Introduction to Software Engineering Methods

Agile Software Development (Extreme Programming)

Literature used

- http://www.extremeprogramming.org/start.html
- Kent Beck. Extreme Programming Explained: Embrace Change, Publisher: Addison-Wesley Professional; 1st edition (October 5, 1999)
- Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts. Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional; 1st edition (June 28, 1999)

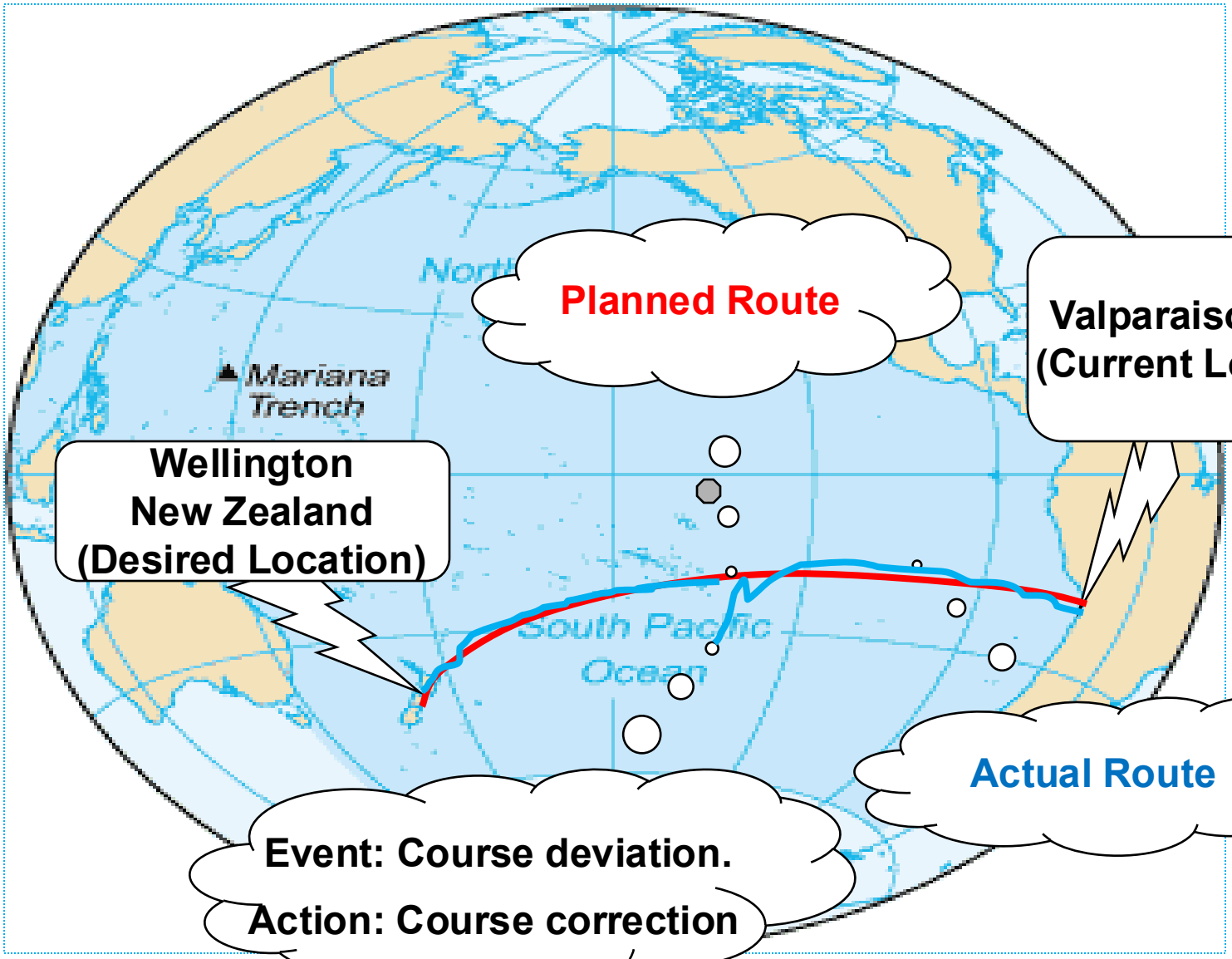
Introduction Content

- Introduction to Agile methods
- Practices of eXtreme Programming (XP)
- XP Process
- Programming XP
- Refactoring
- Exploration phase
- Planning phase
- Iteration planning
- Summary

How much Planning?

