

Who are these  
Programmers?  
And  
What do they Do?

System Administrator

Software engineer

Web site

Webapp

Desktop app

Programmer

Software Life Cycle

Pseudocode

Generalization/Abstraction

Variable Names

Comments

Modularity

Not all computer people are  
programmers.

- Computer people specialize in different things:
  - hardware or software,
  - Unix or Mac or Windows,
  - networking,
  - security,
  - web technologies,
  - databases,
  - testing,
  - documentation....
- The U of A lists 28 different "computer" job titles.

# Programmer

Anyone can call themselves a **programmer**.

They could be self taught (like many system administrators), or

They could have a formal education (software engineering is the best education for a programmer)

Many physicists, mathematicians and other scientists do some programming...though they typically do NOT have the rigorous standards that software engineers do for good design and documentation.

# System Administrator

Someone who takes care of computers (i.e., installs software, checks for viruses, fixes what's broken).

System administrators have strengths and weaknesses, e.g., unix vs windows; networks; hardware, software.

They may be very sophisticated or barely competent.

They are usually self-taught, but may have various technical certificates.

Many have dabbled in programming.

# Software Engineer

\*Should\* have an education in writing software programs.

Trained in principles of good design, documentation and backup.

Has probably learned several programming languages.

May seem slow and expensive, but should produce superior software that is easy to use, understand, and change in the future.

(A computer engineer is NOT a software engineer)

What do Programmers Do?



# Programming is like Building a House

Many people have building skills, but we recognize them as different:

- Handyman, plumber, construction worker, electrician, architect

These people are  
**NOT**  
interchangeable.

- e.g., Just because the handyman fixed your stuck window does **NOT** mean you want him to design and build your house.

# The Software Lifecycle

Like an architect, a programmer should:

- Determine **requirements** through discussion
- Write up **specifications** based on your requests and his/her knowledge of what's possible (and document)
- Identify and **design** the pieces of code that fulfill the specification (and document)
- **Implement** it (and document)
- **Test** it
- **Deploy** it
- **Maintain** it (Support, fix)



**1. Distrust**  
Can I do it?



**2. Excitement**  
I can do it!!!



**3. Astonishment**  
How will I do it?



**4. Enthusiasm**  
I got hold of the flow!!!



**5. Love**  
I am an excellent programmer!



**6. Disillusionment**  
Code is not functioning properly



**7. Fright**  
Will this logic work?



**8. Horror**  
Another A level bug!!!



**9. Fury**  
Damn with computers  
#@#\$%^



**10. Frustration**  
It is not working in expected manner



**11. The End**  
Project Appraisal

# Finding Help

Usually, you start thinking about programmers when you need help.  
Specifically, you need a program that does X.

How should you proceed?

Look for existing tools. Does something similar to what you want already exist?

You may find people who are building something close.

These people are the professionals most likely to point you in the right direction.

# Modifications to Existing Code

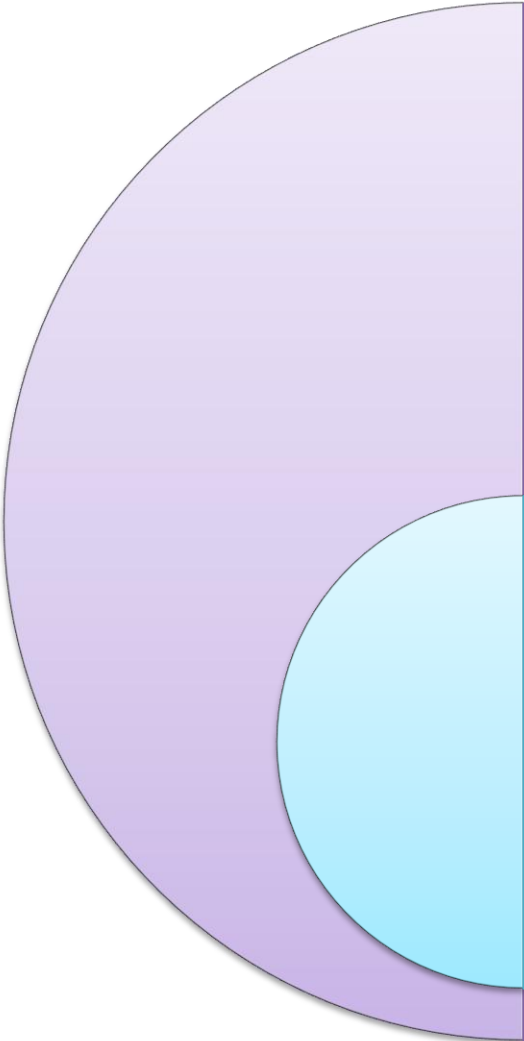
What happens if you find something similar?

