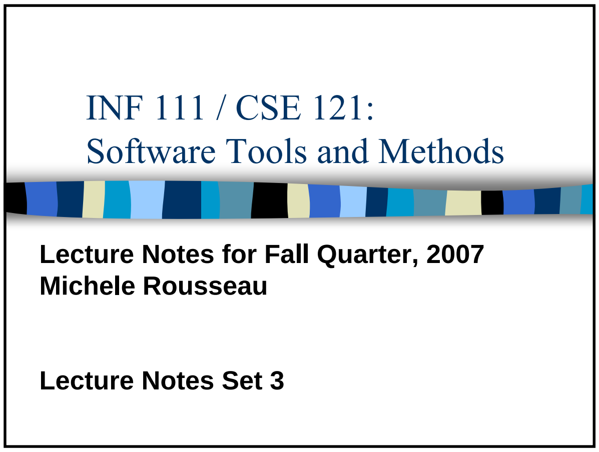
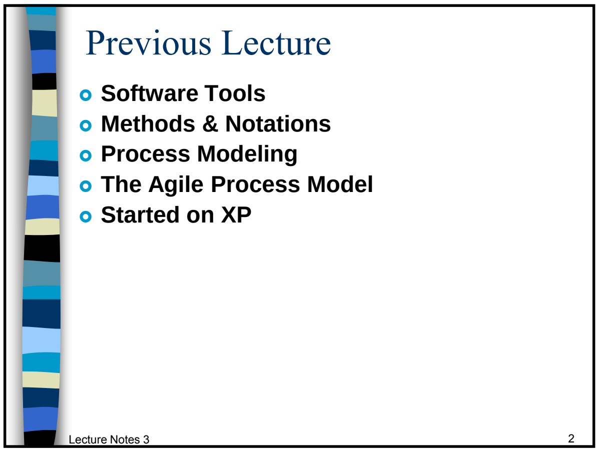


INF 111 / CSE 121: Software Tools and Methods

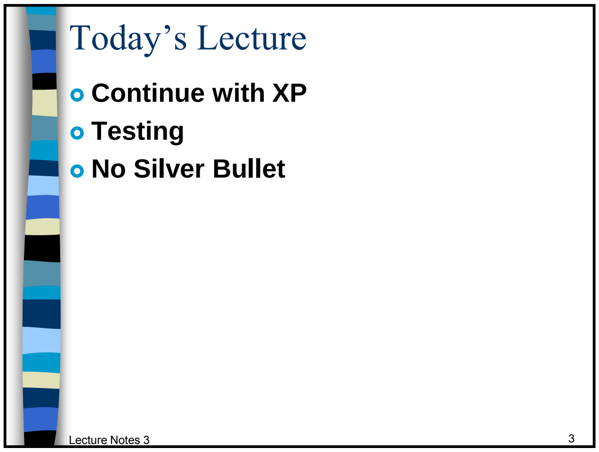
Lecture Notes for Fall Quarter, 2007
Michele Rousseau

Lecture Notes Set 3



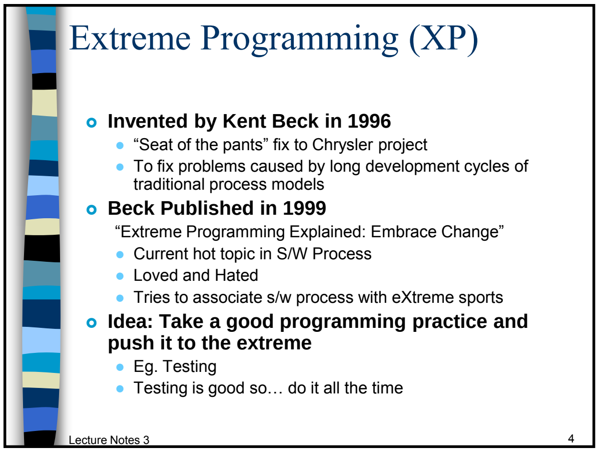
Previous Lecture

- **Software Tools**
- **Methods & Notations**
- **Process Modeling**
- **The Agile Process Model**
- **Started on XP**



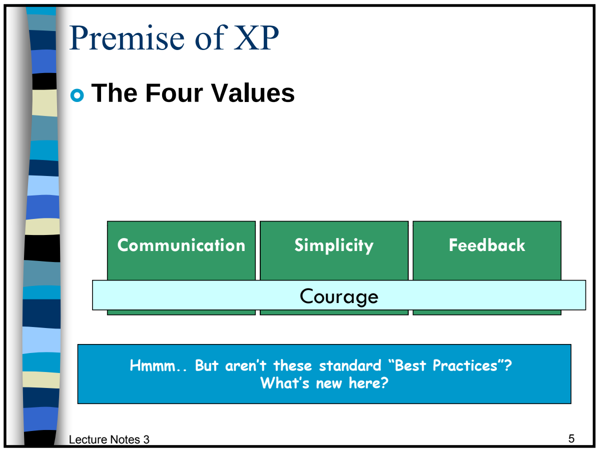
Today's Lecture

- **Continue with XP**
- **Testing**
- **No Silver Bullet**



Extreme Programming (XP)

- **Invented by Kent Beck in 1996**
  - "Seat of the pants" fix to Chrysler project
  - To fix problems caused by long development cycles of traditional process models
- **Beck Published in 1999**
  "Extreme Programming Explained: Embrace Change"
  - Current hot topic in S/W Process
  - Loved and Hated
  - Tries to associate s/w process with eXtreme sports
- **Idea: Take a good programming practice and push it to the extreme**
  - Eg. Testing
  - Testing is good so… do it all the time

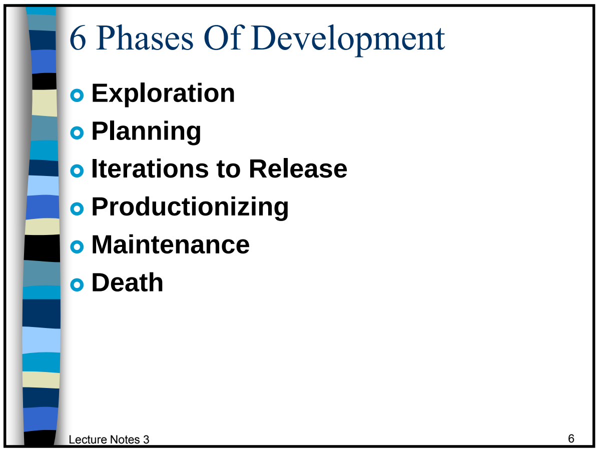


Premise of XP

- **The Four Values**

Communication
Simplicity
Feedback
Courage

Hmmm.. But aren't these standard "Best Practices"?
What's new here?



