Software Engineering
G22.2440-001

Session 11 - Main Theme
Building Software

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Agenda

- Review
- Software Engineering Teams
- Pair Programming
- Pair Programming Effects
- Refactoring
- Test Driven Development (TDD)
- Distributed Development and Agile Methods Scalability
- Languages, Platforms, and Component Infrastructures
- Summary
  - Readings
  - Individual Assignment #5 (ongoing)
  - Project Part 3 (ongoing)
Summary of Previous Session

- Review
  - Business Process Modeling
  - Capturing the Organization and Location Aspects
  - Developing a Process Model
  - Business Process Management
  - Model Driven Architectures
  - Workflow Support Technologies
  - Business Model Engineering Frameworks
- BME Advanced Topics
  - Business Process Interoperability
  - EDOC and ebXML
  - Component Collaboration Architecture
  - ECA Entity Profile
  - ECA Business Events
  - EDOC Business Processes
  - Patterns
- Summary
  - Individual Assignment #5
  - Project Part 3

Part I

Review
XP vs. Learning to Drive

• A story of learning to drive a car
  – … first told the young driver “line the car up in the middle of the lane, straight toward the horizon”
  – … The teenager drives car off the road!
  – … “Driving is not about getting the car going in the right direction. Driving is about constantly paying attention, making a little correction this way, a little correction that way.”

• This is the paradigm for XP: Change is constant, must be constantly monitored and adapted to

Four “Control” Variables

• External forces get to pick the values of any three variables
• The development team picks the value of the fourth

<table>
<thead>
<tr>
<th>Scope</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Four Variables
Pick three, any three . . .

• **Scope** is how much is to be done
  – i.e., the features to be implemented
• **Resources** are how many people are available.
  – i.e., personnel, equipment, etc.
• **Time** is when the project or release will be done
  – i.e., the duration of the project
• **Quality** is how good the software will be and how well tested it will be
  – i.e., the requirements for “correctness”

Visibility

• However, the values of all four variables need to be “visible”
  – If stakeholders (e.g., customers, management) can see all four variables they can consciously choose which variables to control
  – If they do not like the resulting value of the fourth variable, they can choose to change the inputs or choose to control a different set of three
“Scope” is Critical

- XP argues that “scope” is the most important of the four control variables
  - By adjusting project scope based on the values of the other three, you increase your chance of success
- This perspective is backed by XP practices
  - Practice making estimates
  - Implement most important requirements first

Cost Curve

- General SE Wisdom: Cost of Change increases exponentially over time
- It's cheaper to fix a bug if it's caught earlier in the life cycle
  - Requirements: $1
  - Analysis: $10
  - Design: $100
  - Implementation: $1,000
  - Testing: $10,000
  - Production: $100,000
Flattening the Cost Curve

• XP is predicated on the notion that given the right set of practices, the cost curve can be flattened
  – Same change costs same amount throughout lifecycle
• This is a BIG assumption
  – Not widely accepted
  – May make adoption of XP impossible for some organizations

How to Flatten the Curve?

• Technology
  – Objects
    • Used correctly they provide extreme flexibility
    • XP instigators of XP were Smalltalk programmers
  – Components, component model frameworks
• Practices
  – Simple Design, Automated Tests, Refactoring, ...
• Power sharing among stakeholders
• ...
• If you want
  – more software
  – higher quality
  – in less time
  – for less money
  – with more certainty

• Then just do it with
  – fewer people
  – less management
  – less complexity
  – fewer tools
  – developer commitment

Could it be this easy?

Power Sharing

Customer

Developer
## Needs to know

<table>
<thead>
<tr>
<th>Customer</th>
<th>Developer</th>
</tr>
</thead>
<tbody>
<tr>
<td>• How long</td>
<td>• What to do</td>
</tr>
<tr>
<td>• What’s done</td>
<td>• When to do it</td>
</tr>
<tr>
<td>• How good</td>
<td>• When done</td>
</tr>
</tbody>
</table>

### XP Programmer Rights

- You have the right to know what is needed, with clear declarations of priority.
- You have the right to produce quality work at all times.
- You have the right to ask for and receive help from peers, superiors, and customers.
- You have the right to make and update your own estimates.
- You have the right to accept your responsibilities instead of having them assigned to you.
XP Manager and Customer Rights

• You have the right to an overall plan, to know what can be accomplished, when, and at what cost.
• You have the right to get the most possible value out of each programming week.
• You have the right to see progress in a running system, proven to work by passing repeatable tests that you specify.

XP Manager and Customer Rights (continued)

• You have the right to change your mind, to substitute functionality and to change priorities, without paying exorbitant costs.
• You have the right to be informed of schedule changes, in time to choose how to reduce scope to restore the original date.
• You can even cancel at any time and be left with a useful working system reflecting investment to date.
XP Rules of Thumb

• Focus on Simplicity (Do the simplest thing that could possibly work)
• Build for NOW (You’re not gonna need it)
• Build Clean (Continuous Refactoring)
• Customers specify, Developers estimate (Iteration Planning)
• Steering to Success (Release Planning)

But what about ...

• **Unmaintainable code:** Simple, clear code, relentlessly tested, mercilessly refactored
• **Might never get done:** Clear schedule, small frequent releases with business value
• **Might not work:** Highest priority first, write unit tests first, continuous integration
• **People might not know what to do:** User Stories, Engineering Tasks, Pair Programming
XP Approach to Planning

- Write user stories.
- Release planning creates the schedule.
- Make frequent small releases.
- Measure Project Velocity.
- Divide project into iterations.
- Iteration planning starts each iteration.

The Planning Spectrum

![Planning Spectrum Diagram]

*Figure 1. The planning spectrum. Unplanned and undisciplined hacking occupies the extreme left, while micromanaged milestone planning, also known as inch-pebble planning, occupies the extreme right.*

*Source: Barry Boehm “Get Ready For Agile Methods, With Care”, IEEE Computer, Jan 2002.*
Basic Risk Exposure Profile

![Image of Basic Risk Exposure Profile]

Figure 2. Risk exposure (RE) profile. This planning detail for a sample e-services company shows the probability of loss (PL) and size of loss (SL) for several significant threats.

Source: Barry Boehm “Get Ready For Agile Methods, With Care”, IEEE Computer, Jan 2002.

Agile Home-ground Profile

![Image of Agile Home-ground Profile]

Figure 3. Comparative RE profile for an agile home-ground company with a small installed base and less need for high assurance.

Source: Barry Boehm “Get Ready For Agile Methods, With Care”, IEEE Computer, Jan 2002.
Plan-driven Home-ground Profile

![Graph showing comparative RE profile for a plan-driven home-ground company that produces large, safety-critical systems.]

Source: Barry Boehm “Get Ready For Agile Methods, With Care”, IEEE Computer, Jan 2002.

XP User Stories

- Analogous to use cases or usage scenarios, replaces requirements document
- Written by the customers as things that the system needs to do for them
- About 3 sentences in the customer’s terminology
- Estimate development time in “ideal development time” (usually 1-3 weeks)
- Prioritize
XP Release Planning

• Release plan lays out overall project
• Select user stories, considering ideal development time and priority, for first/next release
• Plan by time or scope
• Individual iterations are planned in detail just before each iteration begins, not in advance

Release Planning

• Requirements via User Stories
  • Short (index-card length) natural language description of what a customer wants
  • Prioritized by customer
  • Resource and risk estimated by developers
• Via “The Planning Game”
  • Highest priority, highest risk user stories included in early “time boxed” increments
• Play the Planning Game after each increment
Small Incremental Releases

- Deliver real business value
- On a very short cycle
  - Business value sooner
  - Rapid feedback
  - Sense of accomplishment
  - Reduced risk
  - Customer confidence
  - Adjustments to requirements

Plan by time or scope

- Project velocity is used to determine either how many stories can be implemented before a given date (time) or how long a set of stories will take to finish (scope)
- Planning by time: multiply number of iterations by project velocity to determine how many user stories can be completed
- Planning by scope: divide total weeks of estimated user stories by project velocity to determine how many iterations until the release is ready
**Project Velocity**

- Add up the estimates of the user stories that were finished during the iteration
- Add up the estimates for the tasks finished during the iteration
- During the next iteration planning meeting, customers are allowed to choose the same number of user stories finished in the previous iteration
- Those stories are broken down into technical tasks and the team is allowed to sign up for the same number of tasks completed in the previous iteration
- The problem with any project is the initial estimate

**Iterative Development**

- Divide development schedule into about a dozen iterations of 1 to 3 weeks in length
- Keep iteration length constant throughout project
- This constant makes measuring progress and planning simple and reliable
- Just-in-time planning at beginning of each iteration cycle
Iteration Planning

- Iteration planning meeting at beginning of each iteration to plan out what will be done
- User stories are chosen for this iteration by the customer from the release plan in order of the most valuable to the customer first, considering project velocity
- Failed acceptance tests to be fixed are also selected
- The user stories and failed tests are broken down into engineering tasks

Tasks

- Written in developer’s language, not customer’s
- Developers sign up to do the tasks and then estimate how long their own tasks will take to complete
- Important for the developer who accepts a task to also be the one who estimates how long it will take to finish
- Estimate 1-3 ideal development days
- Use project velocity to determine if iteration is overbooked, if necessary move user stories to future iterations
XP Approach to Designing

- Simplicity.
- Choose a system metaphor.
- Use CRC cards for design sessions.
- Create spike solutions to reduce risk.
- No functionality is added early.
- Refactoring.

