CSC301: Introduction to Software Engineering

Lecture 6

Slides By: Wael Aboulsaadat

Based on Slides by Many Others. Acknowledgment due for all those who agreed to Contribute Slides.

Software Design: designing for change!

1. Avoid creating an object by specifying a class explicitly

- Bad

```
Employee emp;
X emp = new Employee();
```

- You are committing yourself to a specific implementation…

- Patterns: abstract factory, factory method, prototype

2. Avoid dependence on specific operations

- Bad

```
EmailSender emailSender;
emailSender = new EmailSender();
X emailSender.sendEmail(….);
```

- You are committing yourself to one way of satisfying a request
- Patterns: chain of responsibility, command

3. Avoid dependence on hardware and software platforms

- Bad

```
MSWindowsCoolToolbar toolbar;
toolbar = new X MSWindowsCoolToolbar();
toolbar.addIcon( new X MSWindowsCoolIcon (….) );
```

- You are committing yourself to a specific hardware..

- Patterns: abstract factory, bridge

4. Avoid dependence on object representations or implementations

- Bad

  X int arrNums[];

  X arrNums = new int[];

  X arrNums[0] = 10;

- Clients that know how an object is implemented might need to be changed when the object changes

- Patterns: abstract factory, bridge, memento, proxy

5. Avoid algorithmic dependencies

- Bad

  X BubbleSort bSort;
  bSort.sort(narr);

- Algorithms are often changed.

- Patterns: builder, iterator, strategy, template method, visitor

6. Avoid tight coupling

- Bad

```
Employee emp;
Address    address;
X address = new Address( emp , ….);
X emp = new Employee( address );
X emp.setCreditCard( new CreditCard( emp, address, …) );
```

- Tight coupling leads to monolithic system

- Patterns: abstract factory, bridge, chain of responsibility, command, façade, mediator, observer

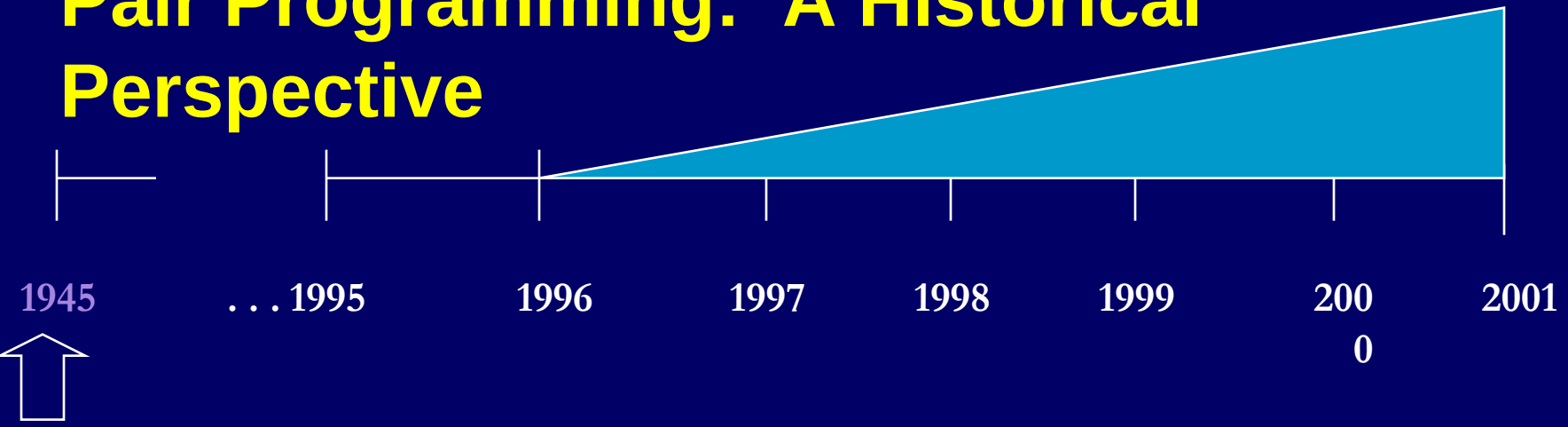
7. Avoid extending functionality by sub-classing…

- Customizing an object by sub-classing has drawbacks. Don't do it unless you are really specializing the parent class.
- Patterns: bridge, chain of responsibility, composite, decorator, observer, strategy

Pair Programming

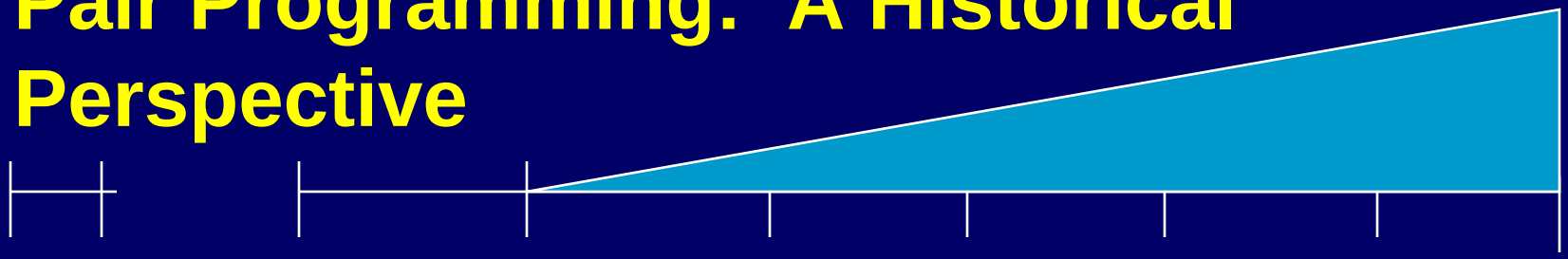
Pair Programming

Pair Programming: A Historical Perspective



John von Neumann, the great mathematician and creator of the von Neumann computer architecture, recognized his own inadequacies and continuously asked others to review his work.