6 Phases Of Development

- **Exploration**
- **Planning**
- **Iterations to Release**
- **Productionizing**
- **Maintenance**
- **Death**

Exploration Phase

- **Customers**
  - Story Cards – 1 feature per card
    - Customer wish list for first release
- **Developers**
  - Get familiar with
    - Tools
    - Technology
    - Practices

    … to be used
  - Architecture possibilities explored – Prototype
  - Tailor process to the project
- **A few weeks to months**
  - How familiar is tech to programmers

Planning Phase

- **Prioritize Stories**
  - First Small release agreement
- **Effort Estimate for each story**
  - Schedule Agreement
    - Usually < 2 months
- **Takes a few days**

Iterations to Release Phase

- **Several Iterations before 1st Release**
- **# of Iterations determined in planning phase**
- **Each iteration takes 1-4 wks to implement**
- **Select stories wisely**
  - these enforce system architecture for the entire system
  - Customer chooses stories for each iteration
- **Functional tests created by Customer**
  - Run at the end of each iteration

At the end of last iteration → Production

Productionizing Phase

- **End testing before release**
- **New changes may be found**
  - Decide whether to include in current release
  - Documented for later implementation

    → Maintenance Phase
- **Iterations shortened**

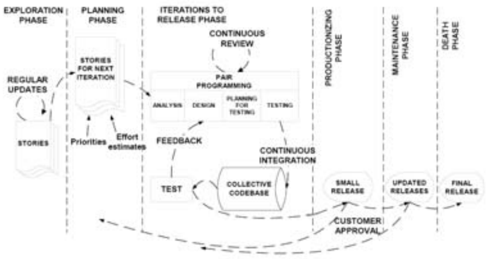
Maintenance and Death Phases

- **Maintenance**
  - May need more people
    - Maintain current production
    - Produce new Iterations
    - Change team structure
  - Development slows
- **Death Phase**

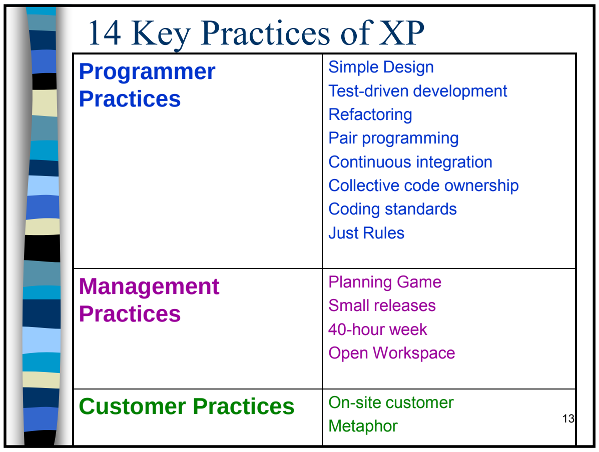
  Either…
  - All stories complete & quality is satisfactory
  - Not delivering expected outcomes
  - Too expensive to continue

XP Lifecycle Model

The life cycle of XP consists of five phases: Exploration, Planning, Iterations to Release, Productionizing, Maintenance and Death

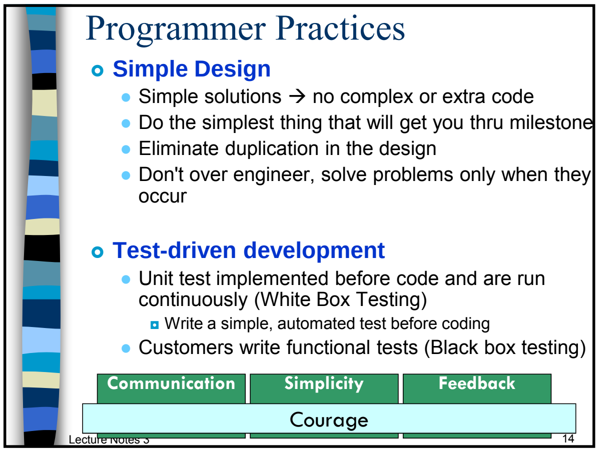


Life cycle of the XP process.



14 Key Practices of XP

Programmer Practices	Simple Design Test-driven development Refactoring Pair programming Continuous integration Collective code ownership Coding standards Just Rules
Management Practices	Planning Game Small releases 40-hour week Open Workspace
Customer Practices	On-site customer Metaphor

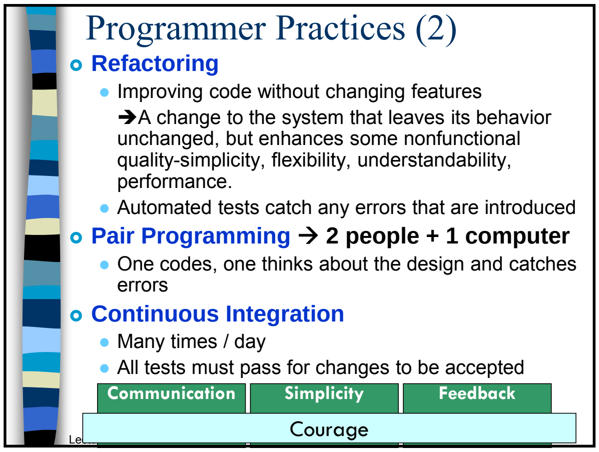


Programmer Practices

- **Simple Design**
  - Simple solutions → no complex or extra code
  - Do the simplest thing that will get you thru milestone
  - Eliminate duplication in the design
  - Don't over engineer, solve problems only when they occur
- **Test-driven development**
  - Unit test implemented before code and are run continuously (White Box Testing)
    - Write a simple, automated test before coding
  - Customers write functional tests (Black box testing)