Simple Design

- Always do the simplest thing that could possibly work
- The right design is one that
  - Runs all the tests
  - Has no duplicated logic
  - States every intention important to the programmers
  - Has the fewest possible classes and methods
Simple Design

- No Big Design Up Front (BDUF)
- “Do The Simplest Thing That Could Possibly Work”
  - Including documentation
- “You Aren’t Gonna Need It” (YAGNI)
- CRC cards (optional)

On-Site Customer

- Customer available on site to clarify stories and to make critical business decisions.
- Developers don’t make assumptions
- Developers don’t have to wait for decisions
- Face to face communication minimizes the chances of misunderstanding
XP Metaphor

- The closest XP comes to architecture
- Gives the team a consistent picture of describing the system, where new parts fit, etc.
- Words used to identify technical entities should be chosen from the metaphor
- C3 payroll . . . The paycheck goes down the assembly line and pieces of information are added.
- Sometimes, you just can’t come up with one

Metaphor

- Name classes, methods, etc. consistently
- Examples:
  - The Internet is like air-traffic control, with all-powerful master controllers observing everything, telling each packet where to go.
  - The Internet is like spiders crawling on a web - when one way is blocked, they just go another way, always trying to get to their goal.
Class, Responsibilities, and Collaboration (CRC)

• Each card represents an object
  – Put class of the object at the top of the card
  – Responsibilities listed down the left side
  – Collaborating classes listed to the right of each responsibility
• Simulate the system by talking about which objects send messages to other objects
• Move the cards around

Spike Solutions

• A very simple program to explore potential solutions to tough technical or design problems
• Only addresses the problem under examination and ignore all other concerns
• Expect to throw it away
• Goal is reduce the risk of a technical problem or increase the reliability of a user story's estimate
Refactor Mercilessly

- Before adding - is there a way to change the program to make the addition simple(r)?
- After adding - is there a way to make the program simpler while still running all the tests?
- Remove redundancy, eliminate unused functionality
- Keep your code clean and concise so it is easier to understand, modify, and extend

Coding and Testing

- The customer is always available.
- Code must be written to agreed standards.
- Code the unit test first.
- All code must pass all unit tests before it can be released/integrated.
- All production code is pair programmed.
- Only one pair integrates code at a time.
- Integrate often.
- Run acceptance tests often.
- Leave optimization until the end.
On-Site Customer

- A real customer must sit with the team, available to answer questions, resolve disputes, and set small-scale priorities
- Can’t afford a customer? (An expert, not a trainee)
- If the system isn’t worth the time of one customer, maybe it’s not worth building

Coding Standards

- Collective code ownership, partnering, refactoring all require common coding standards
- Make it impossible to tell who wrote what
- Standard requires least work possible
- Emphasize communication
- Adopted voluntarily by entire team
Code Unit Tests First

- Helps developer to really consider what needs to be done
- Requirements nailed down firmly by tests
- Immediate feedback while you work
- The harder the test is to write the more you need it

Unit Testing Procedure

- Create one test to define some small aspect of the problem at hand
- Then create the simplest code that will make that test pass
- Then create a second test
- Now add to the code to make this new test pass, but no more!
- Continue until there is nothing left to test
Automated Unit Testing

- Use a unit test framework to create automated unit tests suites
- Release unit tests into the code repository along with the code they test
- Guard against bugs introduced by collective code ownership, refactoring, integration

Sequential Integration

- Parallel integration of source code modules combines code that has not been tested together before
- No clear-cut latest version of either source code or unit test suite
- Take turns: only one development pair integrates, tests and releases changes to the source code repository at any given moment
- Dedicated release computer
Integrate Often

- Integrate and test after a few hours - a day at most.
- If a test fails - you broke it!
- Continuous integration avoids or detects compatibility problems early

Acceptance Tests

- Created from user stories
- User stories selected during an iteration planning meeting must be translated into acceptance tests
- Black box system test represents some expected result from the system
- Customers responsible for verifying the correctness of the acceptance tests and deciding which failed tests are of highest priority
- Used as regression tests prior to a production release
Optimize Last

- Never try to guess what the system's bottle neck will be: Measure it!
- Make it work, make it right, then make it fast

Back to: Why “Extreme”?  

- Extreme Programming takes commonsense software development principles and practices to extreme levels
- Commonsense in this case diverges substantially from the conventional Waterfall process model
- And sounds like Code-and-Fix
- But is essentially a recent special case or refinement of Incremental/Iterative processes
When to Use XP?

• Problem domains whose requirements change
  – Customers may not have a firm idea of what the system should do
  – Functionality is expected to change every few months

When to Use XP?

• High project risk
  – High: If customers need a new system by a specific date
  – Higher: If system is a new challenge for your software group
  – Highest: If system is a new challenge to the entire software industry
When to Use XP?

• Small groups of programmers: 2-12
• Ordinary programmers: don’t need a PhD
• Extended development team: managers and customers as well as developers
• Testability: possible to automate unit and acceptance tests

Why is XP Successful?

• Stresses customer satisfaction
  – Methodology designed to deliver the software your customer needs when it is needed
  – Empowers developers to confidently respond to changing customer requirements, even late in the life cycle
• Emphasizes team work
  – Managers, customers, and developers all part of a team dedicated to delivering quality software
  – Implements simple, yet effective way to enable groupware style development
Summary

• There are 12 practices of XP; some of these are very inter-related and dependent on each other.
• The focus is on getting value to the customer as quickly as possible and on embracing to change throughout the development cycle.

The 12 Practices of XP

1. Metaphor
2. Release Planning
3. Testing
4. Pair Programming
5. Refactoring
6. Simple Design
7. Collective Code Ownership
8. Continuous Integration
9. On-site Customer
10. Small Releases
11. 40-Hour Work Week
12. Coding Standards
Overall XP Process
(review)

Agile Model-Driven Development Lifecycle
(review)
XP Iteration Planning

Part II

Software Engineering Teams
Section Objectives

• Point out Common Problems with Team Structures in Software Engineering
• Describe a variety of team structures:
  • Democratic
  • Chief Programmer
  • Modified Chief Programmer
  • Synchronize and Stabilize
  • Extreme Programming
• Present a categorization of teams

Teamwork Problems

• Most software is too large or complex to be developed by an individual → a team is born!
• The Nature of Tasks:
  – Shared e.g. strawberry picking
    • Can be done in parallel. Four people on the job gets it done in 1/4 the time
  – Individual, e.g. pregnancy
    • Can only be done in serial. Nine women cannot produce a baby in one month
  – Combination, e.g. software implementation
    • Coding can be done separately but requires communication for integration
Communication Explosion

• Assume everyone in a team of $n$ people communicates with everyone else
• Number of communication paths $= 0.5 \times n \times (n-1)$

Democratic Teams

• Requires Egoless Programming:
  – Coders not too attached to their code
  – Needs an atmosphere of cooperation
  – Bug finding is given a positive spin
• 10 Egoless Programmers = A Democratic Teams
• Everyone is equal and the team is self-organizing
• There is no appointed leader $\Rightarrow$ Goes against conventional management wisdom
• Group theory suggests it will work well for complex problems
Chief Programmer Teams

• Modeled on Surgical Teams: A hierarchy is used to overcome communication explosion, specialization to improve productivity
• A chief programmer teams of 6 people reduces communication links from 15 to 5
• Roles:
  – Chief Programmer: designs the architecture; codes complex classes; overall manager
  – Back-Up Programmer: takes over if needed; designs tests
  – Librarian: maintains code base and documentation; runs tests

Chief Programmer Track Record

• Results from Initial Test Project were extraordinary:
  – New York Times automated clipping file
  – 83,000 LOC in 22 months (=11 person-years)
  – Half the modules (200-400 LOC each) were correct on first compilation
  – File Maintenance component operated 20 months without a single fault
• But no comparable wild success stories since
• Because the Chief Programmer (Terry Baker) was one of a breed of super-programmers
Problems with the Chief Programmer

- Good Chief Programmers are rare:
  - Intersection set of highly skilled programmers and successful managers
- Backup Programmers are even more rare:
  - As capable as the Chief Programmer but prepared to accept a lower salary and subordinate position
- Librarians are also difficult to find:
  - Who wants to do nothing but paperwork all day
- Also Doesn’t Scale Well

Modified Chief Programmer Teams

- Split Chief Programmer into:
  - Team Leader (technical issues)
  - Team Manager (non-technical management issues)
- Carefully separate responsibilities to avoid conflicts of interest
- But there may be some areas of overlap (e.g. annual leave)
- Can be scaled with layers of leaders
Synchronize and Stabilize Teams

- Microsoft model; successful for very large projects
  - More than 3000 .dev and .test worked on Windows 2000
- Small parallel teams of 3-8 developers and 3-8 testers
- Individuals allowed latitude to design and implement a spec but:
  - Code must be integrated on a daily basis
  - If your code prevents compilation then it must be fixed immediately
- But Microsoft is more than Synchronize and Stabilize. Their success is based on a strong corporate culture