- Two styles of navigation [Gladwin 1964]
  - European navigation:
    - Current Location and Desired Location
    - Planned Route
    - Route Deviation and Route Correction
  - "Polynesian navigation"
    - No plan but objectives
    - Situational behavior

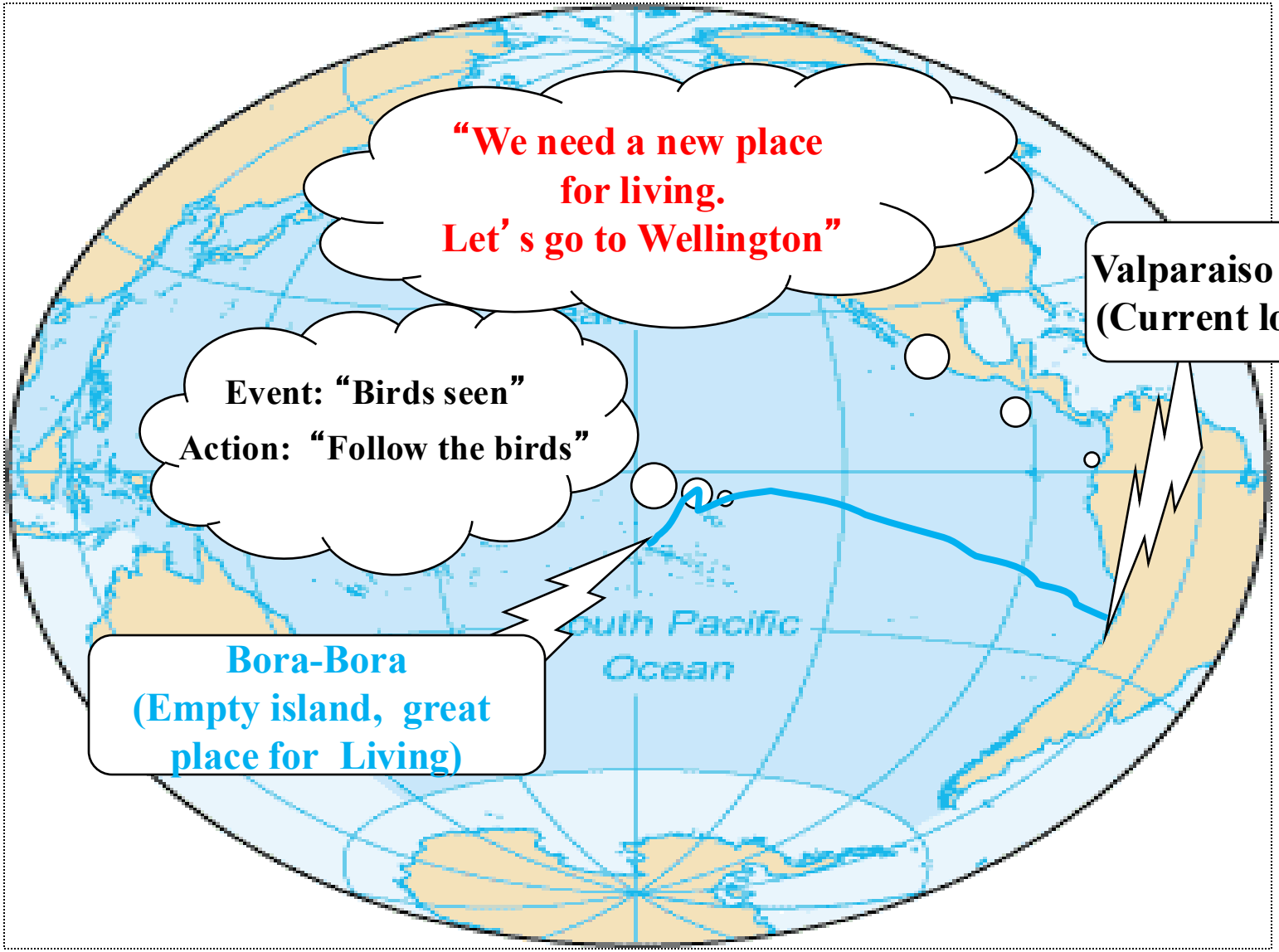
“European Navigation”