Communication Simplicity Feedback

Courage

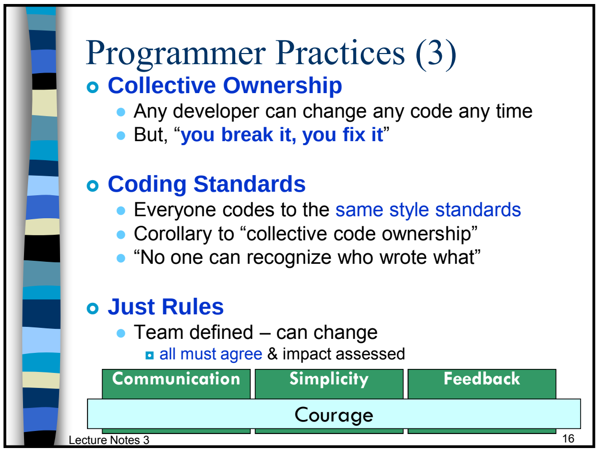


Programmer Practices (2)

- **Refactoring**
  - Improving code without changing features
    →A change to the system that leaves its behavior unchanged, but enhances some nonfunctional quality-simplicity, flexibility, understandability, performance.
  - Automated tests catch any errors that are introduced
- **Pair Programming → 2 people + 1 computer**
  - One codes, one thinks about the design and catches errors
- **Continuous Integration**
  - Many times / day
  - All tests must pass for changes to be accepted

Communication Simplicity Feedback

Courage

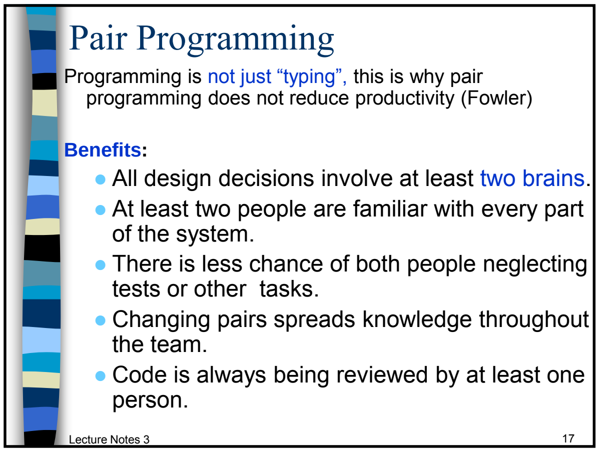


Programmer Practices (3)

- **Collective Ownership**
  - Any developer can change any code any time
  - But, "**you break it, you fix it**"
- **Coding Standards**
  - Everyone codes to the same style standards
  - Corollary to "collective code ownership"
  - "No one can recognize who wrote what"
- **Just Rules**
  - Team defined – can change
    - all must agree & impact assessed

Communication Simplicity Feedback

Courage

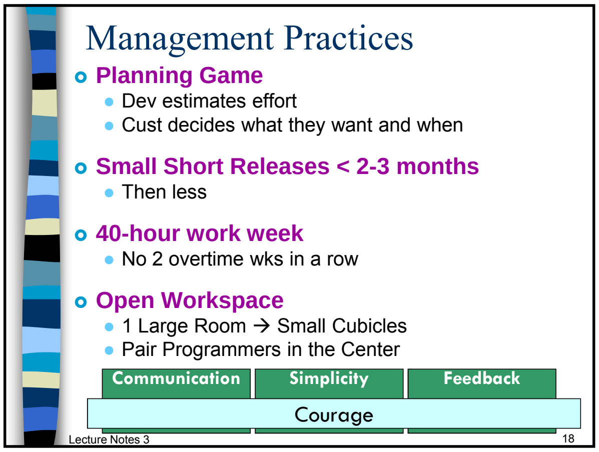


Pair Programming

Programming is not just "typing", this is why pair programming does not reduce productivity (Fowler)

Benefits:

- All design decisions involve at least two brains.
- At least two people are familiar with every part of the system.
- There is less chance of both people neglecting tests or other tasks.
- Changing pairs spreads knowledge throughout the team.
- Code is always being reviewed by at least one person.

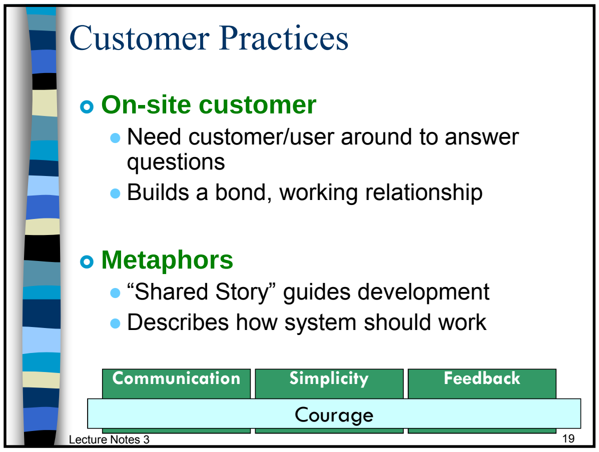


Management Practices

- **Planning Game**
  - Dev estimates effort
  - Cust decides what they want and when
- **Small Short Releases < 2-3 months**
  - Then less
- **40-hour work week**
  - No 2 overtime wks in a row
- **Open Workspace**
  - 1 Large Room → Small Cubicles
  - Pair Programmers in the Center