Pair Programming: A Historical Perspective

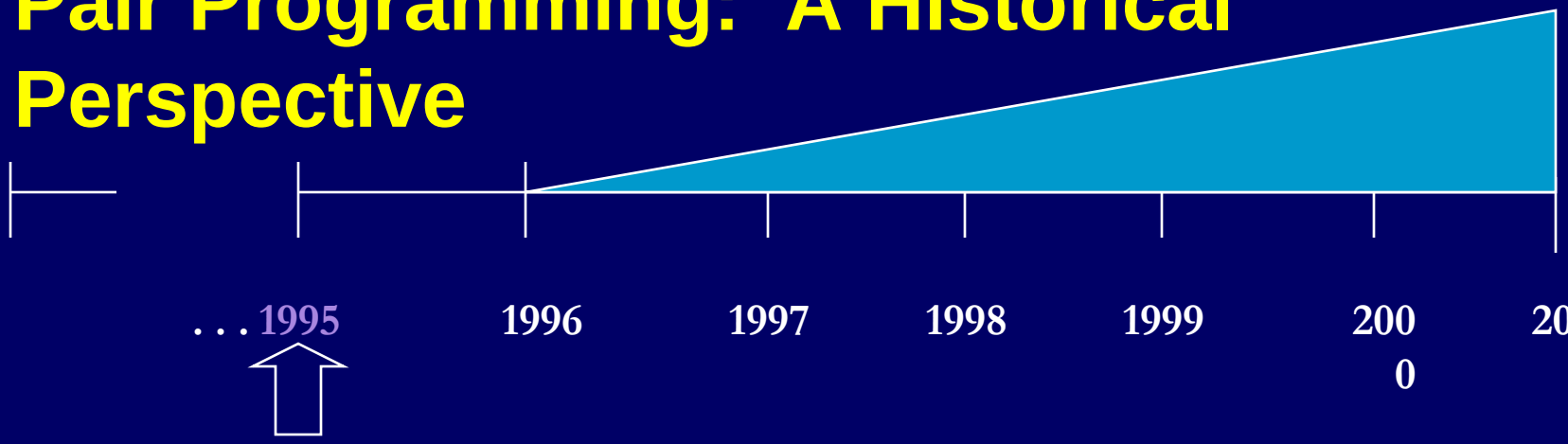


Fred Brooks and many others are pair programming, though they don't know there is a name for it.

"*Fellow graduate student Bill Wright and I first tried pair-programming when I was a grad student. We produced 1500 lines of defect-free code; it ran correctly first try.*

I encourage my own computer architecture students to work in pairs on all design projects. I am firmly convinced that it is not only as productive per person-hour, but also much more educational."

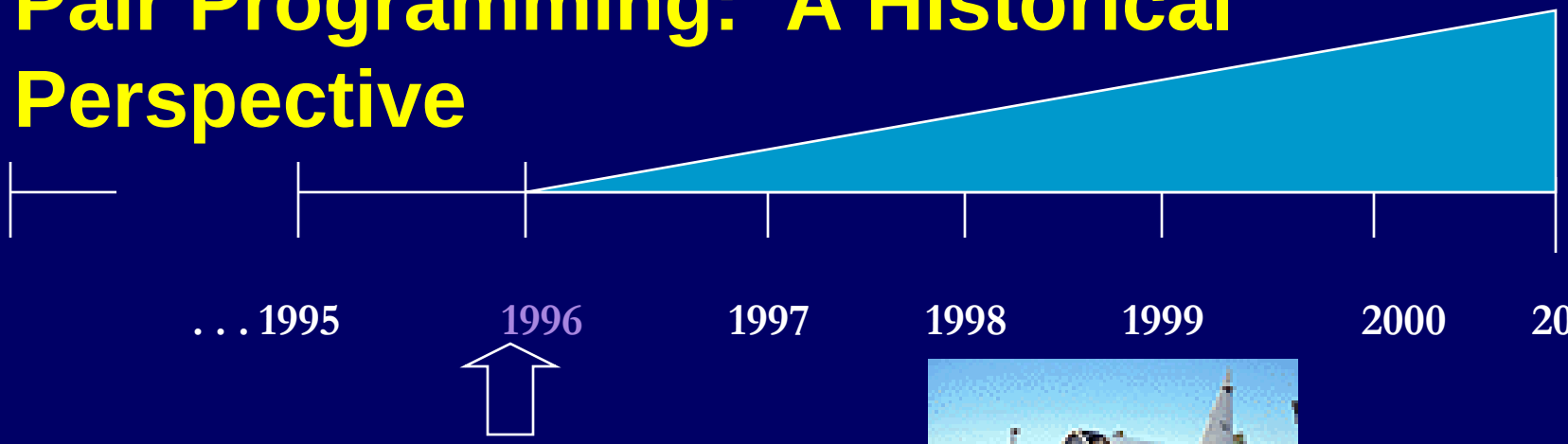
Pair Programming: A Historical Perspective



In "Constantine on Peopleware," Larry Constantine writes about Dynamic Duos producing code faster and more defect-free

Jim Coplien writes a "Developing in Pairs" Organizational Pattern . . . "together, they can produce more than the sum of the two individually."

Pair Programming: A Historical Perspective



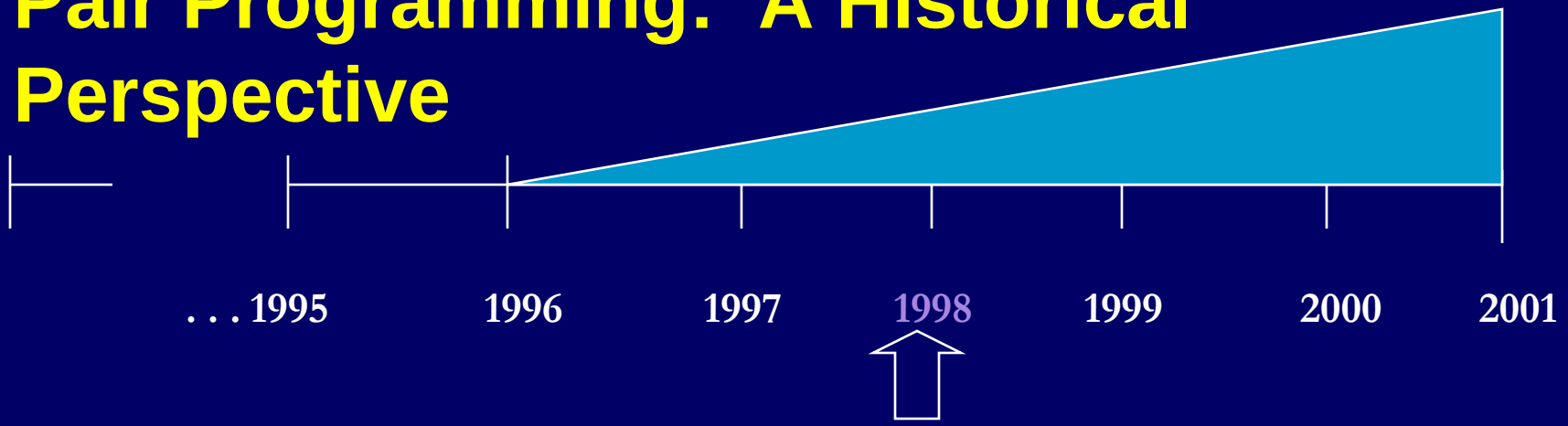
Hill Air Force Base:

"Total productivity was 175 lines per person-month (lppm) compared to a documented average individual productivity of only 77 lppm . . .

