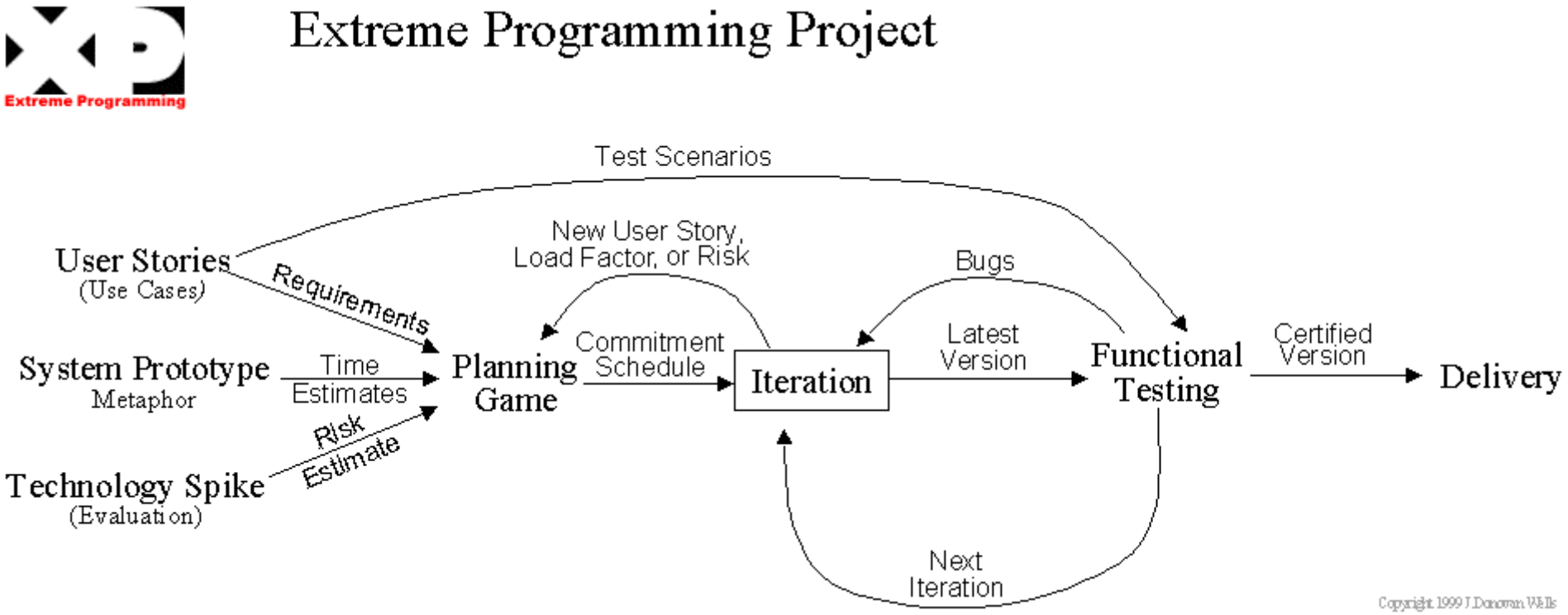
XP Methods

(http://www.extremeprogramming.org/rules.html)

1. Planning
2. Designing
3. Coding
4. Testing

Method Interrelationships



1. Planning

1. User stories are written.
2. Release planning creates the schedule.
3. Make frequent small releases.
4. The Project Velocity is measured.
5. The project is divided into iterations.
6. Iteration planning starts each iteration.
7. Move people around.
8. A stand-up meeting starts each day.
9. Fix XP when it breaks.

1.1 User Stories

- Used for time estimation and planning
  - 1, 2 or 3 weeks of *ideal development time*
- Replace requirements documents
- Written by the customer
- Three sentences
- Source of acceptance tests
- Focus on user needs