Communication Simplicity Feedback

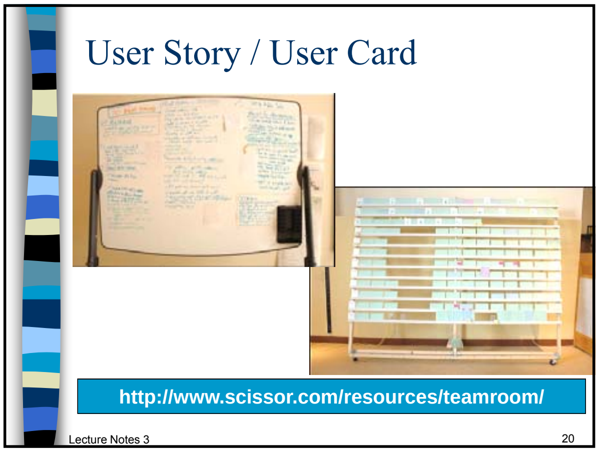
Courage



Customer Practices

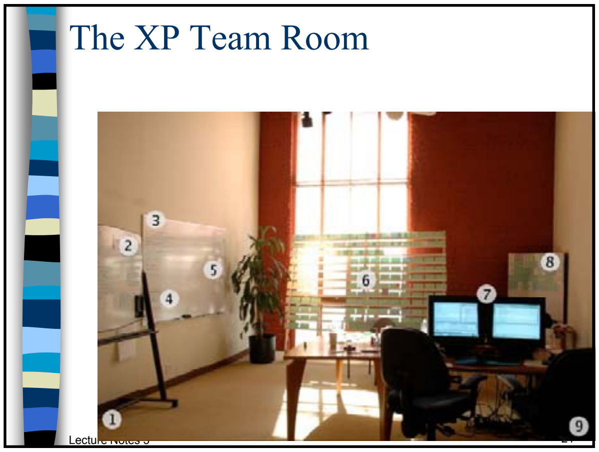
- **On-site customer**
  - Need customer/user around to answer questions
  - Builds a bond, working relationship
- **Metaphors**
  - "Shared Story" guides development
  - Describes how system should work

Communication
Simplicity
Feedback
Courage

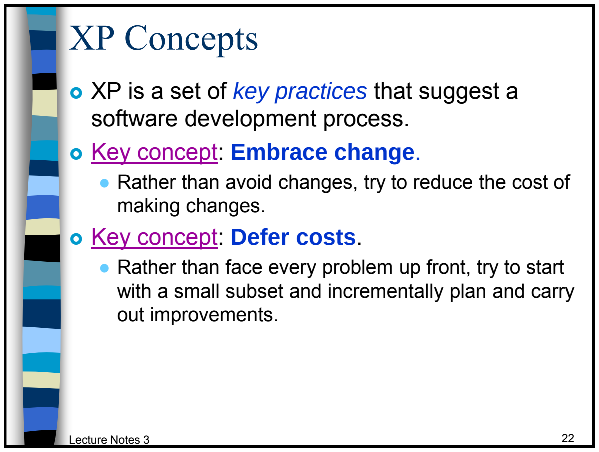


User Story / User Card

http://www.scissor.com/resources/teamroom/

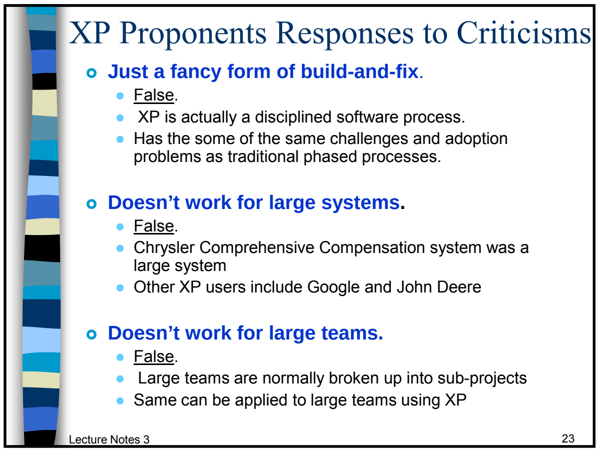


The XP Team Room



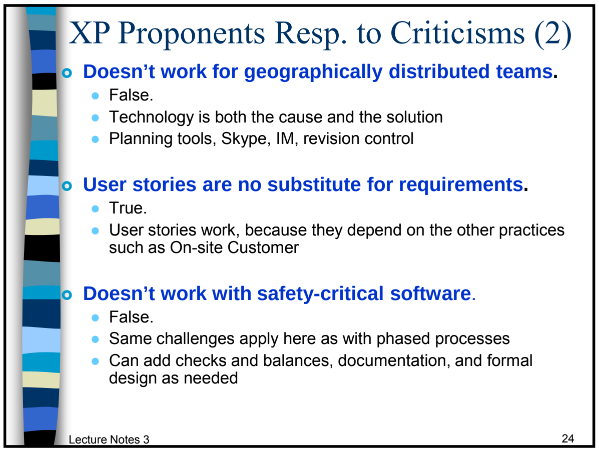
XP Concepts

- XP is a set of *key practices* that suggest a software development process.
- Key concept: **Embrace change**.
  - Rather than avoid changes, try to reduce the cost of making changes.
- Key concept: **Defer costs**.
  - Rather than face every problem up front, try to start with a small subset and incrementally plan and carry out improvements.



XP Proponents Responses to Criticisms

- **Just a fancy form of build-and-fix**.
  - False.
  - XP is actually a disciplined software process.
  - Has the some of the same challenges and adoption problems as traditional phased processes.
- **Doesn't work for large systems.**
  - False.
  - Chrysler Comprehensive Compensation system was a large system
  - Other XP users include Google and John Deere
- **Doesn't work for large teams.**
  - False.
  - Large teams are normally broken up into sub-projects
  - Same can be applied to large teams using XP



XP Proponents Resp. to Criticisms (2)

- **Doesn't work for geographically distributed teams.**
  - False.
  - Technology is both the cause and the solution
  - Planning tools, Skype, IM, revision control
- **User stories are no substitute for requirements.**
  - True.
  - User stories work, because they depend on the other practices such as On-site Customer
- **Doesn't work with safety-critical software**.
  - False.
  - Same challenges apply here as with phased processes
  - Can add checks and balances, documentation, and formal design as needed

XP Proponents Resp. to Criticisms (3)

- **Doesn't produce documentation.**
  - Maybe. XP only produces as much documentation as is needed, when it is needed (simplicity).
- **It is wasteful, because you're doing constantly doing re-design.**
  - False.
  - Planning everything up front is wasteful, because things are going to change anyways.
- **Not suitable for all projects**
  - True.
  - User functionality is simple, algorithms hard
  - Example: scientific applications

Productivity Gains

- **For a Web Dev Project**
  - 66% increase in new lines of code produced
  - 302% inc in new methods developed
  - 283% inc in # of new classes implemented

Maruer & Martel 2002b

Cons

- **Corp Culture must support XP**
  - Any resistance can lead to failure
- **Best for teams < 20**
- **Best if teams are collocated**
  - On the same floor
- **Technology that does not support "graceful change" → may not be suitable**

More Reading if you are interested

- **Agile**
  - Abrahamsson, P, et al. (2002). Agile software development methods: Review and analysis. VTT Publications 478.
  - http://www.vtt.fi/inf/pdf/publications/2002/P478.pdf
- **XP**
  - Beck, K. (1999). Extreme programming explained: Embrace change. Reading Mass., Addison-Wesley

Take a break!