The error rate through software-system integration was three orders of magnitude lower than the organization's norm . . .

A brief list of observed phenomena includes focused energy, brainstorming, problem solving, continuous design and code walkthroughs, mentoring and motivation."

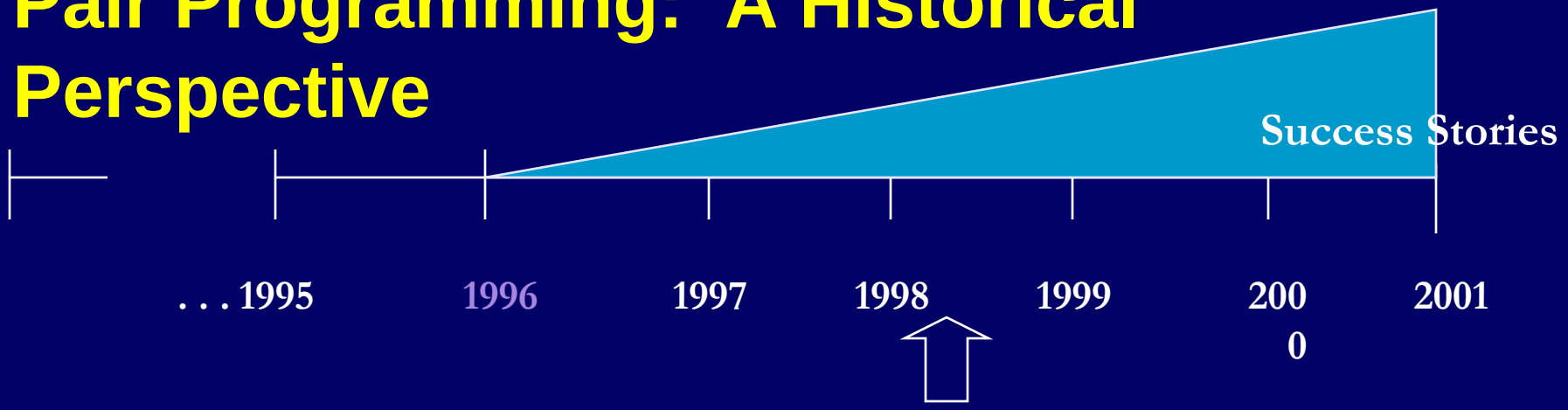
Pair Programming: A Historical Perspective



Temple University Professor Nosek runs (small scale) experiment:

- Pairs spent 60% more minutes on the task (but faster cycle time)
- Pairs produced better algorithms
- Pairs enjoyed the problem solving process more

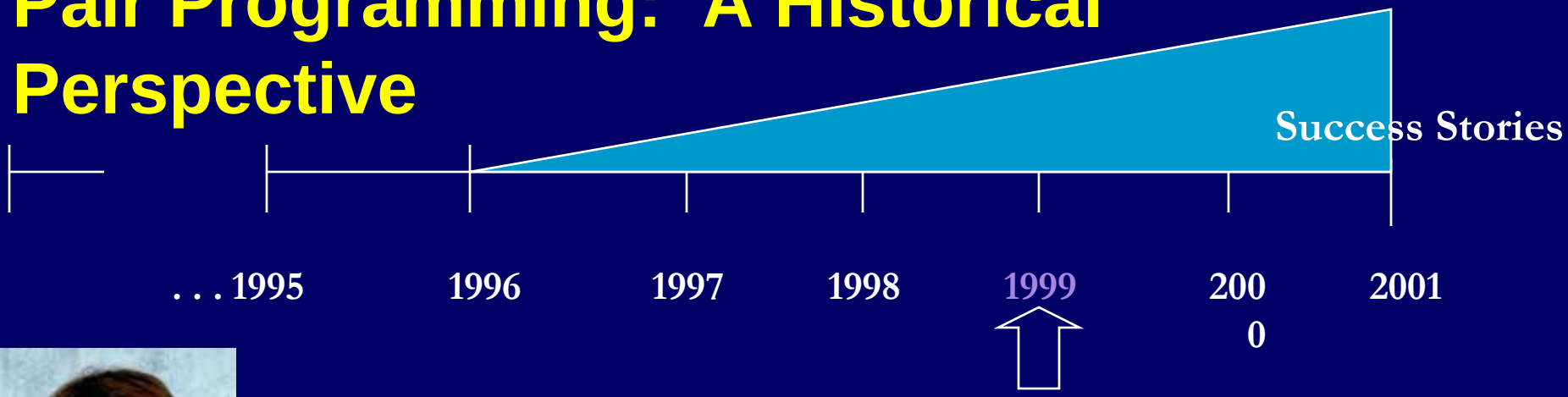
Pair Programming: A Historical Perspective



Wiki

- "Get two people programming in pairs, and they'll work more than twice as fast as one could have."
- "One of the rules of the Chrysler Comprehensive Compensation team is that all production code be written with a partner. As a testimonial, in the last six months before launching, the only code that caused problems was code written solo."