Extreme Programming Teams

- Based on Pair Programming:
  - Spreads knowledge
  - Brings less experienced programmers up to speed
- Evidence in Favor:
  - From Williams et al. “Strengthening the Case for Pair Programming” IEEE Software
  - Subject were students at the University of Utah doing an Advanced Programming course
  - By the end pairs took 60% of the time to do the same programming task as individuals BUT
  - Passed 94% of test cases rather than 78%
  - Pair programming improved job satisfaction and overall confidence
Exercise: Choosing a Team Structure

- Problem: As a project manager you must choose a team structure for the following projects:
  1. A quantum computing project with 5 researchers. There is no strict deadline. If successful this team will remain intact for later projects
  2. A payroll system for a mining company. This has a work estimate of 30 person-years and must be delivered according to a very strict 10 month deadline

- Solution:
  1. Small size, high difficulty, long team lifetime = Democratic or Extreme Programming
  2. Medium size, low difficulty, strict delivery date = Modified Chief Programmer

A Categorization of Team Structures

- Democratic Decentralized (DD):
  - No permanent leader; decisions are made by group consensus
  - Communication and control are horizontal
- Controlled Decentralized (CD):
  - A leader coordinates tasks; Problem solving remains a group activity
  - Communication is horizontal and control is vertical
- Controlled Centralized (CC):
  - A leader coordinates tasks and solves problems
  - Communication and control are vertical
Exercise: Classifying Teams

- Problem:
  - Categorize the Democratic, Classical and Modified Chief Programmer, Synchronize and Stabilize, and XP team structures as Democratic Decentralized (DD), Controlled Decentralized (CD) or Controlled Centralized (CC)

- Solution:
  - Democratic = DD
  - Classical and Modified Chief Programmer = CC
  - Synchronize and Stabilize = CD
  - XP = DD

Jelling

- If business groups are assigned to work together; often without team spirit
- An effective tightly knit group displays Jell or “Esprit de Corps”. “Once a team begins to jell, the probability of success goes way up. The team can become unstoppable, a juggernaut for success”.
- Team Toxicity (factors that work against jelling):
  1. A frenzied work atmosphere (which wastes energy and lacks focus)
  2. High frustration caused by personal, business, or technological factors that cause friction among team members
  3. Fragmented or poorly coordinated procedures
  4. Unclear definition of roles resulting in a lack of accountability
  5. Morale damaged by continuous and repeated failure
# A Comparison of Teams

<table>
<thead>
<tr>
<th>Organization</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Democratic</td>
<td>Collective code ownership, Handles hard problems</td>
<td>Cannot be externally imposed, May not scale</td>
</tr>
<tr>
<td>Classical Chief Programmer</td>
<td>Miraculous success with New York Times</td>
<td>Impractical</td>
</tr>
<tr>
<td>Modified Chief Programmer</td>
<td>Scales well, Handles strict delivery</td>
<td>No successes comparable to New York Times</td>
</tr>
<tr>
<td>Synchronize and Stabilize</td>
<td>Encourages creativity, Scales very well</td>
<td>Is it effective outside Microsoft?</td>
</tr>
<tr>
<td>Extreme Programming</td>
<td>Many - sharing info, group ownership of code, improved quality</td>
<td>Not fully proven</td>
</tr>
</tbody>
</table>

## Part III

*Pair Programming*
Overview

- *Two* programmers work side-by-side at *one* computer
- Continuously collaborate on same design, algorithm, code, test, etc.
- Continuous informal *review*
- Demonstrated to improve productivity and quality of software products
  - Two programmers work together more than twice as fast and think of more that twice as many solutions to a problem as two working alone
  - While attaining higher defect prevention and defect removal
Pair Programming

Pair-programming has been popularized by the eXtreme Programming (XP) methodology

With pair-programming:

• Two software engineers work on one task at one computer
• One engineer, the driver, has control of the keyboard and mouse and creates the implementation
• The other engineer, the navigator, watches the driver’s implementation to identify defects and participates in on-demand brainstorming
• The roles of driver and observer are periodically rotated between the two software engineers

Exercise....

• Take a sheet of paper (or an overhead and overhead pens) and draw a “scary” face.....
• Spend about 2 mins drawing two scary faces.....
Exercise as a pair.....

- Pair up with a partner
- Take a sheet of paper and a different color each.....
- Spend about 2 mins drawing two faces.....

Debriefing......

- How did you feel when you were drawing solo vs. drawing as a pair
- Which of the drawings are more artistic or original?
- Did you find yourself concentrating more or less during the pair drawing?
- Was it more fun to draw alone or as a part of a pair
- What did you like, what didn’t you like about drawing alone or as part of a pair
- Did you find yourself mirroring your partner?
- Would you expect that people get better over time drawing as part of a pair?
Pair Programming at Work

Solo Drawing

Pair Drawing

What is pair programming?

TWO programmers working side-by-side, collaborating on the same design, algorithm, code or test. One programmer, the driver, has control of the keyboard/mouse and actively implements the program. The other programmer, the observer, continuously observes the work of the driver to identify tactical (syntactic, spelling, etc.) defects and also thinks strategically about the direction of the work. On demand, the two programmers can brainstorm any challenging problem. Because the two programmers periodically switch roles, they work together as equals to develop software.

-- Laurie Williams
North Carolina State University Computer Science
**Pair Programming and Knowledge Construction**

*KNOWLEDGE IS commonly socially constructed, through collaborative efforts towards shared objectives or by dialogues and challenges brought about by different persons' perspectives.*

**G. Salomon** (book: *Distributed Cognitions: Psychological and Educational Considerations*)

---

**What is pair programming**

*(continued)*

- Think of a good pair driving across the country. One will drive, the other navigate (thinking tactically and strategically)
- Often used as a part of extreme programming
XP “Human” Values

- Communication
- Courage
- Feedback
- Simplicity

eXtreme Programming Project Lifecycle
Pair Programming

• CRC Cards
  – Class-Responsibility-Collaboration

Pair Programming

• Basic rules:
  – Two people, one computer
  – One
    • Types
    • Thinks tactically
    • Interacts
  – Other
    • Observes
    • Thinks strategically
    • Checks and corrects (verbally)
    • Interacts
Pair Programming

• Proponents Claim
  – Increases software quality
  – No impact on delivery time
    • Some claim this is due to higher productivity while programming
    • Some claim this is due to higher quality earlier in the process
    • (Some claim this is bunk)
  – Happier programmers

XP Collective Code Ownership
Some quotes from pair programmers…

“When I explained an idea to my partner, I concentrated on what I was saying, and carefully made things clear and logical because I did not want to confuse my partner and I wanted him to understand what I was talking about. It helped me better understand the problem I was addressing. It also helped me discover some mistakes I had made but did not notice before I talked with my partner.”

Some quotes from pair programmers...

“One problem with single programming is that you can forget what you are doing and easily get wrapped in a few lines of code, losing the big picture. Your partner is able to constantly review what you do, making sure that it is in line with the product design. He/she can also make sure that you are not making the problem too difficult. Many times, these two items alone can waste a lot of time. When it comes down to it, wouldn’t you rather just get the job done correctly and quickly? Collaborative programming will help you do just that.”
PP Teaching Experiences

• Bimodal distribution of scores on the exams
• Some students seem very “cocky” – others are convinced they are “terrible at programming”
• Very little resources to help students
• Self-esteem quite low for many
• Isolation is prevalent among many of the students

PP Teaching Experiences

• Setting the stage is very important (e.g., exercises in class)
• Students for the most part love it and said they will do it again but a few hate it…
• Matching of skills is probably important…. Students seem to think so
• Biggest problem experienced by students is finding a common time to get together at school (some work, etc.)
Code Warriors and Code-a-Phobes

- **Code warriors** see themselves “as a sort of code-warrior, fighting with the enemy compiler, forcing it to assent to their glorious code and to produce a program that obeys their every desire”
- **Code-a-phobes** – seems to be an unfortunate phenomenon in computer science, report that they “hate programming” or that they are “hopeless at programming”
- Mixture of such students is part of the challenge of teaching first year programming

Study on Use of Pair Programming

- Studies indicate about 80-90% of students like pair programming and feel their solutions are more correct
- But other studies show differences between code warriors and code-a-phobes
  - Two variables – attitude and performance, may be independent
  - Better if they are matched in similar pairs w.r.t. attitude (didn’t look at performance in this study)
  - Code warriors are less likely to enjoy pair programming
Discussion points

• Can we create some kind of virtual environment to enable pair programming at distributed locations – or would that not remove the condition that makes it so special?
• How can we make first year programming more fun and interesting? Can technology help?
• How can we build self esteem?

Pair Programming Guidelines
All I Really Need to Know about Pair Programming I Learned in Kindergarten.