If the original author of the program is willing to make additions, so much the better.

If you have well built, well-documented source code, then someone competent can make small additions without too much expense.

If the program structure is a confusing and undocumented disaster, it may be better to start over again.

# Something New

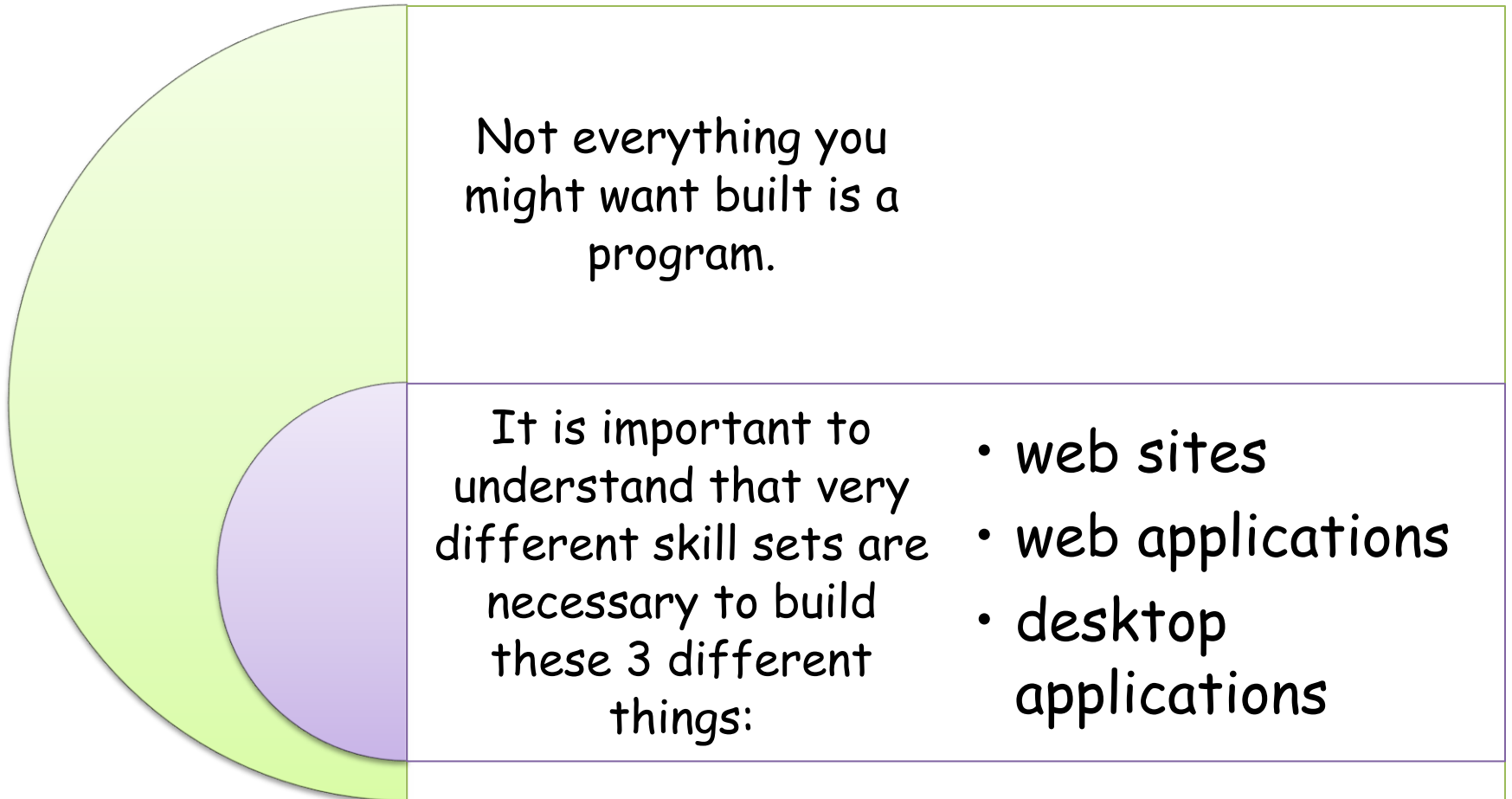


If you want to build something new, it matters whether you want a dog house or a mansion.

- It is fairly easy to automate a process that you can already do manually (unless it involves complex subjective human judgments)
- It should also be easy to apply a mathematical algorithm to some data (given the right scientific programmer)

However, it takes a team years to create, maintain and support a giant body of code that does lots of different things (e.g., FSL, Afni, SPM, BioImage Suite, ImageJ).

# 3 things you might want to build



# Web Sites, Webapps, Desktop Apps

A web site is not a program.

It is a static information source.

It does not require complex programming logic to build.

This means it can be created using HTML and perhaps some commercial software that helps with layouts and incorporates some javascript rollover buttons.



# Web Sites, Webapps, Desktop Apps

A web application is a program accessed over the web.

It may offer a database of information, or some basic data processing.

Programmers may specialize in writing webapps because it can be very complex and involve several layers, often in different languages.

Compared to a desktop app, a webapp is limited by the browser (IE, firefox etc).

# Web Sites, Web Apps, Desktop Apps

A desktop app is probably what you think of when you think of a program.

It is installed on your local machine and accessed there.

Because you have to install it anyhow, it is not limited by the capabilities of the browser.