Pair Programming: A Historical Perspective

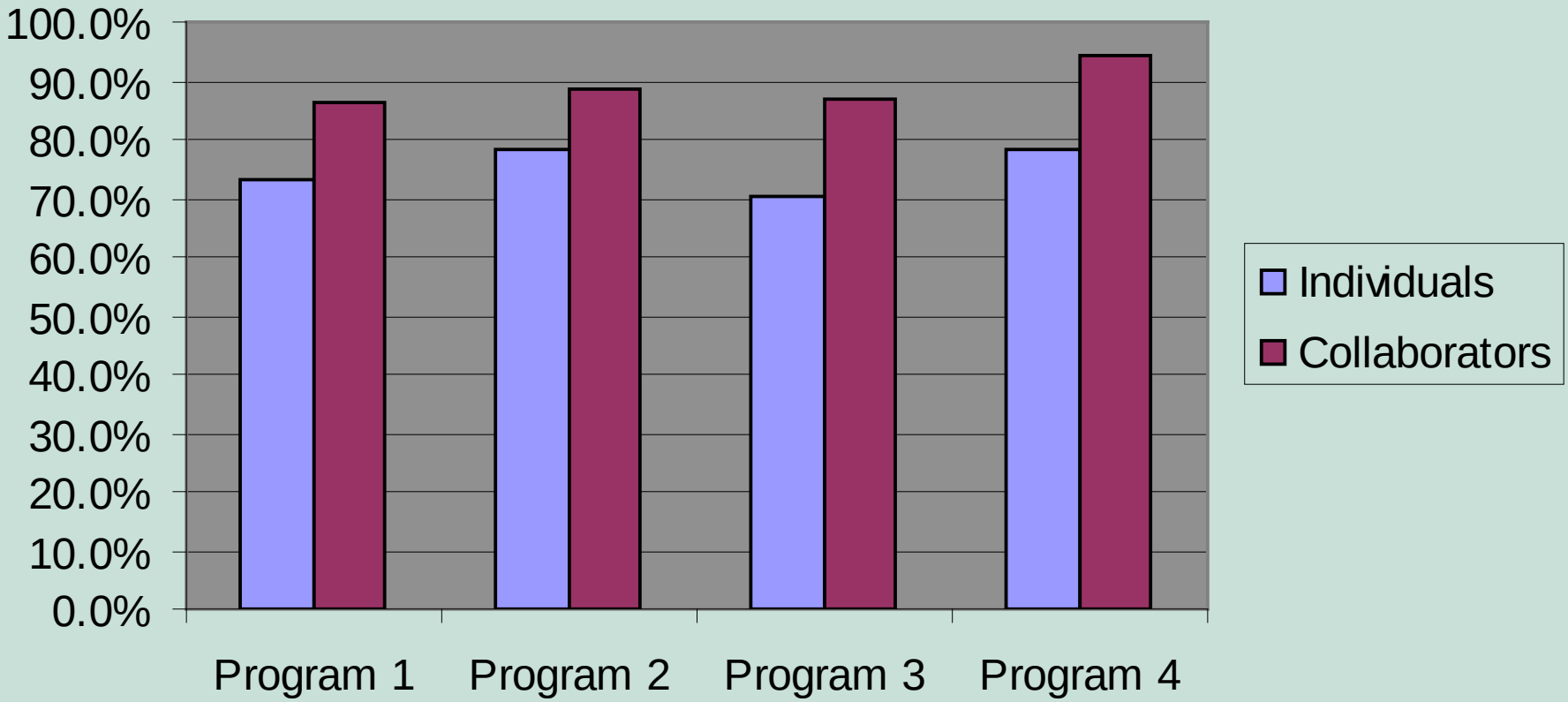


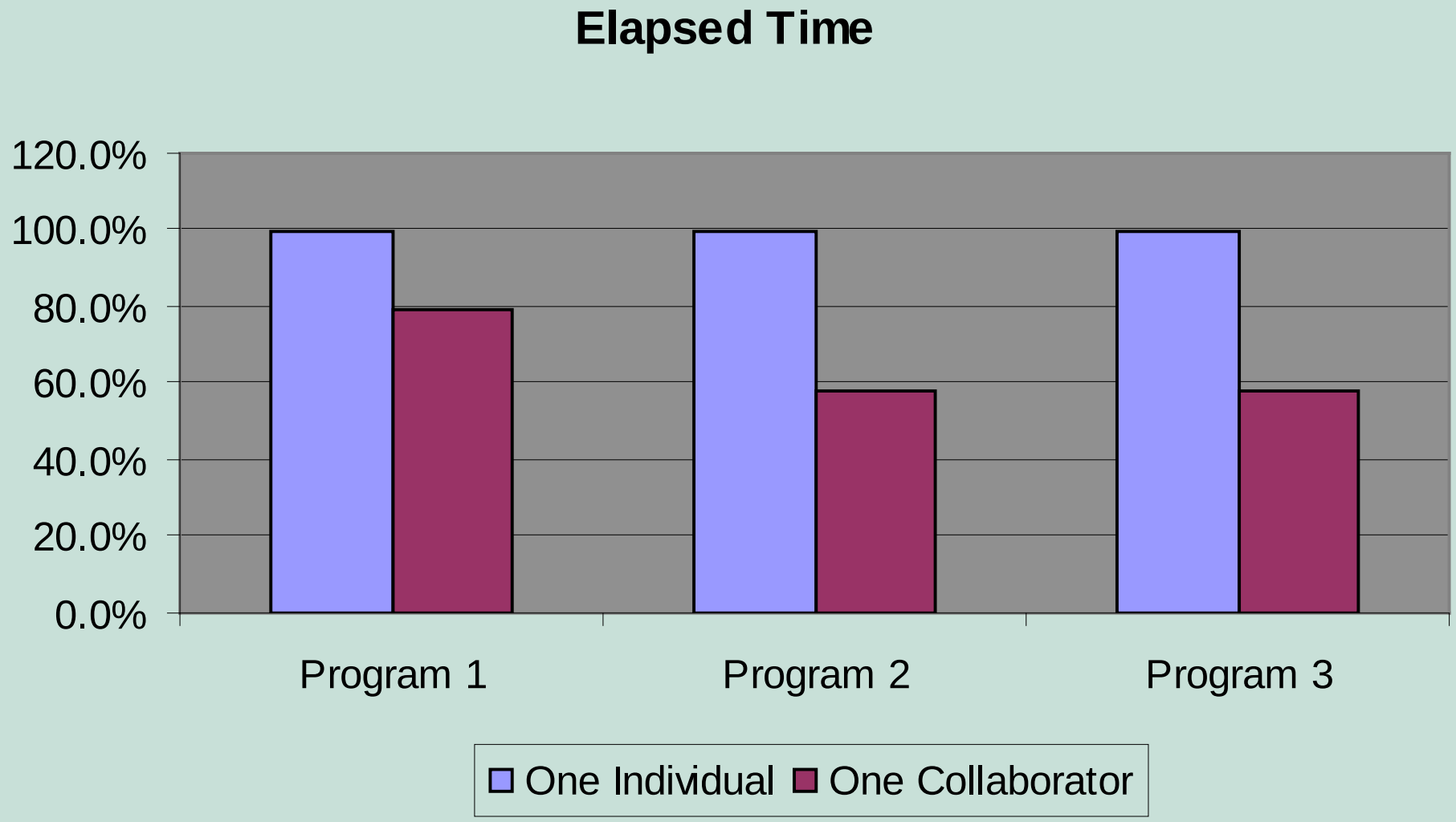
- Alistair Cockburn & University of Utah experiments