All I Really Need to Know I Learned in Kindergarten :
Uncommon Thoughts on Common Things
Robert Fulghum, 1988
Share everything

- Two programmers are assigned to jointly produce one artifact
- One person typing or writing, the other continuously reviewing
- Both equal participants
- Both partners own everything

Play fair

- One person “drives” while the other continuously reviews
- Important to take turns “driving”, even if one programmer much more experienced than other
- Observer is active and engaged, not passive
- Continuous analysis, design and code reviews
Don’t hit your partner

• But make sure your partner stays focused and on-task
• Less time wasted (reading email, surfing web, etc.) than when working alone, since partner awaiting continuous contribution and input
• Each partner expects the other to follow prescribed development practices

Put things back where they belong

• Put negative thoughts in the trash can
• Very difficult to work with someone with great insecurity or anxiety about their programming skills
• Should view pair programming as an opportunity to improve skills by watching and obtaining feedback
• Also difficult to work with big egos
• No one is infallible and above the input of another
Clean up your mess

• Many obvious but unnoticed defects become noticed by another person watching over your shoulder
• Defects removed without the animosity that might develop during a formal inspection meeting

Don’t take things too seriously

• Ego-less programming
• Don’t be defensive with receiving criticism
• But don’t always just agree with your partner
• Initial adjustment period to pair programming in general and partner in particular
Say you’re sorry when you hurt somebody while moving furniture

- Appropriate workspace layout critical to success
- Slide the keyboard, don’t move the chairs
- Effective communication, within pairs and among pairs, is paramount
- Programmers need to see each other, ask each other questions, and make decisions on things such as integration issues

Wash your hands of skepticism before you start

- Partner buy-in critical to success
- A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts
Flush

• When pair programmers work on something independently, need to flush and rewrite jointly
• Or at least review jointly – but more likely then to still have defects

Warm cookies and cold milk are good for you

• Pair programming can be very intense and mentally exhausting
• Periodically take a break
Live a balanced life – learn some and think some and draw and paint and sign and dance and play and work every day some

• Communicate with others on a regular basis
• Most programmers would probably say they preferred to work alone in a place where they wouldn’t be disturbed by other people
• But informal discussions allow for effective idea exchange and efficient transfer of information

Take a break from working together every afternoon

• Might do experimental prototyping, tough deep concentration problems, and logical thinking alone
• Simple, well-defined, rote coding more efficiently done alone and then reviewed with a partner
When you go out into the world, watch out for traffic, hold hands and stick together

- No competition between the paired programmers
- Both work for a singular purpose
- Do not place blame for defects or problems on either partner
- Pair should trust each other’s judgment and loyalty to the team

Be aware of the power of two brains

- When two people working together, each has own set of knowledge and skills
- Large common subset allows to interact effectively
- Pool resources to accomplish tasks
Pair Rotations

Sub-Section Objectives

• PP Knowledge management advantages
• PP Training advantages
• PP Communication advantages
• PP Team-building advantages
The Right Partner

Want to maintain specialists
Want to spread knowledge around a team
but …
Want PP benefits

Natural Pairs/Rotations

• Teams often develop rotations without a prescribed schedule
• Programmers learn to maximize their effectiveness by pairing with people who are appropriate for the subtasks
• New members paired with mentor-types for training
Example Development

• Drink vending machine program

<table>
<thead>
<tr>
<th>Team member</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chris</td>
<td>GUI</td>
</tr>
<tr>
<td>Brian</td>
<td>Customer capabilities</td>
</tr>
<tr>
<td>Kim</td>
<td>Machine maintenance</td>
</tr>
<tr>
<td>Lisa</td>
<td>Data structures</td>
</tr>
</tbody>
</table>

Initial engineer subtask assignments

Example Development

• Possible pairs to consider

<table>
<thead>
<tr>
<th>Task</th>
<th>Task owner</th>
<th>Partner</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI for “buy drink”</td>
<td>Chris</td>
<td>Brian</td>
</tr>
<tr>
<td>GUI for “add inventory”</td>
<td>Chris</td>
<td>Kim</td>
</tr>
<tr>
<td>GUI for “add recipe”</td>
<td>Chris</td>
<td>Kim</td>
</tr>
<tr>
<td>Input coins / return coins</td>
<td>Brian</td>
<td>Chris</td>
</tr>
<tr>
<td>Select drink</td>
<td>Brian</td>
<td>Lisa</td>
</tr>
<tr>
<td>Ingredient data structure</td>
<td>Lisa</td>
<td>Kim</td>
</tr>
<tr>
<td>Recipe data structure</td>
<td>Lisa</td>
<td>Kim</td>
</tr>
<tr>
<td>Add ingredients</td>
<td>Kim</td>
<td>Lisa</td>
</tr>
<tr>
<td>Customer analysis</td>
<td>Kim</td>
<td>Chris</td>
</tr>
</tbody>
</table>
Rationale

• Chris is GUI expert
  – He owns all GUI tasks
  – He partners with whomever the code touches
• Chris and Lisa never pair
• Kim and Brian never pair
  – Their code does not directly interact
  – They may choose to pair to broaden knowledge

Assignment Logistics

• Project responsibilities must be broken down into manageable tasks of < 1 week
• Chunk owners must recruit partners for working on the chunk

“I said it would take 3 months to complete the GUI… I didn’t know I’d have to help Kim and Brian do their code!! “
Assignment Logistics

• 2 main methods have been observed for successfully allocating pairs:
  – Short daily meeting
  – “Just say yes”

Short Daily Meetings

• Important practice of XP and Scrum
• Brief description of accomplishments of the day before… and problems
• “I’ve seen that before… I’ll pair with you today and work on it…”
• Manager makes pairs from remainders
Short Daily Meetings

- Other benefits
  - 15 minutes gives team synchronization
  - Efficient communications
  - Entire teams gets information
  - Minimum communication overhead
  - Prevents tunnel vision

“Just Say Yes”

- Task owner asks appropriate team member to pair… you cannot say no
- Only issue is when you 2 will pair… this is scheduling
- Whomever asks first, goes first
- Expect to give as you receive
Knowledge Management

• Knowledge compartmentalization on a project is a risk
  – Loss of team member is loss of knowledge
  – Vacations can be the same… temporary loss

• Pair rotation spreads knowledge
  – Thereby, reduces risk

Communities of Practice

• CoP are groups of professionals who form work-related common bonds
  – R. McDermott, “Knowing in Community”

• PP/PR causes CoP formation via
  – Jointly solved problems together
  – Common sense of purpose
  – Sharing project and job knowledge
  – Personal relationships
Communities of Practice

• CoP in an XP team means
  – Team members consider each other more approachable
  – Less time wasted struggling with problems alone before help is sought
  – Person-to-person sharing of knowledge, insight, experience

Communities of Practice

• Note this happens in research groups
  – Student/advisor mentor arrangement for knowledge sharing
  – Student bangs head on desk for 3 days, *comes to research meeting*, says “I am not sure where to go with this”
  – Advisor says “Karen has some experience in that area, go find her and get some pointers”
Communities of Practice

• Experience sharing can bring dramatic gains
  – NyNex (1996): CoP allowed reduction of # days needed to set up data services for new customers from 17 to 3

• This efficiency generation capability of CoP partially explains why PP pair can complete a program in a little more than half the time of a single programmer.

Training

• “Not only do you have experience walking out the door, you have INexperience walking in the door” (Scott Eliot)

• “Adding more people to a late project will make it later” (Fred Brooks)
Training

- Survey of 24 professionals

<table>
<thead>
<tr>
<th></th>
<th>With PP (pair rotation)</th>
<th>Without PP</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT: Assimilation time (work days)</td>
<td>13</td>
<td>28</td>
</tr>
<tr>
<td>MT: Mentoring time (%)</td>
<td>25</td>
<td>32</td>
</tr>
<tr>
<td>TE: Training effort (work days)</td>
<td>17</td>
<td>37</td>
</tr>
</tbody>
</table>

\[ TE = AT \times (1+MT) \]


Cockburn’s Approach

- “Day care” style
- Two slightly mixing teams
  - Progress team (85-95% of production)
  - Training team (5-15% of production)
- People transfer from one team to the other as they are ready
- Experts/mentors from progress team join training team as needed for situation (need special personality)
- Newbies can pair on training team as advanced training
Conclusions

- PP with no rotations does reap benefits
  - Quality gains
  - Effective communications
  - Team building
- Rotating pairs brings additional benefits

Part IV

Pair Programming Effects
**PP 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%)
- India Technology Company
  - 24% increase in productivity (KLOC/Person-Month)
  - 10-fold reduction in defects

**Extreme Programming Context**

- Deliberate and disciplined approach to software development
- Stresses customer satisfaction
- Rules and practices:
  - User stories, planning of small releases
  - Project divided into small iterations
  - Simple design
  - Unit tests coded first
  - Refactoring
  - Pair Programming
PP: How does this 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

PP: How does this work (part two)?