- **Stretch, Relax**
- **Get some water, Use the restroom**
- **Get to know your classmates…**
- **Etc…..**

When we return…

- **No Silver Bullet**
- **Testing**

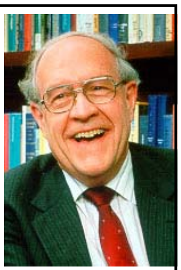
Moving on..

- **No Silver Bullet**
- **Testing**

The Mythical Man-Month

- **Originally Published in 1975**
  - Fred Brooks
  - Based on Experiences From OS/360 in mid-60's
- **So why should we care?**
- **Some interesting Stats**
  - Amazon.com Sales Rank:
    - #3,201 in Books
    - #1 in Microprocessor Design
    - #3 in Systems Analysis & Design
    - #12 in Software Engineering

Who is Fred Brooks?



- "Father of IBM OS/360"
- 1992 Computer Pioneer Award (IEEE)
- 1999 Turing award winner
- 2007 Harvard Centennial Medal
- Founded UNC-Chapel Hill CS dept

No-Silver Bullet

"There is *no single development,* in either technology or management technique, which by itself *promises even one order-of-magnitude improvement within a decade* in productivity, in reliability, in simplicity"

Essence & Accident

- **Essential Tasks**
  - Specifications, design & testing of conceptual constructs
- **Accidental (or incidental) Tasks**
  - Programming & Compiling

The essential tasks are the hard part.

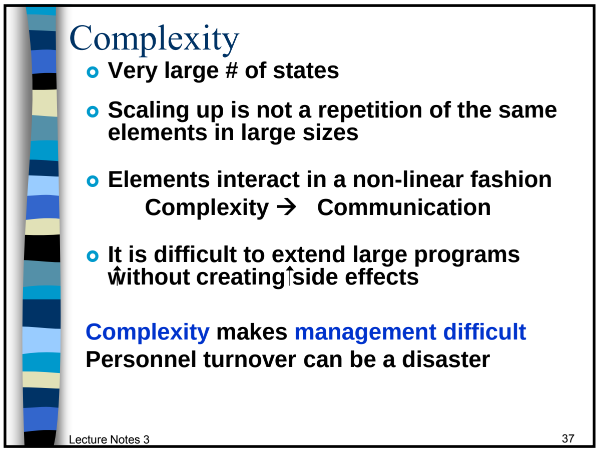
Why is building s/w difficult?

"I believe that hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation"

- **It is the nature of s/w – inherent in the process**
- **Conceptual errors are the problem**

Four Inherent Difficulties

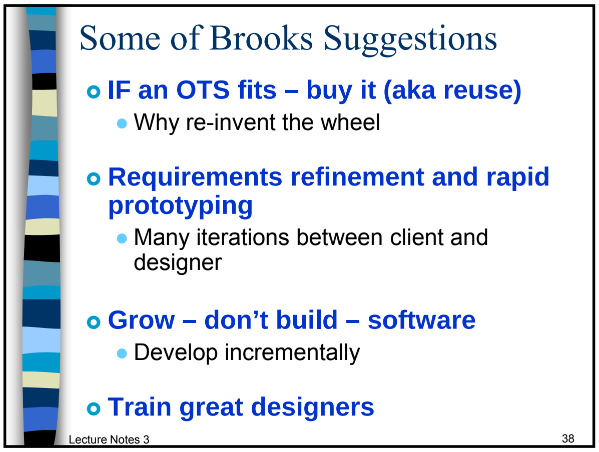
- **Complexity**
- **Conformity**
- **Changeability**
- **Invisibility**



Complexity

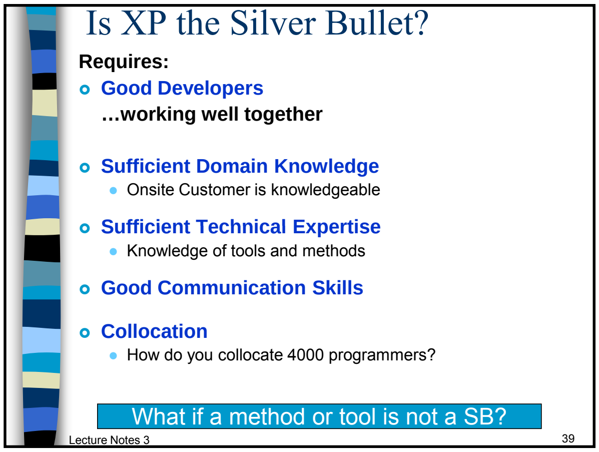
- Very large # of states
- Scaling up is not a repetition of the same elements in large sizes
- Elements interact in a non-linear fashion
  Complexity → Communication
- It is difficult to extend large programs without creating side effects

Complexity makes management difficult
Personnel turnover can be a disaster



Some of Brooks Suggestions

- IF an OTS fits – buy it (aka reuse)
  - Why re-invent the wheel
- Requirements refinement and rapid prototyping
  - Many iterations between client and designer
- Grow – don't build – software
  - Develop incrementally
- Train great designers

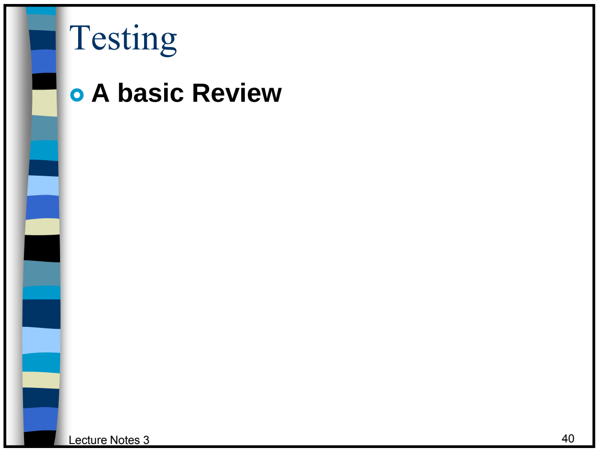


Is XP the Silver Bullet?