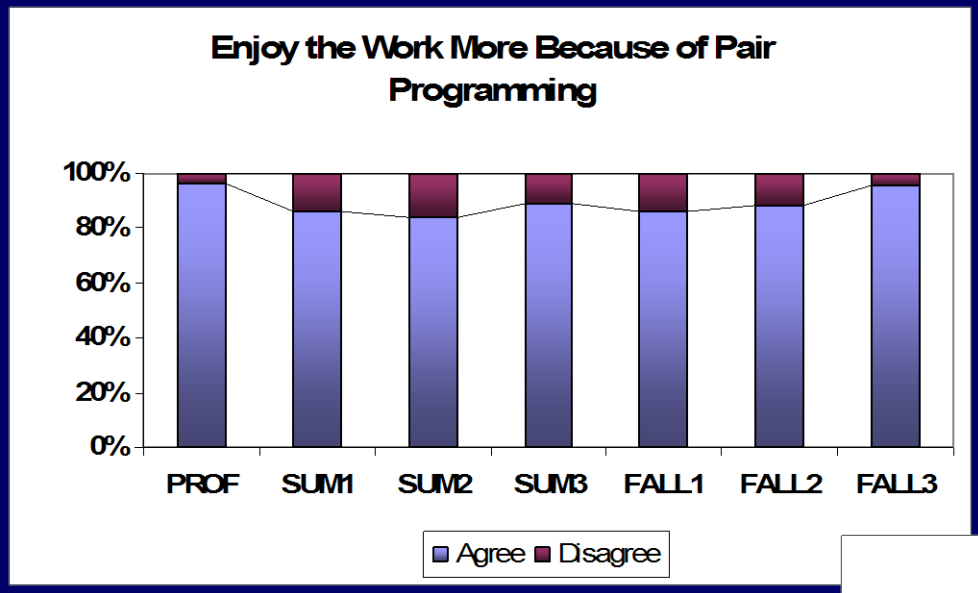
Empirical Study for Validation

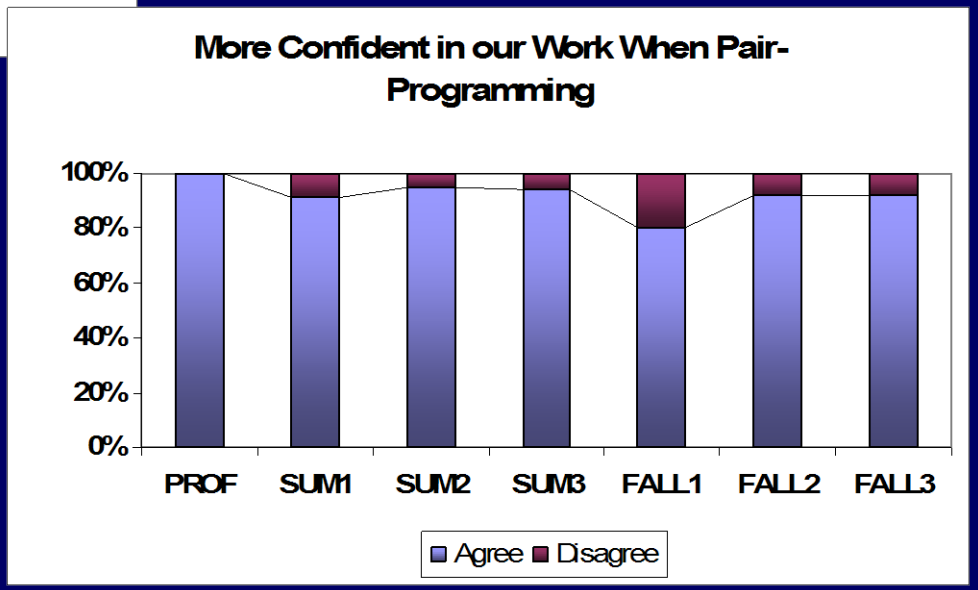
- Practice: Summer 1999
  - 20 Students (Sophomore/Junior)
    - All worked collaboratively
  - Generated more anecdotal/qualitative evidence

- Solo vs Pair: Fall 1999
  - 41 Students (Junior/Senior)
    - 28 Worked Collaboratively
    - 13 Worked Individually
  - Software development process was controlled
    - The only experimental variable: pair-programming
  - Quantitative: Time, Quality, Enjoyment, Confidence


Post Development Test Cases Passed
100.0%
90.0%
80.0%
70.0%
60.0%
50.0%
40.0%
30.0%
20.0%
10.0%
0.0%
Program 1
Program 2
Program 3
Program 4
Individuals
Collaborators


Elapsed Time
120.0%
100.0%
80.0%
60.0%
40.0%
20.0%
0.0%
Program 1
Program 2
Program 3
One Individual
One Collaborator


Enjoy the Work More Because of Pair Programming
100%
80%
60%
40%
20%
0%
PROF
SUM1
SUM2
SUM3
FALL1
FALL2
FALL3
Agree
Disagree


More Confident in our Work When Pair-Programming
100%
80%
60%
40%
20%
0%
PROF
SUM1
SUM2
SUM3
FALL1
FALL2
FALL3
Agree
Disagree

Research Findings to Date

- Strong anecdotal evidence from industry
  - "We can produce near defect-free code in less than half the time."
- Empirical Study
  - Pairs produced higher quality code
    - 15% less defects (difference statistically significant)
  - Pairs completed their tasks in about half the time
    - 58% of elapsed time (difference not statistically significant)
  - Most programmers reluctantly embark on pair programming
    - Pairs enjoy their work more (92%)
    - Pairs feel more confident in their work products (96%)

Why does it work?

- Pair-Pressure
  - Keep each other on task and focused
  - Don't want to let partner down
  - "Embarrassed" to not follow the prescribed process
  - Parkinson's Law "Work expands to fill all available time."
- Pair-Negotiation
  - Distributed Cognition: "Searching Through Larger Spaces of Alternatives"
    - Have shared goals and plans
    - Bring different prior experiences to the task
    - Different access to task relevant information
    - Must negotiate a common shared of action
- Pair-Relaying
  - Each, in turn, contributes to the best of their knowledge and ability
  - Then, sit back and think while their partner fights on

Why does it work ?

