Extreme Programming (XP)

Md Joynal Abedin Joy

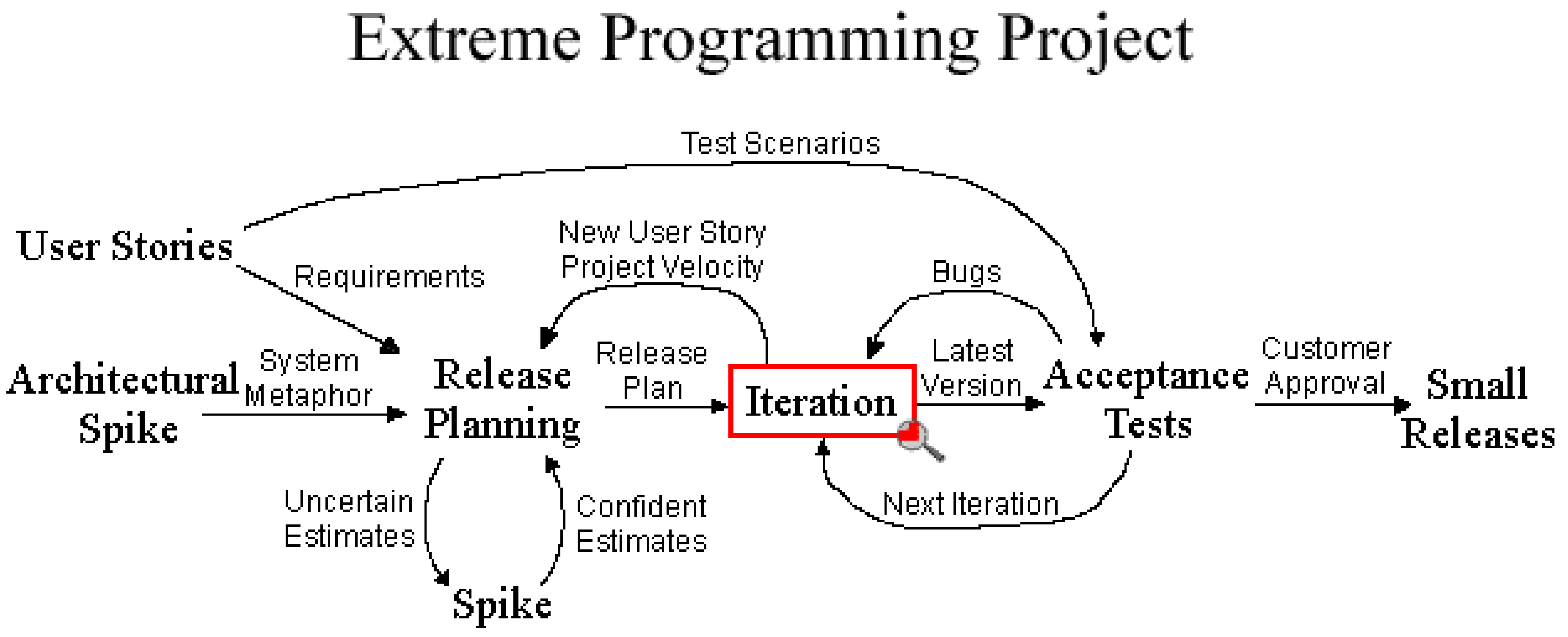
IT-21058

3rd year 2nd semester

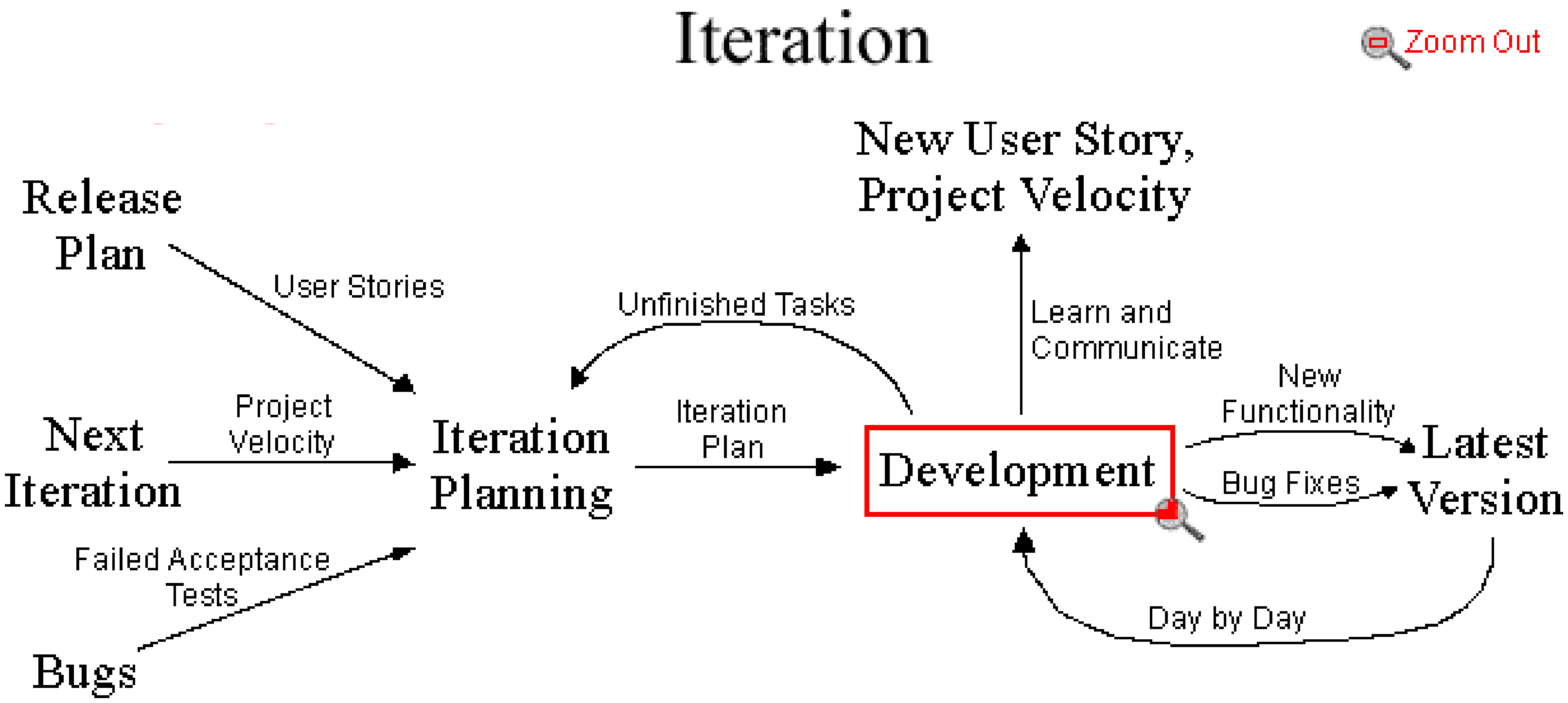
Introduction Extreme Programming?

- An agile development methodology XP is “a light-weight methodology for small to medium-sized teams developing software in the face of vague or rapidly changing requirements
- It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation?
- Created by Kent Beck in the mid 1990’s
- A set of 12 key practices taken to their “extremes”
- A mindset for developers and customers