Auckland Project Plan (European Navigation)

- Project Goal: Wellington
- Desired Outcome: Wellington is found
- Team: Captain and 60 sailors
- Organization: Flat hierarchy
- Tools: Compass, speed meter, map
- Methods: Determine planned course, write planned course before departure.
- Work breakdown structure
    - Task T1 (Check direction): Determine current direction of ship
    - Task T2 (Compute deviation): Determine deviation from desired course
    - Task T3 (Course Correction): Bring ship back on course
- Process:
  – T1 and T2 are executed hourly. If there is a sufficient deviation, T3 is executed to bring the ship back on the planned course.
- Schedule: With good wind 50 days, if doldrums are encountered, 85 days.

Polynesian Navigation



Auckland Project Plan (Polynesian Navigation)

- Project Goal: Wellington
- Desired Outcome: A new place for living is found
- Team: Captain and 50 sailors
- Organization: Flat hierarchy
- Tools: Use stars for navigation, measure water temperature with hand
- Methods: Set up a set of event-action rules. When an event occurs, determine the action to be executed in the given context.
- Work breakdown structure
  - Task T1 (Determine direction): Set direction of ship to a certain course
  - Task T2 (Check Clouds): Look for non-moving clouds in the distance
  - Task T3 (Check Birds): Look for birds and determine their direction
  - Task T4 (Compute course): Determine new course for ship
  - Task T5 (Change course): Change direction to follow new course
- Process:
  - Start with T1. Tasks T2 and T3 are executed regularly. The result (cloud detected, birds detected, nothing happened) is interpreted in the current context. If the interpretation makes a new course more promising, execute task T4 and T5.
- Schedule: None

Situated action

- Situated action [Suchman 1990]
  - Selection of action depends on the type of event, the situation and the skill of the developer
- Examples of navigation events: “Course deviation”, “Birds seen”, “Clouds seen”.
- European Navigation is context independent:
  - Event: “Course deviation in the morning”
    - Action: “Course correction towards planned route”
  - Event: “Course deviation in the evening”
    - Action: “Course correction towards planned route”
- Polynesian Navigation is context dependent:
  - Event: “Birds seen”, Context: Morning
    - Action: “Sail opposite to the direction the birds are flying”
  - Event: “Birds seen”, Context: Evening
    - Action: “Sail in the direction the birds are flying

How much Planning?

- What is the first thing the developer knows about any software project?
- Course planning
  - Exam date and period end are primary constraints
  - “Polynesian navigation” might be reasonable but is it applicable?

What are problems?

- Trying to find answer to unanswerable questions
- Building estimates on no data
- Managing the process

Limitations of Waterfall, Spiral and UP Models

- Neither of these model deals well with frequent change
- What do you do if change is happening more frequently?

Agile software development

Manifesto of the Agile Software Development:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Adopted from: http://agilemanifesto.org/

Agile software development principles:

- The highest priority is to satisfy the customer through early and continuous delivery of valuable software
- Welcome changing requirements, even late in development.
- Deliver working software frequently
- Business people and developer must work together daily through the project
- Build a project around motivated individuals
- The most important and effective method of conveying information to and within a development team is face-to-face conversation
- Working software is a primary measure of progress
- Agile processes promote sustainable development
- Continuous attention to technical excellence and good design enhances agility
- Simplicity – the art of maximizing the amount of work not done – is essential
- The best architectures, requirements and designs emerge from self-organizing teams
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly

eXtreme Programming (XP)

- Made up of a set of simple interdependent practices
- Each step feels very small
- No up-front design

Practices of XP [Beck]

- Short releases (short cycles)
- Pair programming
- Test-Driven development (testing)
- Collective ownership
- Continuous integration
- The planning game
- Simple design
- Refactoring
- Metaphor
- 40-hours week (sustainable pace)
- Coding standards
- On-site customer

Who decide what?