- Pair-Reviews
  - Continuous design and code reviews
  - Ultimate in defect removal efficiency
  - Removes programmers distaste for reviews
    - 80% of all (solo) programmers don't do them regularly or at all
- Pair Debugging
- Pair-Learning
  - Continuous reviews → learn from partners techniques, knowledge of language, domain, etc.
  - "Between the two of us, we knew it or could figure it out"
  - Apprenticeship

Distributed Pair Programming

- Net Meeting
- Yahoo Messenger
- Many more tools

Pair Programming Partner Picking Principles

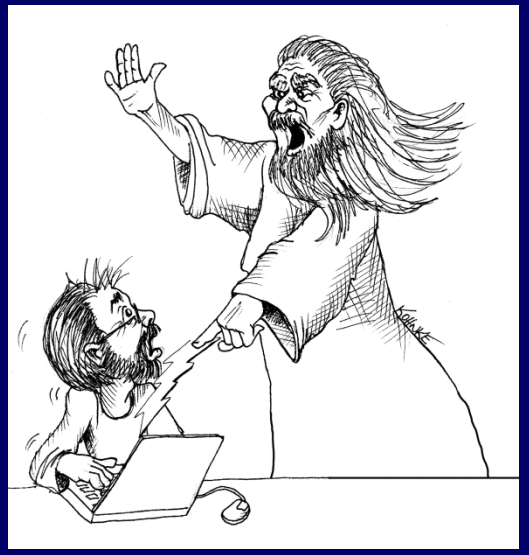
- Beneficial Pairs
  - Expert-Expert
  - Expert-Novice
  - Novice-Novice
- Problem Pairs
  - The Professional Driver
  - Excess Ego
  - Too Little Ego
- Non-issue Pairs
  - Gender
  - Culture

Expert/Expert

- Intent – To get the most complex job done well
- Success Keys
  - R E S P E C T!!!!!
  - Warp speed (no time explaining things)
  - Intense/Exhausting – Joking around helps
  - Even experts in different areas works
  - Each learns new stuff

- Warnings
  - Use expert-expert as your ultimate resort to fix a hard problem
  - Watch out for ego issues

Expert/Novice



- Intent – To get the easier job done,
  while training and integrating the novice
- Success Keys
  - Why ever do this? (Waste of expert's time)
    - TRAINING
    - Novice can watch decisions being made
    - Novice sees the processes being used
    - Helps to be assimilated into the collective
    - Master/apprentice is VERY POWERFUL
  - Amazingly – novice helps the expert find mistakes
    - Slowing down to answer questions and often asks 'WHY?'
- Warnings
  - Expert must be a "teacher" and be willing to be a teacher
  - Must have PATIENCE!
  - Must create an environment that is non-threatening to the novice

This is NOT Pair Programming

http://www.youtube.com/watch?v=_sOscpW1zus

Novice/Novice

- Intent – To produce production code in a relatively non-complex area of the project, gaining valuable experience for both in the process.
- Success Keys
  - Good technique for learning
    - What one doesn't know, the other might know or can look up
    - Can reduce time required by a supervisor
  - Each novice can educate the other in their particular area of specialty
  - Tend to not just struggle (as a single often does), if they both don't know it, they will ask someone more experienced
- Warnings
  - Instructor, coach or mentor is REQUIRED!!!
  - A few people say they don't do it because novices spin their wheels
    - better than solo novices…

Pro Driver Anti-Pattern

- Root Cause – Desire for power; driver's lack of confidence in navigator; or navigator's lack of confidence in self.
- General Form
  - Driver ALWAYS at the keyboard
  - Navigator feels disjoint, out of the loop, or unimportant
  - Gets worst if the driver doesn't listen to the navigator
    - This can kill the pairing!!
  - An "innocent" pro driver can happen when the navigator becomes a professional navigator and lacks the self confidence to drive
  - A "good" professional driver is covering for a physical or other limitation of the navigator – NO PROBLEM
  - Or under a tight deadline and the driver "knows" the tools
- Refactored Solution
  - Teach the driver to give up control
    - Can be very difficult
    - Have driver observe a good driver/navigator pair
  - Get navigator to drive

Bob and Laurie – Expert/Expert Pairing

```
/**
 * Composes and sends a message to Broker requesting that a
 * room be scheduled.
 * @param room - the room to be scheduled
 */
 protected List scheduleRoom(Place room) {
    //set originating AID as the one from the agent
    AID fromAID = this.agent.getAID();
    //set the destination AID as the one of the Broker
    AID toAID = new AID(this.BROKER_NAME, AID.ISLOCALNAME);
    // Store the actions
    /*** START HERE

 }
```

Bob and Laurie – Expert/Expert Pairing

Bob *(typing with two fingers)*: If we utilize a list of components here, we will be able easily scan them in the execution phase. *(He quickly types*: List list = new Vector(1);

list.add(action);

Laurie *(watching)*: Yes, but if you use an ArrayList, we'll be able to manage it later on in the next version.

Bob *(typing)*: Good idea. *(He quickly changes it to an ArrayList.)*

Laurie *(watching)*: Wait, you changed two lines instead of one and you deleted an important line.

Bob *(typing)*: Damn, I did it again. My RSI is acting up again. *(He hits control Z to undo the deletion.)*

Laurie *(laughing)*: You were the world's worst typist before the RSI. I can't stand to watch you type!!! Let me drive.

Bob *(mumbles to himself)*: Maybe we can get this speech input thing to work on code?

Laurie *(screaming)*: Arrrrrgggggghhhhhh. Give me the keyboard!!!