• 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
  – Tell it to the Furby

• 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
Pair Programming (PP)

- [Design,] Coding, Tests by a 2-person team
- “Driver“
  - Control of keyboard, mouse
  - Programming
- “Observer“
  - Observation of the code for all kinds of errors
  - Gives feedback to Driver
  - Strategically thinking ahead
- Frequently changing roles

Studies on PP

- Nosek:
  - Experiment with students
  - Comparison: PP – Solo Programming
  - PP needed 60% of time Solo P. needed
  - PP code less complex, more effective
- Williams:
  - Similar design, similar results
  - Subjective impressions of PP-users very positive
Case Study

• Check results of former studies in case study setting:
  – Larger teams (5-6 persons)
  – Larger project (~4000 LOC, ca. 700 person-hours)
• Evaluation of PP by
  – Quality of produced code
  – Effort
  – (New) subjective impressions from participants

Challenges (1/2)

• University course requirements
  – Students have to be taught the same topics
    • All students had access to the same material, but we tried to motivate some students to use PP
• Students can work at home
  – Effort and other measurement depends on reliability/discipline of students; quite low
    • We make turning in sheets part of grading
    • We tried to capture process conformance through questionnaire
Challenges (2/2)

• Students are graded
  – Fear that measurement may be used against them
  – No clear reports about problems and errors, or effort
    • Data collection done by a student
• GUI-intensive system
  – Use Scenarios and quiz-specifications as test cases for quality assessment of final system
  – However, testing not useful for quality assessment
    • Used different metrics (complexity)
• Very few data points for statistical analysis
  – Only trends recognizable, no significance

Case Study Design

• Observation of students during a practical course (one semester; 12 weeks)
• 6 teams, 5-6 persons per team
• 3 teams use PP,
  3 teams use unsystematic collaboration
• Task:
  – Modify and extend an existing system (“Web-based Quiz”)
    • Java/Swing
    • System reads XML-specifications of quizzes
• Measurement through time sheets, questionnaires, and interviews
Design (2/2)

• Iterations (three weeks each)
  – Iteration 1: Three subtasks (extend system functionality)
  – Iteration 2: Move to Client-Server architecture (RMI)
    • First Iteration well-suited for PP
  – Other tasks: Requirements, design

Hypotheses

• Quality of produced code is better
  1. Lines of Code: LOC (PP) ≤ LOC (other)
  2. Comment Ratio: CR (PP) ≥ CR (other)
  3. Coupling Factor: CF (PP) ≤ CF (other)
  4. Complexity: WMPC1 (PP) ≤ WMPC1 (other)
  5. Encapsulation: AHF (PP) ≥ AHF (other)
• Effort is 20% higher:
  6. Effort (other) ≤ Effort (PP) ≤ 1.2*Effort (other)
Results (1/4)

• Quality of code:
  – Lines of Code:
    • PP-teams needed less code for same functionality (hypothesis 1)

Results (2/4)

  – Comment Ratio (Hyp. 2):
    • Almost CR (PP) = CR (other)
      – No confirmation that comment ratio of PP programmers better
  – Coupling Factor (Hyp.3):
    • CF(PP)=47.8 ..., CF(other)=49.7 (It. 1)
      • As stated in hypothesis 3, the coupling factor of PP-teams was lower
  – Complexity (Hyp.4):
    • The worst case complexity was equal in all programs
      – No confirmation that complexity for PP-teams lower
  – Encapsulation (Hyp.5):
    • Confirmation that the code of PP-teams had a higher degree of encapsulation
Results (3/4)

- Effort (coding and testing only)
  - Hypothesis 6 confirmed
  - PP-teams needed only about 9% more effort than the others
  - Iteration 1: 15,12h PP, 13,81h others
  - Iteration 2: 26,44h PP, 24,26h others

Results (4/4)

- Subjective Impressions of students using PP
  - They think their code is easier to understand
  - They think their code annotations were very useful
  - They liked the working atmosphere and described it as very constructive
  - They think with PP errors are discovered earlier
  - Every member of the PP-teams would like to use PP in future projects, too
    - Very positive impressions of PP
Conclusion

• Results (trends):
  – Higher quality:
    • PP needs less code for the same functionality
  – PP needs slightly more effort (<120%)
  – Subjective rating:
    • Working atmosphere described as very constructive
    • All participants have a good impression of PP and would use it again
• Too few subjects for significant results
• Trends confirm the results of other experiments
• Case study context: larger project, more developers

Lessons Learned (1/2)

• Motivation worked well
  – The “motivated” groups used PP
  – However, training learning missing
    • PP performance may be suboptimal
  – Possibility: Ask groups to volunteer for PP; those groups receive training

• Need better Quality Metrics
  – Tool computed worst-case complexity
  – Dominated by existing system
  – Use better metrics or implement own computation?
Lessons Learned (2/2)

- Need better support for effort measurement
  - Manual collection on paper requires high discipline
    - Use automatic collection
      - Hackystat) or
    - Easy collection method
      - web-based or simpler
- Need better measurement of process conformance
  - Questionnaire we used was very “obvious”
  - Develop instrument?
  - Require certain presence times to observe teams?

Empirical Study for Validation

- Practice:
  - 20 Students (Sophomore/Junior)
    - All worked collaboratively
    - Generated more anecdotal/qualitative evidence
- Solo vs. Pair:
  - 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
Design of a controlled experiment 
on pair programming

(Erik Arisholm)
Topics

• Using a former experiment with 182 individuals as a baseline for realistic experiments on pair programming
• Students versus professionals in controlled experiments

Claims About Pair Programming

• In the literature, pair programming is claimed to have the following benefits:
  – Improves Quality – pairs produce code with fewer defects.
  – Reduces Time – pairs produce higher-quality code in about half the time of individuals.
  – Improves Morale – pair programmers are happier programmers.
  – Builds Trust and Teamwork – pair programming builds trust and improves teamwork
  – Facilitates Knowledge transfer – pair programmers, especially if they rotate partners, know more about the overall system.
  – Enhances learning – pairs continually learn by discussing solutions and watching each others techniques.
Independent Variables in a Pair-Programming Experiment:

- Characteristics of a “pair”
  - Personality, e.g., Introvert vs Extrovert.
  - Skill, e.g., Novice vs Expert (or use a calibration task!)
  - Collaboration Scheme: Driver vs Navigator
  - Pair programming experience
  - Pair chemistry: How long have they worked together as a pair? Do they know how to work together? Do they like working together?

- Characteristics of a “task”
  - Size, complexity, duration
  - Activities involved, e.g., analysis, design, coding, testing

  e.g., extrovert expert driver + introvert novice navigator on a complex, large task will probably not result in “happier programmers” 😊

The Main Experiment

- The main experiment took place during a two-month period and was organized in 12 separate one-day sessions.
  - 123 professional developers
    - Accenture, Ementor, Cap Gemini, Tietoenator, Software Innovation, Genera, Ementa, ObjectNet
  - 59 students
The Pair Programming Pilot Experiment:

- 8 professional developers with training in pair programming were formed into 4 (“expert navigator” + “novice driver”) pairs
- They followed the same experimental procedures as in the main experiment, but solved all programming tasks as a pair.
Web-based Experiment Support

Administrator
1: Define experiment
2: Add subjects

Simula Experiment Support Environment

Subjects: Professional developers, at usual workplace using normal development tools

Researcher
1: Define experiment

During 3 & 4: Monitor Experiment
5: Collect & analyze results

1: Define experiment

2: Add subjects

3: Questionnaires
   Task descriptions
   Source code

4: Answer questions
   Task solutions
   Source code

Key Functionality of SESE

- Real-time monitoring of the experiment
- Flexibility of defining new kinds of questions and measurement scales
- Automatic recovery of experiment sessions
- Automatic backup of experimental data
- Multi-platform support for downloading experimental materials and uploading task solutions
Experience level of the subjects

- Undergrad (n=27)
- Graduate (n=32)
- Junior (n=38)
- Intermediate (n=44)
- Senior (n=41)
- Pairs (n=4x2)

Years work experience

Preliminary results: Change Effort

Total effort on the calibration task
Preliminary results: Correctness

Discussion: “Students vs. Professionals”

- The results from this experiment suggest that the effort and correctness of programming tasks depend on a complex interaction of factors such as education level, programming experience, task complexity and task size
  - For example: Kruskal-Wallis test on equality of median effort of graduate students versus junior professionals on the calibration task: \( p=0.002 \)
- Consequently, the subjects should be sampled from the population you wish to make claims about
  - For example, using graduate students in a pair programming experiment may result in biased results if your target population is pairs consisting of junior professional programmers
Conclusions

- The experiment provides a baseline of individual performances on five change tasks on three OO (Java) systems
  - Can be used to assess how different types of pairs perform on different types of tasks compared with individuals
- Simula Experiment Support Environment (SESE) enables increased realism (and scale)
  - Experiment can be located at the pair programmers’ workplace.
  - Tasks are solved using their normal development environment

Part V

Refactoring

etc.
**eXtreme Programming**

- A Software Development Process
  - Planning
  - Design
  - Coding
  - Test

**Refactoring**

- Improve the design of existing code without changing functionality
  - Simplify code
  - Opportunity for abstraction
  - Remove duplicate code
- Relies on testing to ensure nothing breaks in the process of refactoring.
Collective Code Ownership

- Code to belongs to the project, not to an individual engineer
- Any engineer has the power to modify any code
- Cleaner code
  - Engineers are not required to work around deficiencies in objects they do not own
- Faster progress
  - No need to wait for someone else to fix something

Continuous Integration

- Pair writes up unit test cases and code for a task (part of a user story)
- Pair unit tests code to 100%
- Pair integrates
- Pair runs ALL unit test cases to 100%
- Pair moves on to next task with clean slate and clear mind
- Should happen once or twice a day.
Small Releases

- Timeboxed
- As small as possible, but still delivering business value
  - No releases to ‘implement the database’
- Get customer feedback early and often
- Do the planning game after each iteration
  - Do they want something different?
  - Have their priorities changed?