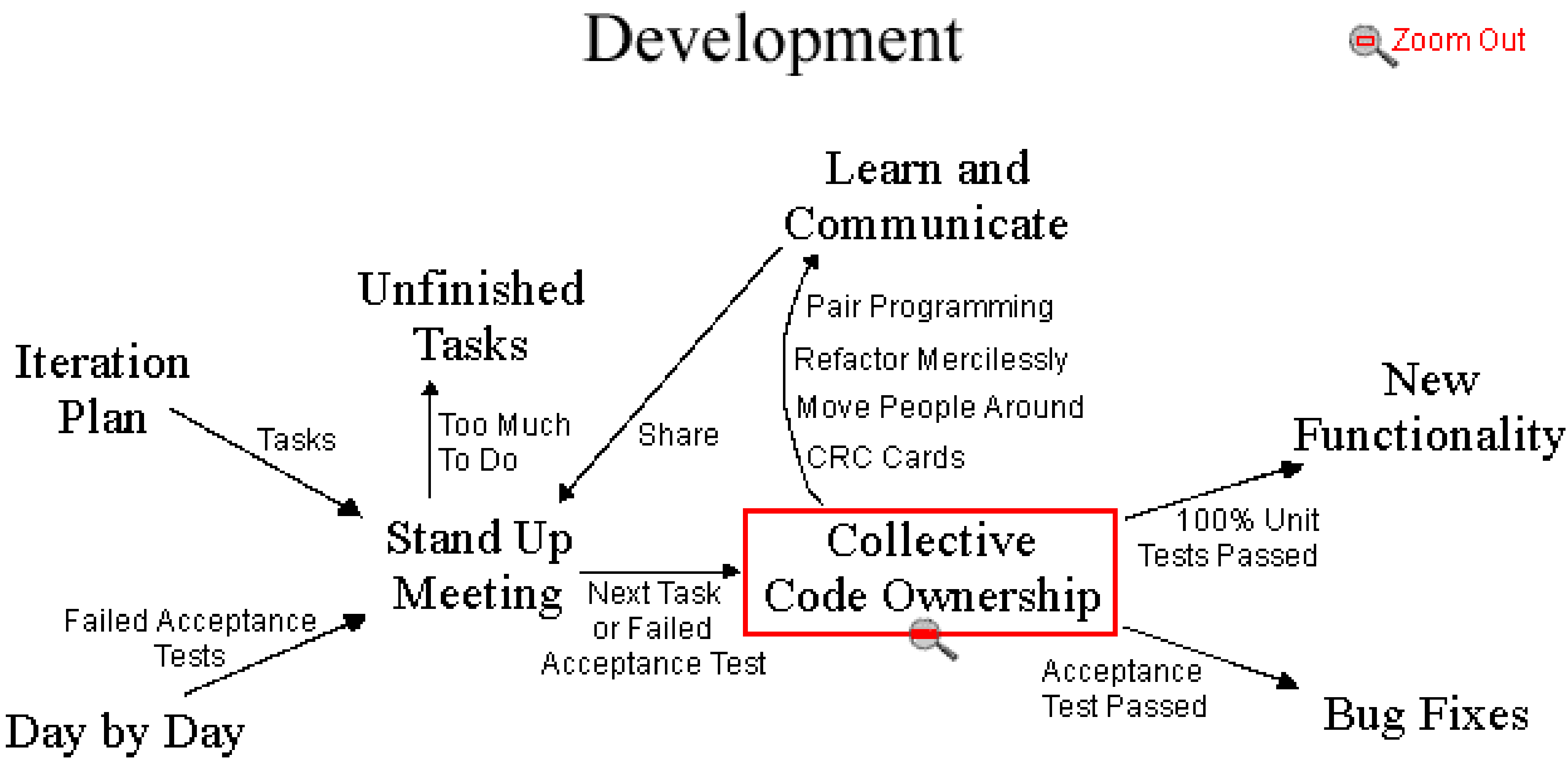
XP MODEL


Extreme Programming Project
Test Scenarios
User Stories
Requirements
New User Story
Project Velocity
Bugs
Architectural Spike
System Metaphor
Release Planning
Release Plan
Iteration
Latest Version
Acceptance Tests
Customer Approval
Small Releases
Uncertain Estimates
Confident Estimates
Spike
Next Iteration