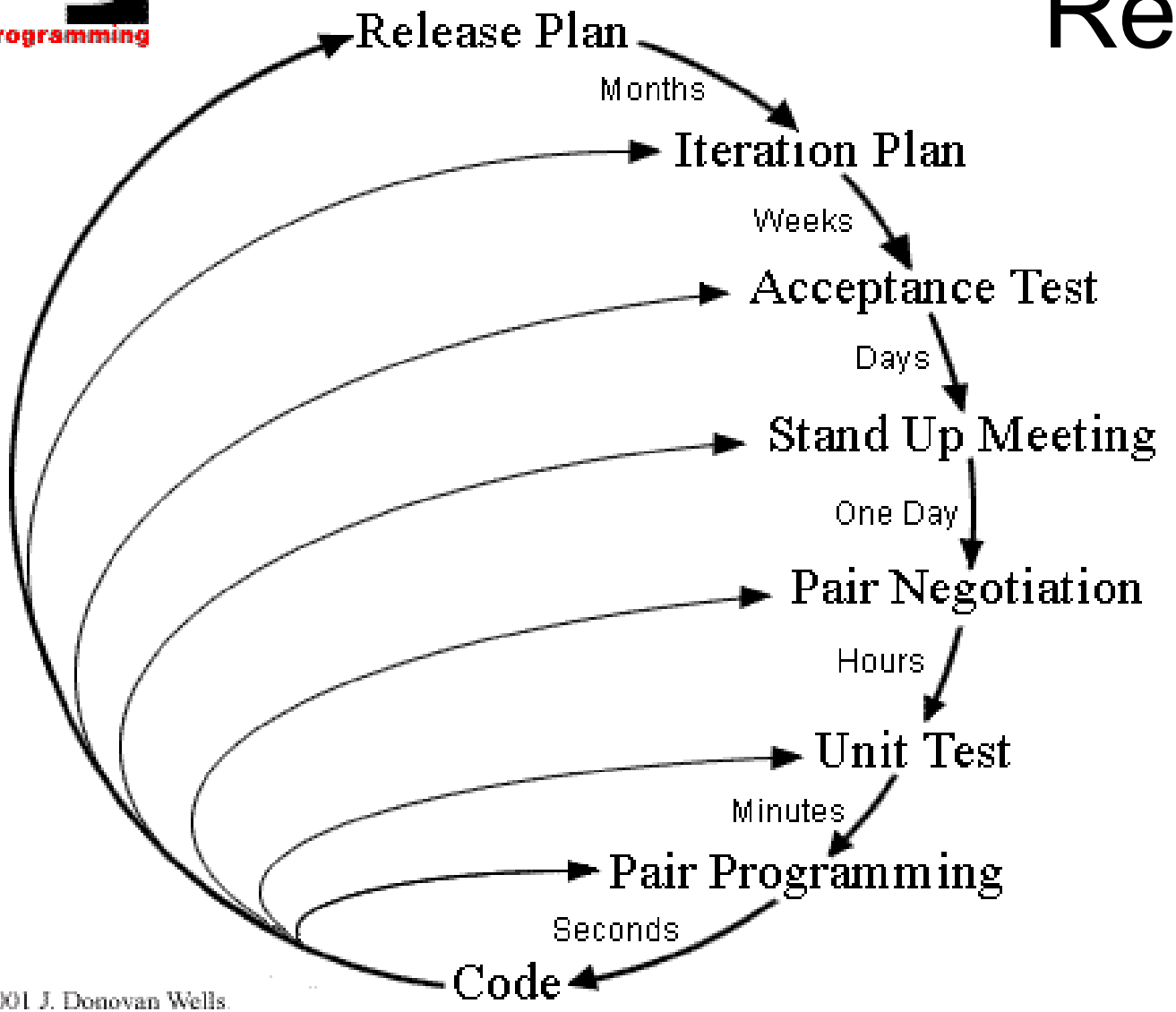
1.2 Release Planning

- Release = iteration*
- Iteration = user story*
- Time or scope (feature) boxed
- Project velocity
- Just-in-time iteration planning
- "scope, resources, time, and quality"
  - Management sets 3 of 4; development the other

1.3 Small Releases

Extreme Programming

Planning/Feedback Loops



1.4 Project Velocity

- Add up estimates for the user stories that were finished during an iteration
  - Use this as a limit for the next iteration
  - [actual time/user-story estimates = fantasy factor]
  - [Available time/fantasy factor = estimated time allowed]
- Total up the estimates for the programming tasks finished during the iteration
  - [Same adjustment as above]
- Must still make an initial (uninformed) estimate