40-Hour Work Week

- Kent Beck says, “. . . fresh and eager every morning, and tired and satisfied every night”
- Burning the midnight oil kills performance
- Tired developers make more mistakes, which slows you down more in the long run
- If you mess with people’s personal lives (by taking it over), in the long run the project will pay the consequences
Coding

• Use Coding Conventions
  • Considering Pair Programming, Refactor Mercilessly, and Collective Code Ownership . . . need to easily find your way around (other people’s) code

• Method Commenting
  • Priority placed on intention-revealing code
    • If your code needs a comment to explain it, rewrite it.
    • If you can't explain your code with a comment, rewrite it.

eXtreme Programming & Coding

• Coding
  – The customer is always available
  – Code must be written to agreed standards
  – Code the unit test first
  – All production code is Pair Programmed
  – Only one pair integrates at a time
  – Integrate frequently
  – Collective code ownership
  – Leave optimization until last
  – No overtime
The 13th XP Practice? 
The Stand Up Meeting

• Start day with 15-minute meeting
  • Everyone stands up (so the meeting stays short) in circle
  • Going around the room everyone says specifically:
    • What they did the day before
    • What they plan to do today
    • Any obstacles they are experiencing
  • Can be the way pairs are formed

Part VI

Test Driven Development
Testing

• Test-Driven Development (TDD)
  • Write tests before code
  • Tests are automated
  • Often use xUnit framework
  • Must run at 100% before proceeding

• Acceptance Tests
  • Written with the customer
  • Acts as "contract"
  • Measure of progress

Part VII

* Distributed Development and Agile Methods Scalability*
Problem Statement

• Collect informal data on Virtual Pairing experience
• How to achieve reasonable experience remotely

One Approach

• Use VNC (Virtual Network Computing)
  • [http://www.uk.research.att.com/vnc/download.html](http://www.uk.research.att.com/vnc/download.html)
• Cross-platform
• Does not require Windows
• Virtual Pairing does not require a telephone
  • Voice communication is required
  • Internet voice messaging is possible
Another Approach

• NetMeeting
• Good for drawing pictures
• Giving and taking control
• Can Share just one application (privacy)
• Requires Windows

VPP Experience Summary

• VPP can be done
  – Even in same office building
• Telephone Phones/headset preferred
  – Net-based voice less desirable
  – Yahoo Messenger or MSN Messenger
• Advantages (Ergonomics)
  – Two keyboards
• Privacy
• One of pair can “mute and munch”
• Often program for 5 hours at a time
Summary and Conclusions

• Not much good empirical work exists.
• VPP is really a sub area of Computer Supported Collaborative Work (CSCW)… much studied
• David Lieb’s “Distributed Cognition”
  – Distributed Cognition Lab at UCSD
• VPP is fun. Reduces “Isolation”
• Productivity gains are large

Bibliography

• http://rockfish-cs.cs.unc.edu/pubs/XPU-03.pdf
• See virtual pair programming discussion at http://c2.com/cgi/wiki?VirtualPairProgramming
Bibliography

- http://www.cse.ucsc.edu/~brianh/PairProgramBib.html

Part VIII

Languages, Platforms, and Component Infrastructures
Construction Analogy

- Analyze business processes
- Define business requirements
- Architect building
- Hire construction company
- Lay foundation
- Construct structural infrastructure
- Build façade
- Occupy

- Model business process
- Define system requirements
- Design system
- Select development environment
- Design database(s)
- Build middleware
- Build client software
- Roll out
What about Reuse?

Can’t reuse bricks and mortar
But can reuse software components

Possible Scenario

Vendor 1

Component_A
Component_B

Vendor 2

Component_C

Your Organization

Component_D

Project_1

Project_2
Supply/Demand Metaphor

- Does not scale — too simplistic
- Hinders reuse within organization
  - Independent projects may select or build nearly identical components
- In short: no management!

Supply/Demand Revisited

- Vendors
- In-House
- System Assembly

- Existing software components
  Installed/Deployed throughout organization
  License Management
- Quality assurance, testing in multiple contexts
Metaphor Summary

• Treat software components as *assets*
  • Track and manage them
• Develop software component “warehouses”
  • Manage newly acquired components
  • Store and catalog components from previous projects

What about Reuse?

Construction industry relies heavily on standards and prefabricated components
Software Engineering still in early stages
Components are Important

- Software Engineering will only become an “Engineering Discipline” through components
- What are components?
  - Objects? - Files?
  - Classes? - Code Libraries?
  - Modules? - Subsystems?

Component-based Systems

- The whole *comprises* its parts
- The parts *compose* the whole
- To compose, from the Latin *com* "together" and *ponere* "to put"
- The parts that we compose are *components*
Component-based Systems

• By definition, all software systems comprise components
• These components result from problem decomposition, a standard problem-solving technique

Thus the phrase component-based system has no more inherent meaning than "part-based whole"

So what?
Software Component

• A software component is a unit of software that conforms to a component model.

• It can be independently deployed and composed without modification according to a composition standard.

Component Interfaces

• Software entities interact with a component using the component's clearly defined and documented interface(s).

• An interaction standard defines the elements of an interface.
Example Interface

```java
Interface IShopCart {
    /** Return unique ID for newly created cart. **/
    String createShoppingCart();
    /** Add item to specified shopping cart. **/
    void addItem (String shopCartID, String itemID, int quantity);
    /** Remove item from specified shopping cart. **/
    void removeItem (String shopCartID, String itemID, int quantity);
    /** Return list of items CartItemInfo in a shopping cart. **/
    Enumeration getItems (String shopCartID);
    /** Actually checks out a cart. */
    void checkOutCart (String shopCartID);
}
```

Interface Abstraction and APIs

- Provides a mechanism to control the dependencies that arise between modules
- An application programming interface (API) is a specification, in a programming language, of those properties of a module that clients of that module can depend on
- Clients should not depend upon properties that are not specified by the API
APIs

• Design and implementation decisions that are unlikely to change specified in API
• While decisions that are likely to change are "hidden" from clients
• Information hiding makes modules substitutable (e.g., with new versions of a component)
• Hence makes systems easier to change

What APIs Can Express

• Conventional APIs - interface specifications written in programming languages - conveniently express *functional* properties
• Functional properties include the services a module provides and the signature of these services—the types and order of arguments to the service and the manner in which results are returned from the service
What APIs Can’t Express

- Conventional APIs not well-equipped to express *extra-functional* properties
- Quality Attributes or Quality of Service: Performance, accuracy, availability, latency, security, etc.
- Because APIs can not describe these properties, they can not hide them
- Modules may come to depend upon any of these properties, thus reducing the probability that one module can be substituted for another

Example

- Microsoft's Component Object Model (COM) assigns interface specifications a globally unique identifier (GUID) at creation
- Each revision of interface assigned a new GUID
- Components are binary implementations bound to interfaces they implement via interface GUIDs
- Clients linked to components via interface GUIDs
- Clean separation of interface and component, with clients and component bound directly to interfaces and only indirectly to each other
Example

• Java programming language distinguishes interface specification from class specification by introducing an interface type
• Clients depend upon interfaces and never upon implementations
• Components can implement any arbitrary number of interfaces

But

• A module implementation has properties in addition to those specified in abstract interface
• Assume that interface could express performance properties, e.g., that a particular sorting operation must exhibit complexity no worse than $M \cdot \log(N)$
• Even if a component were compliant, it might bind a particular value to $M$
• The client could conceivably come to depend upon this particular binding
Interface Dependencies

• Interfaces generally understood to specify a one-way flow of dependencies from modules that implement services to the clients that use these services
• Client has some assumptions about the service and hence comes to depend upon these assumptions
• The module interface specifies those assumptions that are sanctioned by the designer of the module

Interface Dependencies

• However, client and module are co-dependent
• A client depends upon a module to provide a service in a certain way
• And the module depends upon the client to access and use these services in a certain way
Real-World Contracts

• Contracts are *between* two or more parties
• Parties often negotiate the details of a contract *before* becoming signatories
• Contracts prescribe normative and measurable behaviors on *all* signatories
• Contracts can *not be changed* unless the changes are agreed to by all signatories

Software Component Model

• The Component Model transforms a specific technology into an *industry*

  ![Diagram](supply-demand-diagram)

• The goal is not (necessarily) to produce interchangeable parts, but to produce components that can interact with each other
Software Component Model

Defines a set of standards for:

– Interfaces
– Naming
– Metadata
– Interoperability

– Customization
– Composition
– Evolution
– Deployment

Component Model

• Interfaces
  Specification of component behavior and properties, defined using an Interface Description Language (IDL)

• Naming
  Globally Unique names for interfaces and components
Component Model$_2$

- Metadata
  Information about components, interfaces, and their relationships, APIs to services providing such information
- Interoperability
  Communication and data exchange between components from different vendors, implemented in different languages

Component Model$_3$

- Customization
  Interfaces for customizing components, needed to adapt to system under development
- Composition
  Interfaces and rules for combining components to create larger structures and for substituting/adding components to existing structures
Component Model

- Evolution
  Rules and services for replacing components or interfaces by newer versions
- Packaging and Deployment
  Packaging implementation and resources needed for installing and configuring a component

Software Component

A Component is:
- An opaque implementation of functionality
- Subject to third-party composition
- Conformant with a component model
Examples