# Summary

- Anyone can call themselves a programmer. You need to try to select the right person for the job.
  - Find similar tools
  - Inquire with the authors of those tools to get pointers
  - (Sort of like asking your neighbors to recommend a good plumber)

# Summary

- Programming tasks vary in difficulty
  - Try to distinguish easy from hard tasks.
  - Try to guess what kinds of skills your programmer will need.
  - It is important to distinguish whether you want a web site, web app or desktop app.

# What are the principles of good coding?

- 1) Generalization,
- 2) Variable Naming,
- 3) Comments,
- 4) Modularity

## Why do you Care?

- To identify a good programmer
- To write your own code

# Sample Task

- Tar and gzip all the directories and files in a particular location. Label them with their name and today's date.
- (Imagine there are 30-40 of these in a directory called Main):
  - dirA
  - dirB
  - MyC
  - YourD
  - dirE
- Move the tarred gzipped files and directories into a new dated directory called Backup+'date of tarring'.

# Pseudocode

- Pseudocode is just a representation of the logical structure of what you need to do. It doesn't follow the syntax on any particular language. e.g.,
  - Create a backup directory
  - Determine the date and time
  - List files
  - For each file in the list:
    - Tar the file into a correctly named tar file (with date/time info)
    - Move the resulting tar file to the backup directory

# Script 1

```
tar zcvf dirA_10-19-2008.tar.gz dirA
tar zcvf dirB_10-19-2008.tar.gz dirB
tar zcvf MyC_10-19-2008.tar.gz MyC
tar zcvf YourD_10-19-2008.tar.gz YourD
tar zcvf dirE_10-19-2008.tar.gz dirE
mkdir Backup_10-19-2008
mv *.tar.gz Backup_10-19-2008
```



Script 1 saves you from having to sit around all night waiting for each tar job to finish so you can type in the next command.

The problem is that you have to rewrite the script every time the directory names or the date changes.