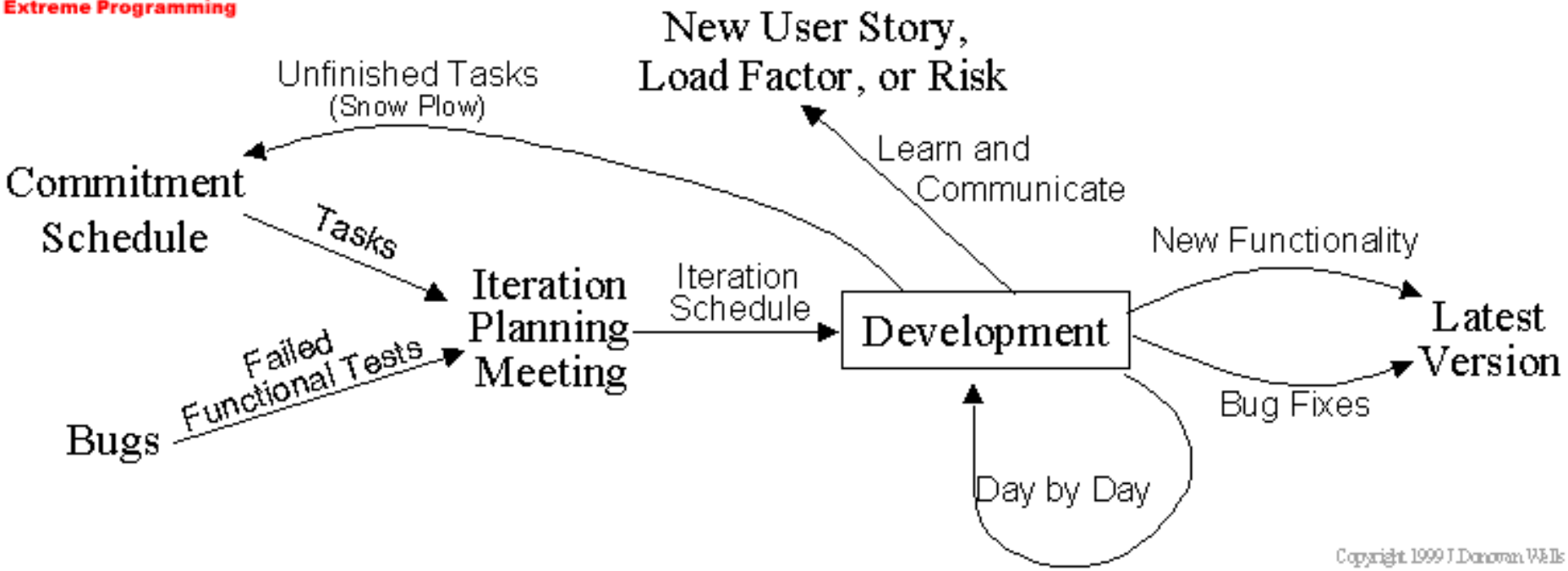
1.5 Iterative Development

- Each iteration 1-3 weeks
- Constant over the course of the project
  - *Heartbeat*
  - Promotes accurate, velocity-based estimates

1.6 Iteration Planning

- 1-3 weeks per iteration
- Based on prioritized user stories and failed acceptance tests
  - *Snow plowing*
- Project velocity from the last iteration is used to determine how much to do
- Programming tasks written on index cards
  - Each task is 1-3 *ideal programming days*
- Developer selection and time estimation

Iteration



1.7 Move People Around

- Cross training
- Risk reduction strategy
- Pair programming

1.8 Daily Stand Up Meeting

- Problems, solutions, focus
  - Requires co-presence and synchrony

1.9 Process Improvement

- Explicit rules

2. Designing

1. Simplicity.
2. Choose a system metaphor.
3. Use CRC cards for design sessions.
4. Create spike solutions to reduce risk.
5. No functionality is added early.
6. Refactor whenever and wherever possible.

2.1 Simplicity

- "Do the simplest thing that could possibly work"
- Little, if any, up-front design

2.2 System Metaphor

- Class and method naming consistency

2.3 CRC Cards

- Design level
- Simulated execution of a user story
- Completed cards can serve as documentation

