r=E*K; W=cos( O); m=K*W; H=K*T; O+=D*_*F/ K+d/K*E*_; B=
sin(j); a=B*T*D-E*W; XClearWindow(e,z); t=T*E+ D*B*W; j+=d*_*D+_*F*E; P=W*E*B-T*D; for (o+=(l=D*W+E
*T*B, E=d/K *B+v+B/K*F*D)*_; p<y; ){ T=p[s]+i; E=c-p[w]; D=n[p]-L; K=D*m-B*T-H*E; if (p [n]+w[ p]+p[s
]= 0|K < fabs(W-T*r-l*E +D*P) | fabs(D=t *D+Z *T-a *E)> K)N=1e4; else{ q=W/K *4E2+2e2; C= 2E2+4e2/ K
*D; N=1E4&& XDrawLine(e ,z,k,N ,U,q,C); N=q; U=c; } ++p; } L+=_* (X*t +P*M+m*l); T=X*X+ l*l+M *M;
XDrawString(e,z,k ,20,380,f,17); D=v/l*15; i+=(B *l-M*r -X*Z)*_.; for( ; XPending(e); u *=CS!N){
XEvent z; XNextEvent(e ,&z);
++*((N=XLookupKeysym
(&z.xkey,0)) -IT?
N-LT? UP-N?& E:&
J:& u: &h); --*(
DN -N? N-DT ?N=
RT?&u: & W:&h:&J
); } m=15*F/l;
c+=(l=M/ l, l*H
+l*M+a*X)*_.; H
=A*r+v*X-F*l+(
E=.1+X*4.9/l, t
=T*m/32-l*T/24
)/S; K=F*M+(
h* 1e4/l -(T+
F*5*T*E)/3e2

```



Essential for project success:

- Planning, Evaluation
- Integrated testing

Main Coding style issues:

- Identifier naming
- Source code formatting
- Avoidance/Use of specific language constructs



Remote machine access

- Login to one of the XSEDE machines (e.g. 'stampede') and one of the LSU HPC machines ('supermike II')
- Familiarize yourself with transfer between these machines and your local machine

Simple programming

- Write a short, simple (a bit more than "hello world") program (surprise us)
- Write it well
- Compile and run at on XSEDE/HPC machine

Write short report on what you did, and commit that and source (coursework/A2/)

Deadline: **Fri Sep 13 2013**