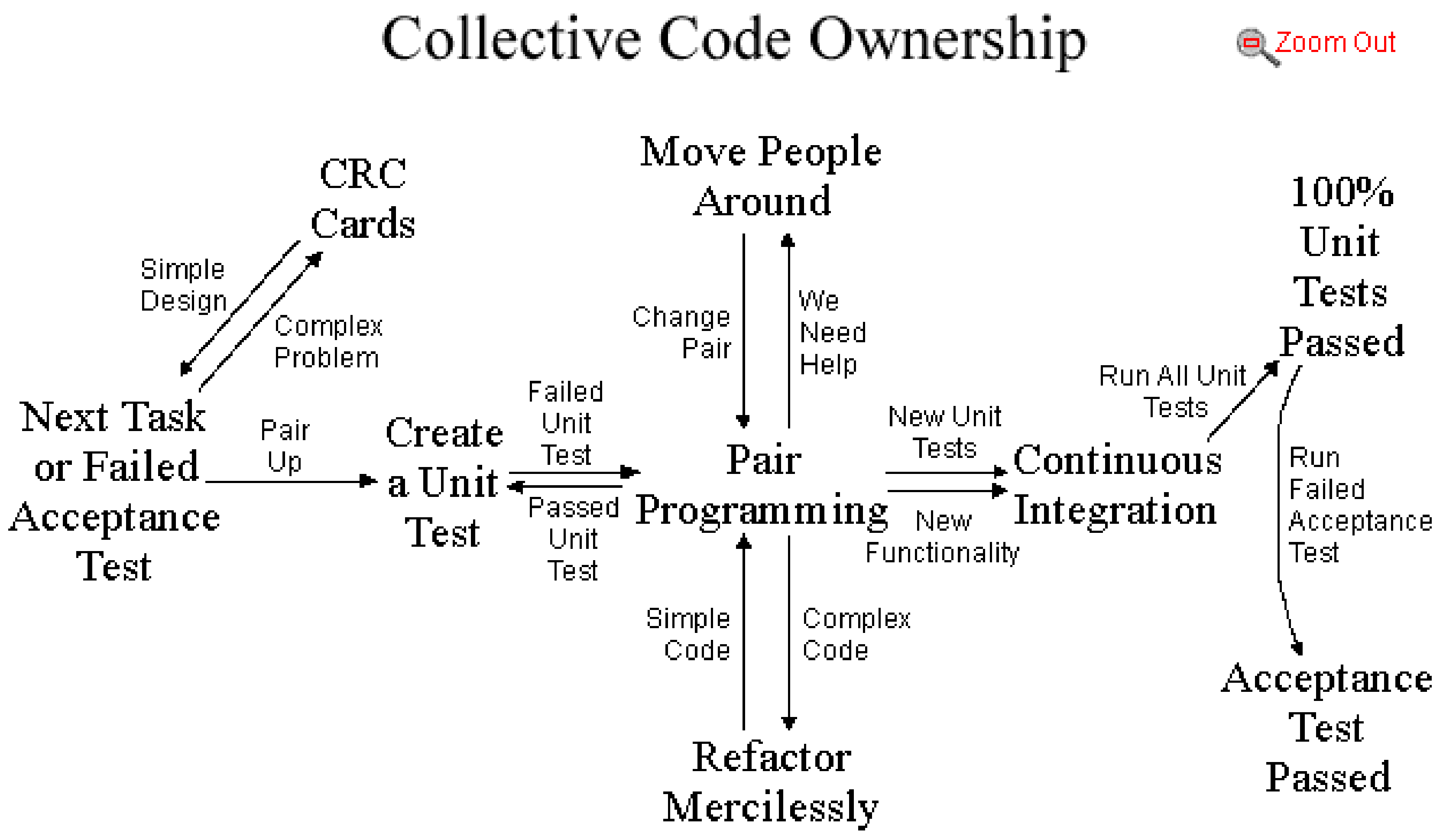
XP emphasizes iteration



XP emphasizes communication



Test-driven development



Four Core Values of XP

- Communication
- Simplicity
- Feedback
- Courage

XP Practices

- The Planning Game
- Small Releases
- Metaphor
- Simple Design
- Testing
- Refactoring
- Pair Programming
- Collective Ownership
- Continuous Integration
- 40-Hour Workweek
- On-site Customer
- Coding Standards

The Planning Game

- Planning for the upcoming iteration
- Uses stories provided by the customer
- Technical persons determine schedules, estimates, costs, etc
- A result of collaboration between the customer and the developers

Advantages

- Reduction in time wasted on useless features
- Greater customer appreciation of the cost of a feature
- Less guesswork in planning

Disadvantages

- Customer availability
- Is planning this often necessary?

Small Releases

- Small in terms of functionality
- Less functionality means releases happen more frequently
- Support the planning game

  Advantages
  - Frequent feedback
  - Tracking
  - Reduce chance of overall project slippage

  Disadvantages
  - Not easy for all projects
  - Not needed for all projects
  - Versioning issues

Simple Design

- K.I.S.S (Keep it simple Stupid)
- Do as little as needed, nothing more

Advantages

- Time is not wasted adding superfluous functionality
- Easier to understand what is going on
- Refactoring and collective ownership is made possible
- Helps keeps programmers on track

Disadvantages

- What is “simple?”
- Simple isn’t always best

Testing

- Unit testing
- Test-first design
- All automated

  Advantages
  - Unit testing promote testing completeness
  - Test-first gives developers a goal
  - Automation gives a suite of regression test

  Disadvantages
  - Automated unit testing isn’t for everything
  - Reliance on unit testing isn’t a good idea
  - A test result is only as good as the test itself

Pair Programming

- Two Developers, One monitor, One Keyboard
- One “drives” and the other thinks
- Switch roles as needed

Advantages

- Two heads are better than one
- Focus
- Two people are more likely to answer the following questions:
  - Is this whole approach going to work?
  - What are some test cases that may not work yet?
  - Is there a way to simplify this?

Disadvantages

- Many tasks really don’t require two programmers
- A hard sell to the customers

Extreme Programming Roles

- **Customer**
  - Writes User Stories and specifies Functional Tests
  - Sets priorities, explains stories
  - May or may not be an end-user
  - Has authority to decide questions about the stories
- **Programmer**
  - Estimates stories
  - Defines Tasks from stories, and estimates
  - Implements Stories and Unit Tests
- **Coach**
  - Watches everything, sends obscure signals, makes sure the project stays on course
  - Helps with anything

Extreme Programming Activities

- **Coding:** You code because if you don't code, at the end of the day you haven't done anything.
- **Testing:** You test because if you don't test, you don't know when you are done coding
- **Listening:** You listen because if you don't listen you don't know what to code or what to test
- **Designing:** And you design so you can keep coding and testing and listening indefinitely (good design allows extension of the system with changes in only one place)