- CORBA Component Model (CCM)
- Component Object Model (COM+)
- Enterprise JavaBeans (EJB)
- .NET
- Eclipse plugins, Flash MX, Quicktime API, …

Opaque Implementation

- “Black boxes”
- Predominant and most successful business model for software components is [still] based upon software as intellectual capital protected from disclosure
- Clients of software components should not rely upon implementation details that are likely to change
Third Party Composition

- Use of components should not depend upon tools or knowledge of the component that is in the possession of only the component provider
- Implies that a component-based system can comprise components from multiple, independent sources
- And that a system can be assembled by a third party system integrator who is not also a component supplier

Conformant with a Component Model

- What differentiates components from conventional COTS software products
- Component models prescribe how components interact with each other, and therefore express global, or architectural, design constraints
- Component-based systems are based on uniform, standard coordination schemes
Uniform Composition

• Two components can interact if and only if they share consistent assumptions about what each provides and each requires of the other
• Obviously some assumptions will refer to some unique aspect of each component, usually the function computed by a component

Uniform Composition

• But other assumptions might be standardized across all components, reducing chances for accidental mismatches that inhibit composition of components
• These standards might address how components are located, how control flow is synchronized, which communication protocol is used, how data is encoded, etc.
Quality Attributes

- Quality attributes of a system will depend upon its "software architecture"
- Standardizing the *types of component* used in a system and their *patterns of interaction* is one way to ensure that a system composed from third-party components will possess the desired quality attributes

Deployment of Components and Applications

- Success depends on the emergence of a robust market in third-party components
- A precondition for component composition is that components can be deployed from the developer environment into the composition environment
- And that applications that have been composed from components can be deployed from the composition environment into the customer environment
Component Model
Standards and Conventions

- Component types
- Interaction schemes
- Resource binding

Component Types

- Component is defined in terms of the interfaces it implements
- If a component implements three different interfaces X, Y and Z, then it is of type X, Y and Z
- A component that implements X, Y and Z is polymorphic with respect to these types—it can play the role of an X, Y, or Z at different times
Interaction Schemes

• Component models specify how components are located
• Which communication protocols are used
• How qualities of service such as security and transactions are achieved

Interaction Schemes

• Component model may describe how components interact with each other
• Includes constraints on which component types can be clients of other types, the number of simultaneous clients allowed, etc. (topology constraints)
Interaction Schemes

- And how they interact with the component framework
- Includes things relating to resource management such as component lifecycle (activation, deactivation), thread management, persistence, etc.
- Interaction schemes may be common across all component types or unique to particular types

Resource Binding

- The process of composing components is a matter of binding a component to one or more resources
- A resource is either a service provided by a framework or by some other component deployed in that framework
- A component model describes which resources are available to components
- And how and when components bind to these resources
- Deployment is the process by which frameworks are bound to components
Decompose problem into projects and subprojects

Problem to be Solved

Master Software Development Plan

Design logical component infrastructure

• Select suitable component model

• Provided by vendor – or, if a proprietary component model, must be implemented

Evaluate Success of Software System

Legend

- Project
- Subproject

Increasingly detailed refinement of design leads to an implemented component infrastructure

Logical Component Infrastructure

Evaluate Success of Software System

Analyze component infrastructure from multiple viewpoints

Component Infrastructure

Legend

- Component
- Interaction

Component Model Implementation
Component Infrastructure

- A software component infrastructure is a set of interacting software components
- The infrastructure is designed to ensure that a software system constructed using those components will satisfy clearly defined performance requirements

Designing a Component Infrastructure

- The Component Infrastructure is layered
- Layering is a strategy for decomposing complex structures
Four Layers

- User
- Workflow and Process Control
- Business Services
- Data and Operating System Services

User

- User components provide the external interface
  - Graphical user interfaces (GUI), Web-based, or batch-oriented
- Understand user interactions
- Request services from other components in response to commands issued by the user
Workflow and Process Control

- Workflow and process control components manage complex, automated business processes that interact with (potentially) many business services

Business Services

- Business service and legacy wrapping components provide the implementation of the business rules and the operational activity
- This is accomplished using internal business object implementations, or by wrapping legacy systems behind the component interface
Data and Operating System Services

- Interfaces with persistent storage (e.g., using JDBC, ODBC)
- Access Operating System-specific functionality

Design Layers and Designers
e-Commerce Example

- Goal: Develop shopping cart system to allow browser-based shopping from catalog

Catalog of items already in existing database

Use online payment system, such as PayPal

Initial Decomposition

GUI

- Presents interface

ShopCart

- Maintains all the Active shopping Carts, Able to recover from a server crash

Finance

- Knows everything about the financial aspects

Inventory

- Interacts with existing persistent storage, "Wraps" existing database system
Initial Decomposition

- GUI
- ShopCart
- Finance
- Inventory

- Request running Cost total
- Request purchase

Update Shopping Cart
Update Inventory
Query Inventory to Locate items

Discussion

- How will ShopCart store persistent information?
- Should there be two separate interfaces to Inventory (ReadOnly, Writable)?
- If a user adds an item to shopping cart, is the Inventory decremented?
- Any other issues?
Discussion

• A component model implementation is the dedicated set of executable software entities required to support the execution of components that conform to the model.
• Component model implementation enables system developers to focus on business functionality, rather than interaction plumping.

Design of Component Infrastructure

• Lead designer
  – Makes tradeoffs
  – Designs overall behavior and interaction
• Individual component designers
  – Accurately specify each component
  – Assure conformation to component model
Relationships

• Software Component Infrastructure
  – Composed of components that conform to a particular component model
  – You maintain control

• Component Model Implementation
  – Provided by a vendor (typically)
  – “Locked In” to model, not implementation

Pitfalls

• No lead designer to mediate between designers of each component
• Immature Data/System Services
• Size and Packaging of Subcontracted Components
• Distributed Development
• Large-Scale Legacy Integration is difficult
• Ambiguous Communication
Opportunities

- A product-line component infrastructure creates component systems for reuse by other teams
- Incremental component model adoption can work

Component Reference Model
Component Reference Model

- A component is a software entity that can be executed on a physical or logical device
- Implements one or more interfaces
- This reflects that the component satisfies certain obligations - a contract
- Contractual obligations ensure that independently developed components obey certain rules so that components interact (or can not interact) in predictable ways
- And can be deployed into standard build-time and run-time environments

Component Reference Model

- A component-based system is based upon a small number of distinct component types
- Each of which plays a specialized role in a system
- And is described by an interface
- A component model is the set of component types, their interfaces, and, additionally, a specification of the allowable patterns of interaction among component types
Component Reference Model

• A component framework provides a variety of runtime services to support and enforce the component model
• Component frameworks are like special-purpose operating systems

Component-Based Software Engineering

• Key technical challenge: to ensure that the properties of a system of components can be predicted from the properties of the components themselves
Motivations for CBSE

- Independent extensions
- Component markets
- Reduced time-to-market
- Improved predictability

Independent Extensions

- One problem of legacy software is lack of flexibility (lack of extensibility)
- Components are units of extension
- Component model prescribes exactly how extensions are made
**Independent Extensions**

- The framework itself may constitute the running application into which extensions (components) are deployed
- The component model and framework ensure that extensions do not have unexpected interactions
- Thus extensions (components) may be independently developed and deployed

**Component Markets**

- Component models prescribe the necessary standards to ensure that independently developed components can be deployed into a common environment
- And will not experience unanticipated interactions such as resource contention
- The integration of support services in a framework simplifies the construction of components
- And provides a platform upon which families of components can be designed for particular application niches
Reduced Time-to-Market

- Availability of components promises to drastically reduce the time it takes to design, develop and field systems
- Design time is reduced because key architectural decisions have already been made
- Embodied in the component model and framework
- Uniform component abstractions will reduce development and maintenance costs overall

Improved Predictability

- Component models and frameworks can be designed to support the most important quality attributes for particular application areas
- Component models express design rules uniformly enforced over all components deployed in a component-based system
- This means that global properties can be "designed into" the component model so that properties such as scalability, security, etc. can be predicted for the system as a whole
Compositional Reasoning

- Premise behind component certification is that there is a causal link between those properties of a component that are certified and the properties of an end system that uses that component.