Requires:

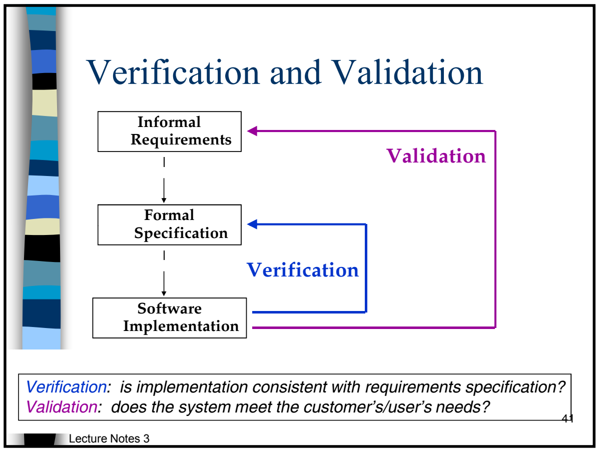
- Good Developers
  ...working well together
- Sufficient Domain Knowledge
  - Onsite Customer is knowledgeable
- Sufficient Technical Expertise
  - Knowledge of tools and methods
- Good Communication Skills
- Collocation
  - How do you collocate 4000 programmers?

What if a method or tool is not a SB?



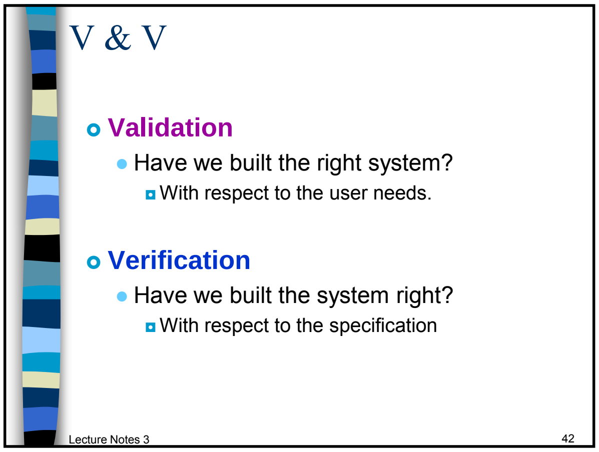
Testing

- A basic Review



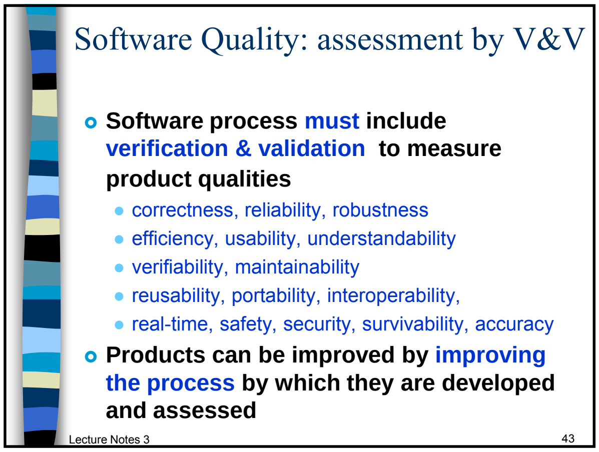
Verification and Validation

Verification: is implementation consistent with requirements specification?
Validation: does the system meet the customer's/user's needs?



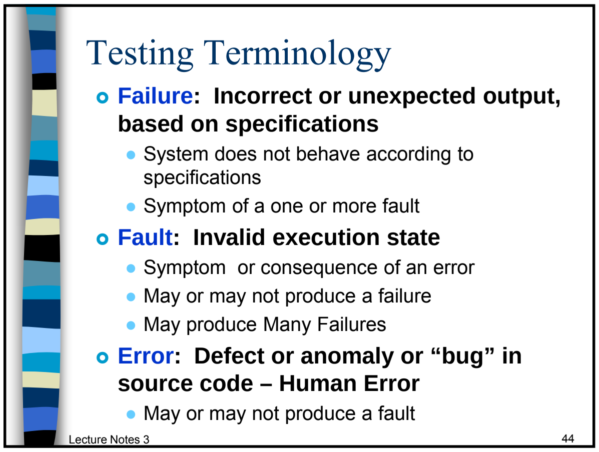
V & V

- Validation
  - Have we built the right system?
    - With respect to the user needs.
- Verification
  - Have we built the system right?
    - With respect to the specification



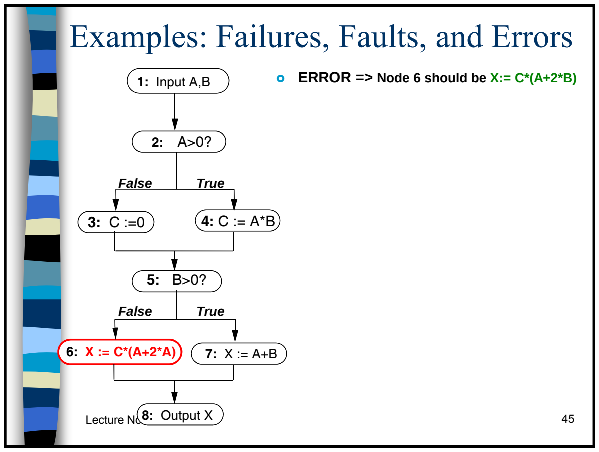
Software Quality: assessment by V&V

- **Software process must include verification & validation to measure product qualities**
  - correctness, reliability, robustness
  - efficiency, usability, understandability
  - verifiability, maintainability
  - reusability, portability, interoperability,
  - real-time, safety, security, survivability, accuracy
- **Products can be improved by improving the process by which they are developed and assessed**