Hera (God) and Plutonius (Scared Lamb) – Expert/Novice Pairing

```
        ...
        boolean result=false;
        EventHandler t=null;
        journal.printInfo(3, getName() + " checking");
        Enumeration eventHandlerList = eventHandlers.elements();
        // Check all non-default transitions.
        while (eventHandlerList.hasMoreElements() & !result) {
            t=(EventHandler)eventHandlerList.nextElement();
            result=t.eventMatch(e);
        };
        /*** START HERE



    }
```

Hera (God) and Plutonius (Scared Lamb) – Expert/Novice Pairing

Hera: x = frobnatz.bar(1, y,
Pl – meekly: excuse me
Hera: WHAT – keeps typing: errorNum++,
Pl – cowering: Why did you type a space after the comma and before the variable 'y'?
Hera: That is how I always do it. *(continues typing)* 7);
Pl - Why?
Hera: BECAUSE, THAT IS HOW I ALWAYS DO IT!!! Now it is your turn to drive.
Pl - z = frobnatz.mumble(3,4
Hera *(screams)*: Noooooo!!!! *(and whacks Pl in the back of the head)*
Pl *(whimpering and typing even slower)*: <backspace> <space> 4);

Danny and Julie – Novice/Novice Pairing

```
/**
 * Basic main test program
 * @param args – command line arguments
 */
public static void main(String[] args) {

  /*** START HERE

} //~ END main()
```

Danny and Julie – Novice/Novice Pairing

First Newbie *(driving)*: int x;

Second Newbie *(watching)*: Wait, why are you using x for the variable name?

First Newbie: Well, it is the name that I just thought of.

Second Newbie: But why not use y? I like y's much better than x's. X's remind me of that XOXO thing that means kiss hug kiss hug. Yuck.

First Newbie: Hum, I never thought of that, ok let's use y's. *(typing commences replacing the variable name with y)*

Second Newbie *(watching again)*: Wait, if we use y, someone might think of the word "WHY".

First Newbie *(thinking hard)*: Yes, you are absolutely correct. Let's use z. I can't think of anything that will get confused with z. *(so the variable is changed again)*

Second Newbie *(watching and starts to laugh)*: Hey, I just remembered something.

First Newbie *(stops typing)*: What?

Second Newbie *(chuckling)*: In my Software Practice class, Professor Kessler told us to choose variable names that actually meant something. I completely forgot about that. You know, that was a pretty good class.

First Newbie *(remembering fondly)*: Yes, I took it the year before you did and we discussed the same thing. I completely forgot that too. Maybe we should choose a better name.

Second Newbie *(thinking hard again)*: Yes, but what makes sense here?

Pro and Rookie – Pro Driver Pairing

```
private Place selectRoom(ACLMessage msg) {
    Object obj;
    Place room = null;
    ExpOneOf instanceCollection;
    Iterator instances;
    List list = null;
    try {
        list = agent.extractContent(msg);
    } catch(Exception ex) {
        agent.printError("Content extraction error");
    }
    /**** START HERE



    return room;
}
```

Pro and Rookie – Pro Driver Pairing

Pro Driver *(typing very fast)*: Ah grasshopper, if you can take the keyboard from my hand, then you will be the master.

Grasshopper *(reaches for the keyboard)*

Pro Driver *(deftly moving the keyboard to the left while still typing)*: You must be much quicker than that grasshopper.

Grasshopper *(observes the typing and tries a feint to the right and a grab from the left)*

Pro Driver *(grabs the keyboard by the escape key and lifts it up 3 inches)*

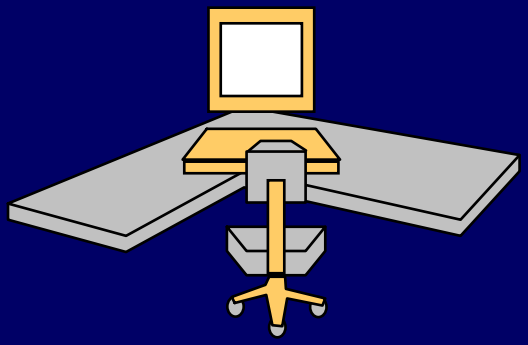
Grasshopper *(gets nothing but air)*: Damn.

Pro Driver *(typing even faster now, using his elbows)*: Grasshopper, that is the oldest trick in the book.

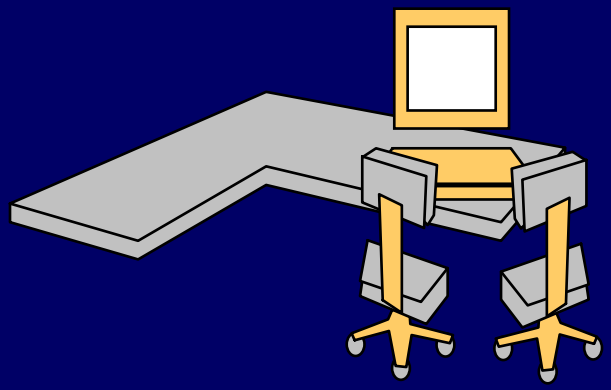
Grasshopper *(excitedly points)*: Hey, look at what Jerry left on his monitor!!

Pro Driver *(expertly typing with his feet)*: Grasshopper, no, I was mistaken, THAT is the oldest trick in the book.

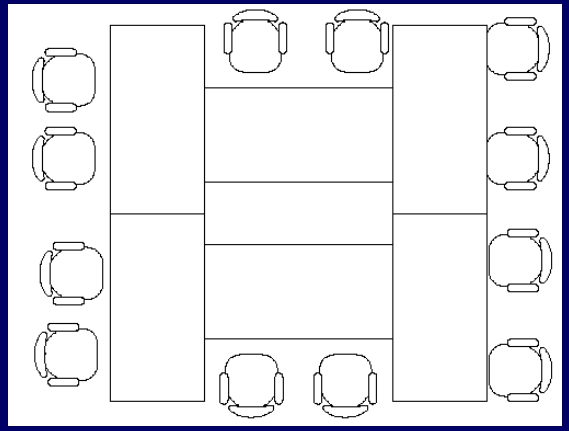
Workplace Layout



Bad



Better



Best (RoleModel XP Studio)

Code Reviews

Formal Code reviews

- *Code presenter* is not the author of the code.
- The other participants are the *inspectors* & coder
- There is a *moderator* to assure that the rules are followed and the meeting runs smoothly.
- After the inspection a report is composed. The programmer then makes changes and a re-inspection occurs, if necessary.
- Formal code inspections are effective at finding bugs in code and designs and are gaining in popularity.

Code reviews

- Reviewer:
  - Ask questions rather than make statements
  - Avoid the "Why" questions.
  - Remember to praise
  - Make sure you have good coding standards to reference
  - Make sure the discussion stays focused on the code and not the coder
  - Remember that there is often more than one way to approach a solution

3. Communication: code reviews

- Coder:
  - Remember that the code isn't you
  - Create a checklist for yourself of the things that the code reviews tend to focus on
  - Help to maintain the coding standards

Code review checklist: Data reference errors

1. Is an un-initialized variable referenced?

2. Are array subscripts integer values? within the array's bounds?

3. Are there off-by-one errors in indexing operations or references to arrays?

4. Is a variable used where a constant would work better?

5. Is a variable assigned a value that's of a different type than the variable?
6. Is memory allocated for referenced pointers?