Certification Authorities

- Underwriters Laboratories - *UL Standard for Safety for Software in Programmable Components*
CBSE

- Component-based software engineering is concerned with the *rapid assembly* of systems from components where
- Components and frameworks have certified properties
- These certified properties provide the basis for *predicting the properties of systems* built from components

Prediction

- Predicting the properties of a solution from its constituent parts, and the interactions among those parts (and the framework), is fundamental to all mature engineering disciplines
- The desire to achieve this result for software systems has motivated much research in software engineering science, e.g., work in "formal methods" and software architecture
- This stipulation can be viewed as the "Holy Grail" of software engineering
Certification

• It is highly improbable that an engineer could predict the properties of a system without having the benefit of knowing the properties of the parts that comprise the system
• Certification of component and framework properties implies that engineers will be working with software products that are (or have) “known quantities”
• Also implies an authoritative industrial approach to obtaining trust in the properties of the fundamental building blocks of software systems

Rapid Assembly

• Rapid assembly is important because reduced time to market is a primary motivation for adopting software component technology
• CBSE won’t be used if doing so requires substantially more development time than is already the case
• Prediction and certification are important precisely because they support the ultimate objective of reducing the time it takes to design and build software systems
Part IX

Conclusion

XP Game

• Solutions:
  – Automate Customer Tests and Make Them Easier to Write
  – Whole Team
  – Customer Speaks with One Voice
  – Release/Iteration Planning
  – Confront Technical Risks Early
  – Make Tests Run Faster
  – Simplify Stories
  – Readable Tests Provide Executable Documentation
  – Test-Driven Development
  – Brief, Daily Standup Meeting
  – Create/Update Release Plan
  – Temporarily Compromise Practices to Deliver on Time
  – Obtain Complete Local Development Environment
XP Game

• Problems
  – Cowboy Coders Ignore Team’s Process
  – Code Lacks Test Coverage
  – Stories Are Too Complicated
  – Customer Doesn’t Help Test System
  – Team Produces Few Customer Tests
  – QA Department Wants Requirements Specification
  – Team is Asked to Produce Way More Than it Can
  – System Contains Many Defects
  – Team Encounters Risky Technical Challenges
  – Management Wants System Documentation
  – Tests Not Run Before Integration
  – Tests Run Too Slowly

Selecting a Project

• On-campus customers
  – Ideally, the students themselves
• Open Source
  – Free, high quality tools
  – Easy to gain customers
XP: Pair Programming

- Better Design, Fewer Bugs
  - Quick development pace

- Knowledge Transfer
  - Students pair with experienced developers
  - Effective against very high turnover

XP: Unit Testing

- Unit tests for every non-trivial method!
  - Enforce tests are run before committing

- Confidence to make changes
  - Won’t break old functionality

- “Executable Documentation”
  - Quickly learn the code
**XP: Continuous Refactoring**

- Let students improve *any* part of the code
  - Collective ownership: sense of pride
  - Prevents fragile code

- Unit tests provide safeguard
  - Always safe to refactor!

**XP: Incremental Development**

- Break down into small tasks
  - Estimate time-to-completion
  - Keeps codebase stable

- Release frequently
  - Students get feedback from real users
  - Much more powerful than just grades
But… XP in the classroom?

- Difficult to apply XP in a course:
  - Scheduling pair programming time?
  - Maintaining unit test coverage?
  - Finding on-site customers?
  - Managing a work force?

Applying Pair Programming

- Schedule time to pair program in class
  - Two lectures, one closed lab per week?

- Allow students to choose pairs
  - Avoid scheduling conflicts
Ensure Unit Tests are Written

- *Hard* to get students in right mindset
  - Early assignments for writing tests
  - Emphasize the importance of tests!

- Test-Driven Development
  - Write the test *first*, then the code

Providing On-site Customers

- Students themselves should be customers
  - Careful selection of project

- Discuss new features, specs in class

- Also support off-campus customers
Course Management

• Experienced TA’s as Project Managers
  – Pair program with new students
  – Sustainable

• Supervise progress on tasks
  – Monitor students like employees

SourceForge.net

• Free open source project hosting

• Professional Management Tools
  – Track features/bugs online
  – Task management
  – Respond to customers
Open Source Tools

• JUnit
  – Framework for writing unit tests

• Ant
  – Build tool with XML build scripts
  – Enforce all tests pass before committing code

Case Study: DrJava

• Pedagogic IDE developed at Rice University
  – Used in intro Java courses
  – Also useful for advanced developers
DrJava: Ideal Project Candidate

- Open Source
- Full unit test coverage
  - >35% of codebase is test code
- Students can be customers!
- Other customers around the world
  - 14+ schools, 10+ countries

DrJava as Course Project

- Many small projects in pairs
  - Bug fixes, feature requests from customers
- Three major projects
  - JUnit Integration
  - Configurability
  - Integrated Debugger
Part VIII

Conclusion

Course Assignments

- Individual Assignments
  - Problems and reports based on case studies or exercises

- Project-Related Assignments
  - All assignments (other than the individual assessments) will correspond to milestones in the team project.
  - As the course progresses, students will be applying various methodologies to a project of their choice. The project and related software system should relate to a real-world scenario chosen by each team. The project will consist of inter-related deliverables which are due on a (bi-) weekly basis.
  - There will be only one submission per team per deliverable and all teams must demonstrate their projects to the course instructor.
  - A sample project description and additional details will be available under handouts on the course Web site.
Course Project

• Project Logistics
  • Teams will pick their own projects, within certain constraints: for instance, all projects should involve multiple distributed subsystems (e.g., web-based electronic services projects including client, application server, and database tiers). Students will need to come up to speed on whatever programming languages and/or software technologies they choose for their projects - which will not necessarily be covered in class.
  • Students will be required to form themselves into "pairs" of exactly two (2) members each; if there is an odd number of students in the class, then one (1) team of three (3) members will be permitted. There may not be any "pairs" of only one member! The instructor and TA(s) will then assist the pairs in forming "teams", ideally each consisting of two (2) "pairs", possibly three (3) pairs if necessary due to enrollment, but students are encouraged to form their own 2-pair teams in advance. If some students drop the course, any remaining pair or team members may be arbitrarily reassigned to other pairs/teams at the discretion of the instructor (but are strongly encouraged to reform pairs/teams on their own). Students will develop and test their project code together with the other member of their programming pair.

Sample Project Methodology

Very eXtreme Programming (VXP)

• After teams formed, 1/2 week to Project Concept
• 1/2 week to Revised Project Concept
• 2 to 3 iterations
• For each iteration:
  – 1/2 week to plan
  – 1 week to iteration report and demo
Sample Project Methodology
Very eXtreme Programming (VXP)
(continued)

• Requirements: Your project focuses on two application services
• Planning: User stories and work breakdown
• Doing: Pair programming, write test cases before coding, automate testing
• Demoing: 5 minute presentation plus 15 minute demo
• Reporting: What got done, what didn’t, what tests show
• 1st iteration: Any
• 2nd iteration: Use some component model framework
• 3rd iteration: Refactoring, do it right this time

Revised Project Concept (Tips)

1. Cover page (max 1 page)
2. Basic concept (max 3 pages): Briefly describe the system your team proposes to build. Write this description in the form of either user stories or use cases (your choice). Illustrations do not count towards page limits.
3. Controversies (max 1 page)
First Iteration Plan (Tips)

• Requirements (max 2 pages):
  • Select user stories or use cases to implement in your first iteration, to produce a demo by the last week of class
  • Assign priorities and points to each unit - A point should correspond to the amount of work you expect one pair to be able to accomplish within one week
  • You may optionally include additional medium priority points to do “if you have time”
  • It is acceptable to include fewer, more or different use cases or user stories than actually appeared in your Revised Project Concept

First Iteration Plan (Tips)

• Work Breakdown (max 3 pages):
  • Refine as engineering tasks and assign to pairs
  • Describe specifically what will need to be coded in order to complete each task
  • Also describe what unit and integration tests will be implemented and performed
  • You may need additional engineering tasks that do not match one-to-one with your user stories/use cases
  • Map out a schedule for the next weeks
  • Be realistic – demo has to been shown before the end of the semester
2nd Iteration Plan (Tips):
Requirements

- Max 3 pages
- Redesign/reengineer your system to use a component framework (e.g., COM+, EJB, CCM, .NET or Web Services)
- Select the user stories to include in the new system
  - Could be identical to those completed for your 1st iteration
  - Could be brand new (but explain how they fit)
- Aim to maintain project velocity from 1st iteration
- Consider what will require new coding vs. major rework vs. minor rework vs. can be reused “as is”

2nd Iteration Plan (Tips):
Breakdown

- Max 4 pages
- Define engineering tasks, again try to maintain project velocity
- Describe new unit and integration testing
- Describe regression testing
  - Can you reuse tests from 1st iteration?
  - If not, how will you know you didn’t break something that previously worked?
- 2nd iteration report and demo to be presented before the end of the semester
2nd Iteration Report (Tips): Requirements

- Max 2 pages
- For each engineering task from your 2nd Iteration Plan, indicate whether it succeeded, partially succeeded (and to what extent), failed (and how so?), or was not attempted
- Estimate how many user story points were actually completed (these might be fractional)
- Discuss specifically your success, or lack thereof, in porting to or reengineering for your chosen component model framework(s)

2nd Iteration Report (Tips): Testing

- Max 3 pages
- Describe the general strategy you followed for unit testing, integration testing and regression testing
- Were you able to reuse unit and/or integration tests, with little or no change, from your 1st Iteration as regression tests?
- What was most difficult to test?
- Did using a component model framework help or hinder your testing?
Project Presentation and Demo

• All Iterations Due
• Presentation slides (optional)

Readings

• Readings
  • Slides and Handouts posted on the course web site
  • Slides and Handouts posted on the course web site
  • Documentation provided with software engineering tools
• Project Frameworks Setup (ongoing)
  • As per references provided on the course Web site
• Individual Assignment
  • See Session 9 Handout: “Assignment #5”
• Team Assignment
  • See Session 10 Handout: “Team Project” (Part 3)
Next Session:  
Software Verification and Validation

- Integration and System Testing
- Static Confirmation
- Dynamic Testing
- Traceability Matrices
- Automated Testing
- Summary
  - (Individual) Assignment #7
  - Project (Part 4) ongoing