2.4 Spike Solutions

- Risk-reduction strategy
- Throw-away solution to a programming problem

2.5 No Early Functionality

- Avoid implementing future requirements
- Avoid added generality

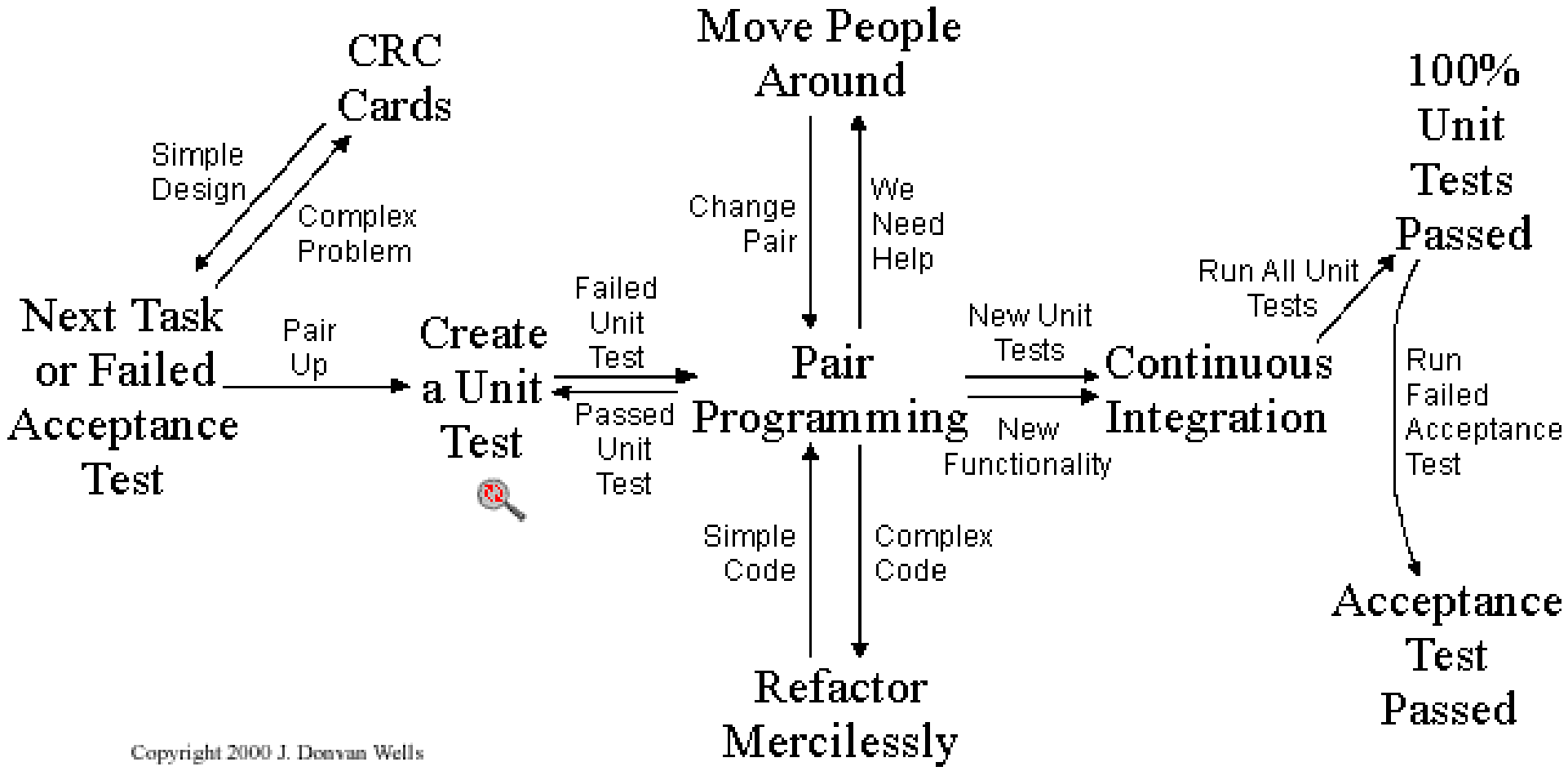
2.6 Refactor Mercilessly

- Replaces up-front design
  - Amortization
- Bad smells
- Refactoring catalog

3. Coding

1. The customer is always available.
2. Code must be written to agreed standards.
3. Code the unit test first.
4. All production code is pair programmed.
5. Only one pair integrates code at a time.
6. Integrate often.
7. Use collective code ownership.
8. Leave optimization till last.
9. No overtime.

Collective Code Ownership



3.1 On-Site Customer

- Customer is part of the development team
- Writes user stories
  - Supplements with details
- Negotiate priorities
- Helps create test data

3.2 Coding Standards

- Agreed to before hand
- Promotes collective code ownership

3.3 Test First

- Write the unit tests before writing the code
- Firms up requirements
- Helps define when the coding is done
- One test; then simplest code to satisfy it; then another; ...
- Actually speeds things up

3.4 Pair Programming

- Improved quality without reduction in productivity
- Tactics and strategy
- Jelling

3.5 Sequential Integration

- Sequential $\Rightarrow$ clear cut *latest* version
- Requires a locking mechanism
  - Physical token
  - Single machine

3.6 Integrate Often

- Source code repository updates several times per day
- Forces frequent communication and rapid response to bugs
- One pair at a time integrates
  - [Frequent updates required]

3.7 Collective Code Ownership

- Anyone can change any code
- Unit tests protect integrity
- [Egoless programming] - Weinberg
- [Release bugs imply everyone stops what they are doing to help fix the problem]
- Improved understanding of the code
  - Risk reduction

3.8 Optimize Last

- Measure first
- "Make it work, make it right, then make it fast"

3.9 No Overtime

- Cannot be sustained
- Adding resources typically fails
- Better to reduce scope

4. Testing

1. All code must have unit tests.
2. All code must pass all unit tests before it can be released.
3. When a bug is found tests are created.
4. Acceptance tests are run often and the score is published.

4.1 Unit Tests

- Framework
- All code
- Write tests before code
- Protects your code when others change it
- Enables refactoring
- Enables frequent integration

4.2 Tests Control Release

- Tests released with code
- No release without all tests being passed

4.3 Unit Tests for Bug Fixes

- Acceptance test
- Leads back to unit tests

4.4 Acceptance Tests

- Created from user stories
- Black box system tests
- Verified by customer
- Used as a progress metric [hurdle scoring]

Critique

- Volatile requirements
- Small groups
- Modest projects; functionality dominant
- Customer availability
- Limited external document requirements
- Discipline replaces management control
- Teamwork