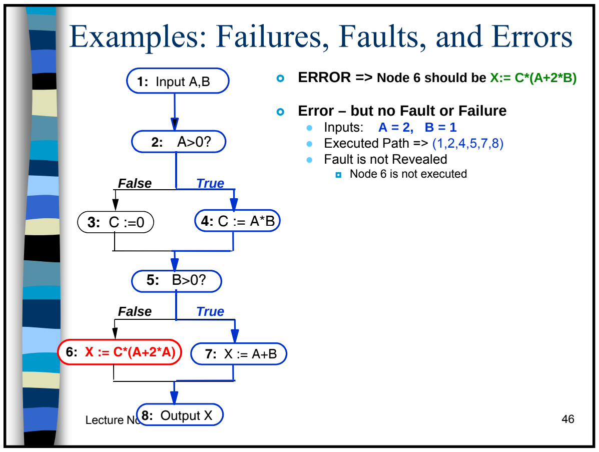
Testing Terminology

- **Failure: Incorrect or unexpected output, based on specifications**
  - System does not behave according to specifications
  - Symptom of a one or more fault
- **Fault: Invalid execution state**
  - Symptom or consequence of an error
  - May or may not produce a failure
  - May produce Many Failures
- **Error: Defect or anomaly or “bug” in source code – Human Error**
  - May or may not produce a fault



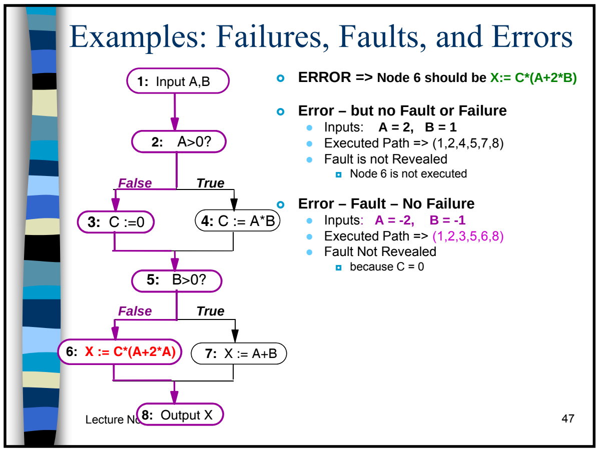
Examples: Failures, Faults, and Errors

- **ERROR => Node 6 should be X:= C*(A+2*B)**



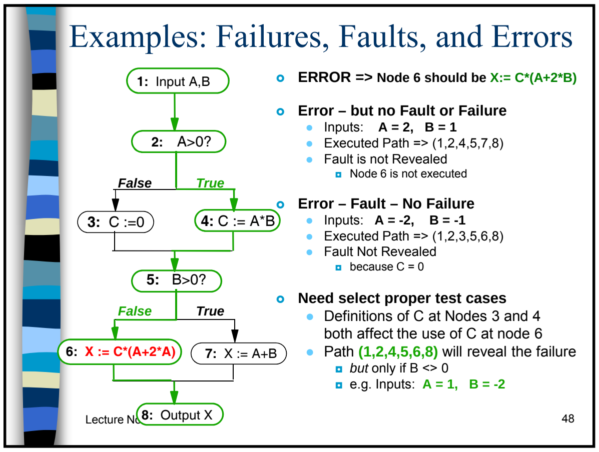
Examples: Failures, Faults, and Errors

- **ERROR => Node 6 should be X:= C*(A+2*B)**
- **Error – but no Fault or Failure**
  - Inputs: **A = 2, B = 1**
  - Executed Path => (1,2,4,5,7,8)
  - Fault is not Revealed
    - Node 6 is not executed



Examples: Failures, Faults, and Errors

- **ERROR => Node 6 should be X:= C*(A+2*B)**
- **Error – but no Fault or Failure**
  - Inputs: **A = 2, B = 1**
  - Executed Path => (1,2,4,5,7,8)
  - Fault is not Revealed
    - Node 6 is not executed
- **Error – Fault – No Failure**
  - Inputs: **A = -2, B = -1**
  - Executed Path => (1,2,3,5,6,8)
  - Fault Not Revealed
    - because C = 0



Examples: Failures, Faults, and Errors

- **ERROR => Node 6 should be X:= C*(A+2*B)**
- **Error – but no Fault or Failure**
  - Inputs: **A = 2, B = 1**
  - Executed Path => (1,2,4,5,7,8)
  - Fault is not Revealed
    - Node 6 is not executed
- **Error – Fault – No Failure**
  - Inputs: **A = -2, B = -1**
  - Executed Path => (1,2,3,5,6,8)
  - Fault Not Revealed
    - because C = 0
- **Need select proper test cases**
  - Definitions of C at Nodes 3 and 4 both affect the use of C at node 6
  - Path **(1,2,4,5,6,8)** will reveal the failure
    - *but* only if B <> 0
    - e.g. Inputs: **A = 1, B = -2**

Why do we care about Errors / Faults that never show up?

- **Latent faults**
  - Can be subsumed by previous statements
  - Maybe that state is never entered
- **Software is often reused later**
- **Conditions not hit in prev. version may be accessed later**
  - Code Changes

For Example: Ariane 5

- **Capable of hurling 2 – 3 ton satellites into orbit**
- **10 years**
- **$7 Billion**
- **Would have given Europe supremacy in the commercial satellite business**

Some Slides Adapted from Sommerville

Arian 5 (2)

- **Successor to the successful Ariane 4 launchers**
- **Ariane 5 can carry a heavier payload**

Whoops!

- **40 seconds into maiden flight**
  - veers off course & self-destructed
- **39 seconds after lift off**
  - Altitude reaches 2.5 miles
  - Ariane 5 goes into self destruct
  - Carrying 5 expensive - uninsured satellites

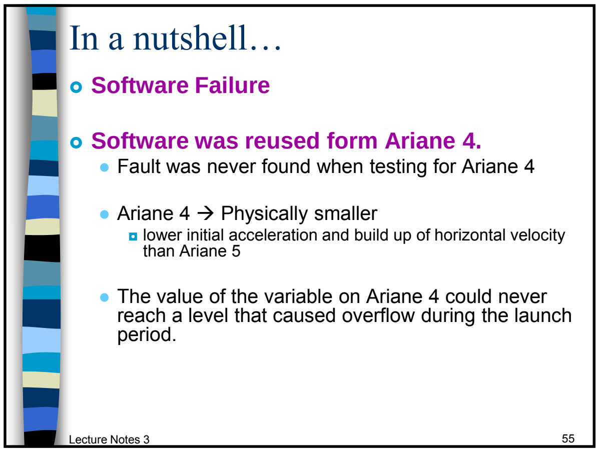
Why?