The script does not **generalize** because it has no **abstractions**

# Script 2

```
x='date:M:D:Y'
```

```
y=`ls Main`
```

```
for z in $y
```

```
do
```

```
tar zcvf $z_"$x".tar.gz $z
```

```
done
```

```
mkdir "Backup_"$x
```

```
mv *.tar.gz "Backup_"$x
```

Script 2 is better. It is general and short.

The problem shows up when you want to re-use or expand the reasoning in the code.

The **variable names** are unhelpful (it's hard to keep track of x, y and z).

There are no **comments** in the code explaining what it does.

The longer the program/script, the more this lack of **documentation** (unhelpful variable names; no comments) makes it difficult to modify in the future.

# Script 3

*# Get Today's date automatically and put it in a variable*

**today='date:M:D:Y'**

*# List all the directories to backup and put the list in a variable*

**Dirs\_to\_Backup=`ls Main`**

*# Get each item from the list of dirs to back up*

*# tar and gzip the item with today's date appended.*

**for item in \$Dirs\_to\_Backup**

**do**

**tar zcvf \$item\_"\$today".tar.gz" \$item**

**Done**

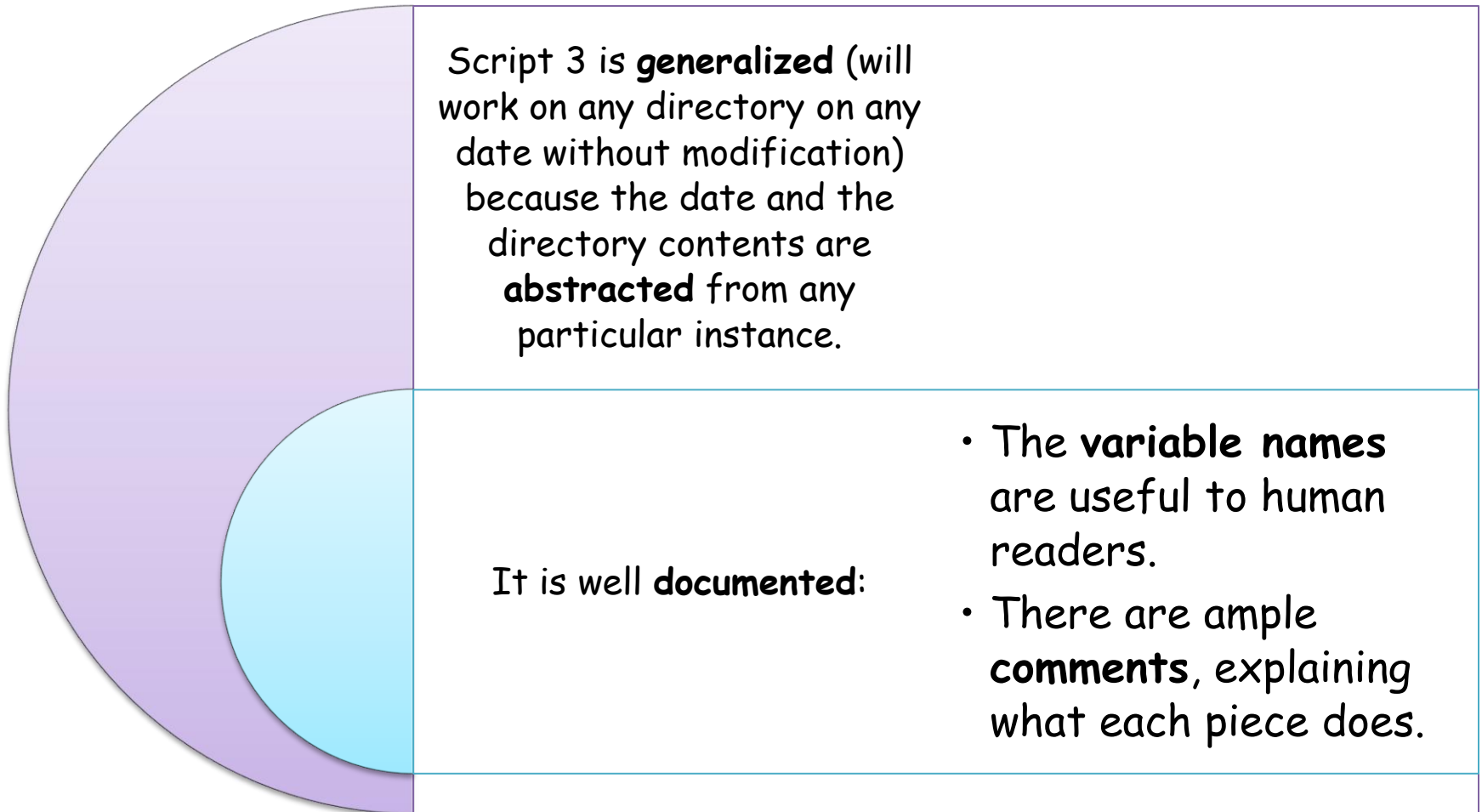
*# Make a dated directory for the gzipped files,*