7. Are data structures that are referenced in different functions defined identically?

Code review checklist: Data declaration errors

8. Are the variables assigned the correct length, type, storage class?
   - E.g. should a variable be declared a string instead of an array of characters?

9. If a variable is initialized at its declaration, is it properly initialized and consistent with its type?

10. Are there any variable with similar names?

11. Are there any variables declared that are never referenced or just referenced once (should be a constant)?

12. Are all variables explicitly declared within a specific module?

Code review checklist: Computation errors

13. Do any calculations that use variables have different data types?
    - E.g., add a floating-point number to an integer
14. Do any calculations that use variables have the same data type but are different size?
    - E.g., add a long integer to a short integer
15. Are the compiler's conversion rules for variables of inconsistent type or size understood?
16. Is overflow or underflow in the middle of a numeric calculation possible?
17. Is it ever possible for a divisor/modulus to be 0?
18. Can a variable's value go outside its meaningful range?
    - E.g., can a probability be less than 0% or greater than 100%?
19. Are parentheses needed to clarify operator presence rules?

Code review checklist: Comparison errors

20. Are the comparisons correct?
    - E.g., < instead of <=

21. Are there comparisons between floating-point values?
    - E.g., is 1.0000001 close enough to 1.0000002 to be equal?

22. Are the operands of a Boolean operator Boolean?
    - E.g., in C 0 is false and non-0 is true

Code review checklist: Control flow errors

23. Do the loops terminate? If not, is that by design?

24. Does every `switch` statement have a `default` clause?

25. Are there `switch` statements nested in loops?

    - E.g., careful because `break` statements in `switch` statements will not exit the loop … but break statements not in `switch` statements will exit the loop.

26. Is it possible that a loop never executes? Is it acceptable if it doesn't?

27. Does the compiler support short-circuiting in expression evaluation?

Code review checklist: Subroutine parameter errors

28. If constants are passed to the subroutine as arguments are they accidentally changed in the subroutine?

29. Do the units of each parameter match the units of each corresponding argument?
  - E.g., English versus metric
  - This is especially pertinent for SOA components

30 .Do the types and sizes of the parameters received by a subroutine match those sent by the calling code?

Code review checklist: Input/Output errors

31. If the file or peripheral is not ready, is that error condition handled?

32. Does the software handle the situation of the external device being disconnected?

33. Have all error messages been checked for correctness, appropriateness, grammar, and spelling?

34. Are all exceptions handled by some part of the code?

35. Does the software adhere to the specified format of the date being read from or written to the external device?

Software Engineering Rules

1) Brook's Rule

➔ Adding people to a late project makes it later

- Because the people already in the project are now spending time getting new staff up to speed

- Implication:
  - If the project is behind:
    - Re-prioritize
    - Get other people to deal with distractions

2) Glass's Rule

➔ Any new tool or technique initially makes the adopter slower. .

- It's faster to do today's assignment in Notepad than it is to learn Emacs
- P.S.: 5-35% productivity improvement is the best you'll see (despite "order of magnitude" claims from vendors or inventors…)

- Implication:
  - It's only worth adopting new tools and techniques if you're willing to be patient

3) False Alerts Rule

➔ As the rate of erroneous alerts increases, operator reliance, or belief, in subsequent warnings decreases.

- Implication:
  - DO NOT cut on QA resources if number of actual bugs decreases

4) Hick's Law

➔ The time to make a decision is a function of the possible choices he or she has.

- Implication:
  - DO NOT hand developers vague requirements, why?

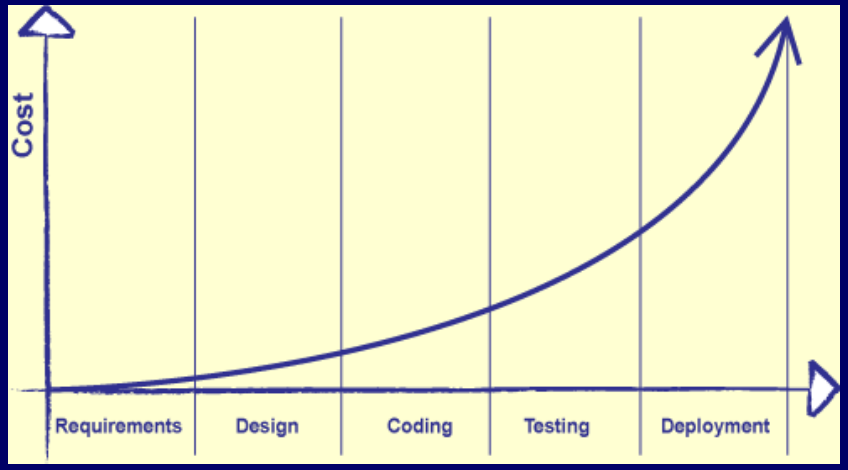
5) Lister's Law

➔ People under time pressure don't think faster!

- Implication:
  - Expect almost constant performance/output from your team

6) Boehm's Curve Rule

➔ Up to 100 times more expensive to fix requirements error in production than in early stages of development



- Implication:
  - Get user on board early

7) Sixty-sixty Rule

➔ Sixty percent of software's dollar is spent on maintenance, and sixty percent of that maintenance is enhancement.

- Implication:
  - Get user on board early

8) Maintenance Rules

- **Maintenance makes up 40-80% of the cost of a software project.**
  - finding has been validated many times since
  - The single largest cost in most projects, but almost always underestimated

- **Enhancement is roughly 60% of "maintenance".**
  - Enhancement usually a result of changing requirements
    - Yes, they keep changing after software is in production
  - How much effort goes into other kinds of maintenance?
    - 18%: adaptive maintenance (i.e., keeping up with a changing environment)
    - 17%: error correction, 5%: miscellaneous

8) Maintenance Rules – cont'd

- 30% of maintenance time is spent figuring out how the software actually works
  - This figure rises as the software ages

- **Better software engineering leads to more maintenance, not less.**
  - the better the system, the longer it will live, and the more changes are possible

8) Maintenance Rules – cont'd

- **Small changes have a higher error density than large ones.**
  - Small changes require the same level of program understanding as large ones
    - So mistakes are just as likely
    - So density is higher
  - This does *not* mean you should batch changes into bigger lumps!