- **Why did it go into self destruct mode?**
  - Incorrect control signals were sent to the engines and these swivelled - Ariane 5 swerved
  - Pressure in boosters and main engine
- **Why did it swerve?**
  - It was making a course correction that was not needed.

Launcher Failure

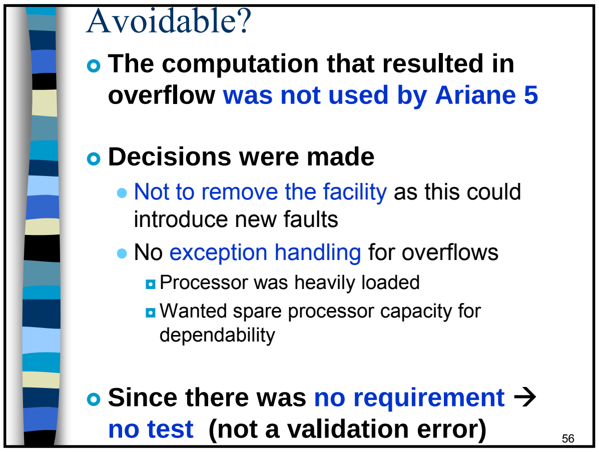
- **Why the course correction?**
  - Steering controlled by onboard computer
  - Thought course change was necessary because of numbers being displayed by the inertial guidance system
  - The numbers looked like data – impossible data- but was actually an error message

  → The guidance system had shutdown
- **Why did the guidance system shutdown?**
  - Tried to convert a 64-bit format velocity to a 16-bit format
  - Overflow error
- **What about the backup?**
  - Backup system failed too..
    - It was running the same software



In a nutshell…

- **Software Failure**
- **Software was reused form Ariane 4.**
  - Fault was never found when testing for Ariane 4
  - Ariane 4 → Physically smaller
    - lower initial acceleration and build up of horizontal velocity than Ariane 5
  - The value of the variable on Ariane 4 could never reach a level that caused overflow during the launch period.

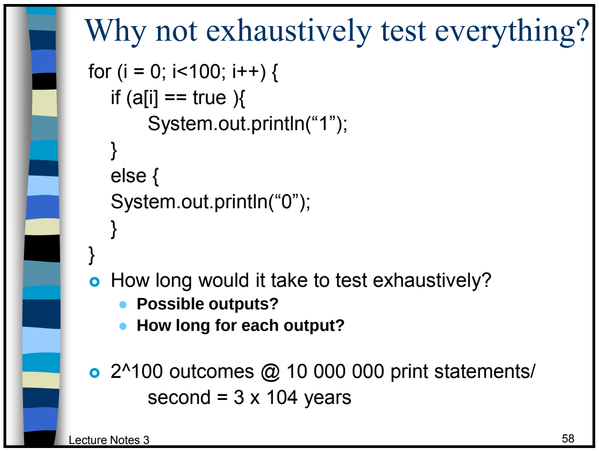


Avoidable?

- **The computation that resulted in overflow was not used by Ariane 5**
- **Decisions were made**
  - Not to remove the facility as this could introduce new faults
  - No exception handling for overflows
    - Processor was heavily loaded
    - Wanted spare processor capacity for dependability
- **Since there was no requirement → no test (not a validation error)**

Happy Ending…

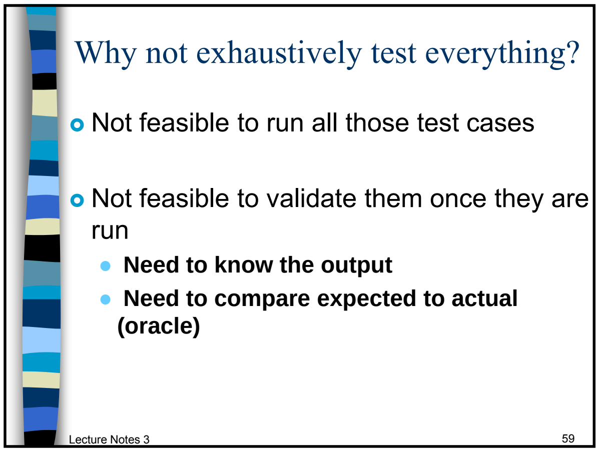
- **They fixed the error and…**



Why not exhaustively test everything?

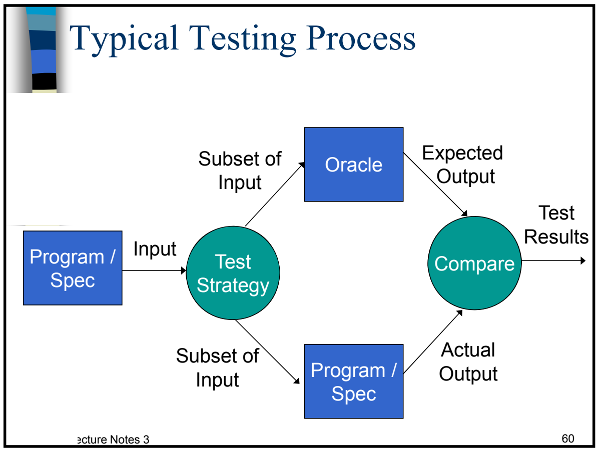
```
for (i = 0; i<100; i++) {
   if (a[i] == true ){
        System.out.println("1");
   }
   else {
   System.out.println("0");
   }
}
```

- How long would it take to test exhaustively?
  - **Possible outputs?**
  - **How long for each output?**
- 2^100 outcomes @ 10 000 000 print statements/ second = 3 x 104 years



Why not exhaustively test everything?

- Not feasible to run all those test cases
- Not feasible to validate them once they are run
  - **Need to know the output**
  - **Need to compare expected to actual (oracle)**



Typical Testing Process