*# and stuff the tar.gz files into it*

**mkdir "Backup\_"\$today**

**mv \*.tar.gz "Backup\_"\$today**

# Comparison: Script 3



# Summary

If you hire someone to write programs, you want them to know these principles and apply them.

In short, if you look at an example of something they've written, it should be more like script 3 than script 1 or 2.

If they have a compiled program, then hopefully, they've provided nice documentation for the end user.

# Modularity

Why is it better to have several small interacting programs than one giant program?

- It is easier to make mistakes and harder to find them in a giant program.
- You'll find you want subsections of the giant program for something else...the more you copy and paste subsections into other programs, the more you've created a nightmare when you find there is a basic flaw in some chunk of code.
- The best choice? Refactor your giant program into useful logical chunks. Each chunk can be reused by being called.

# Non-Modular

- You write some code that backs up the original image if it has more than 0 voxels, thresholds the image, segments it, renames the segmentations, dilates the second segmentation, erodes the segmentation, cleans up any small clusters, runs a statistical procedure on it which it puts in a log file.
- Say this is 350 lines long.



- Several useful smaller programs:
  - **BackNon0**: backs up an image if it is not empty, and warns you if the image is empty: **>BackNon0 image1**
  - **Thresh**: thresholds an image at the stated level:  
**>Thresh image1 0.5**
  - **SegName**: segments an image, and renames the segmentations, displays them so you can check them:  
**>SegName image1**
  - **Dilate**: dilates an image the indicated number of times:  
**>Dilate image1 1**
  - **Erode**: erodes an image the indicated number of times:  
**>Erode image1 1**
  - **ClusterClean**: cleans up any clusters smaller than the indicated size: **>ClusterClean image1 55**
  - **StatsMean**: runs a statistical procedure on the input image and logs the results in a log file that includes the image name: **>StatsMean image1**

# Modular Example

>Newbigfile spgr

\$1=input #always call the image "input"

BackNon0 \$input #back it up if it is not empty

Thresh \$input 0.03 # threshold to remove background

SegName \$input #segment, name wm output \${input}\_wm

Dilate \${input}\_wm #dilate the wm segmentation

Erode \${input}\_wm #erode the wm segmentation

ClusterClean \${input}\_wm 65 #clean up small clusters

StatsMean \${input}\_wm #run stats and log results

# Advantages of Modularity

It is easier to understand

It is easier to modify the small programs for 2 reasons

It is easier to create the "big" programs, because now they are short and easy to understand

Being short helps

The logic for that program appears in only one place, so you don't have to go searching for copies of it in lots of files.

# Summary: Modularity

It is better to build many small reusable interacting programs.

It is better because you can understand short programs more easily.

It is better because it makes fixing mistakes easier than "copy and paste".

System Administrator

Software engineer

Web site

Webapp

Desktop app

Programmer

Software Life Cycle

Pseudocode

Generalization/Abstraction

Variable Names

Comments

Modularity