The Customer gets to decide:

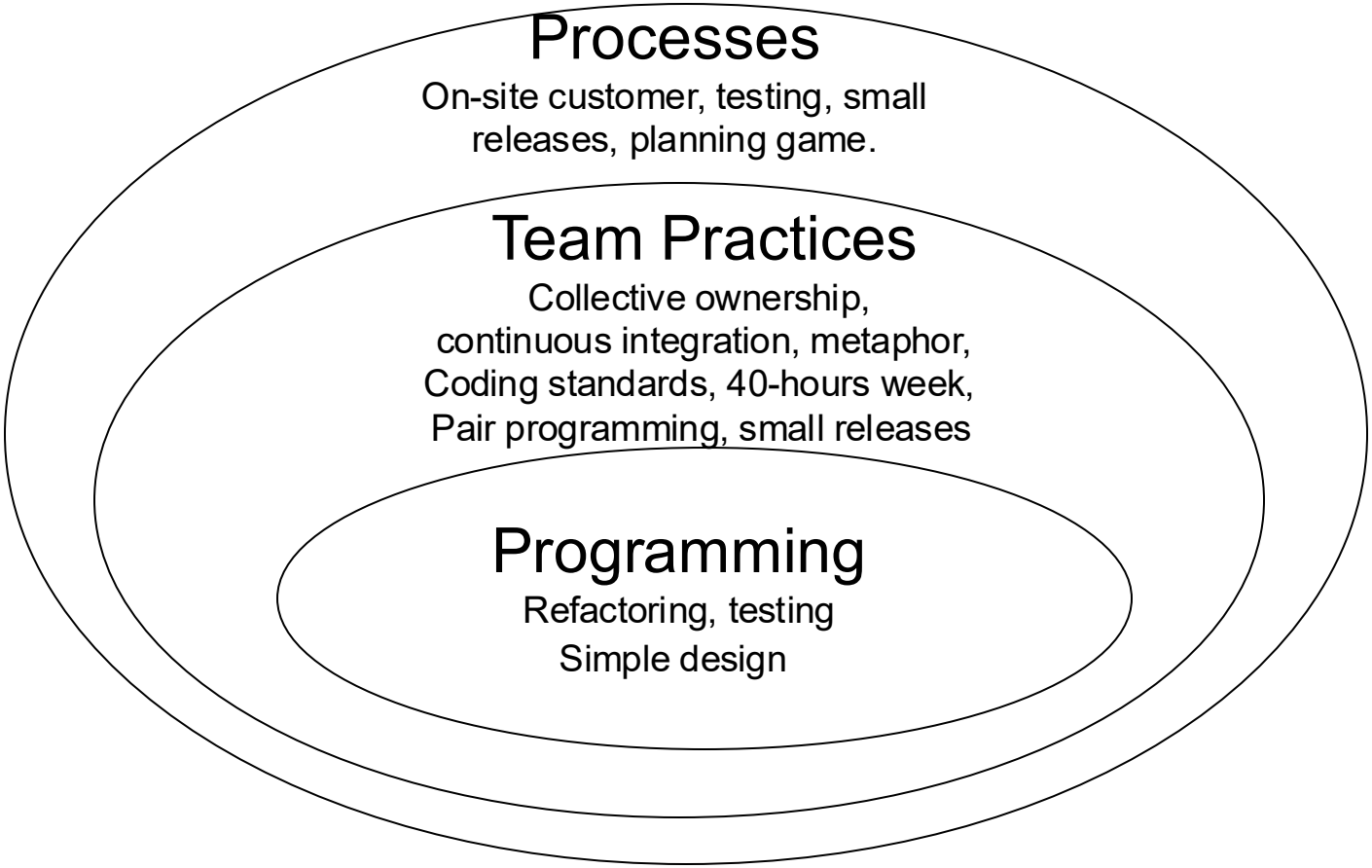
- Scope: what the system must do
- Priority: what is most important
- Composition of Releases: what must be in a release to be useful
- Dates of Releases: when the release is needed.

The Programmers get to decide:

- What it costs: Estimated time to add a feature
- Technical consequences: Programmers explain the consequences of technical choices, but the Customer makes the decision
- Process: how the team will work
- Detailed Schedule (within an iteration)

[summarized from Extreme Programming Explained]

XP practices grouped
[from Wake: Extreme programming explored]



XP Layers: XP programming practices

- Write code via incremental test-first programming
- Make tests and run them
- Code just enough to make tests work
- Refactor code but ensure that the tests work

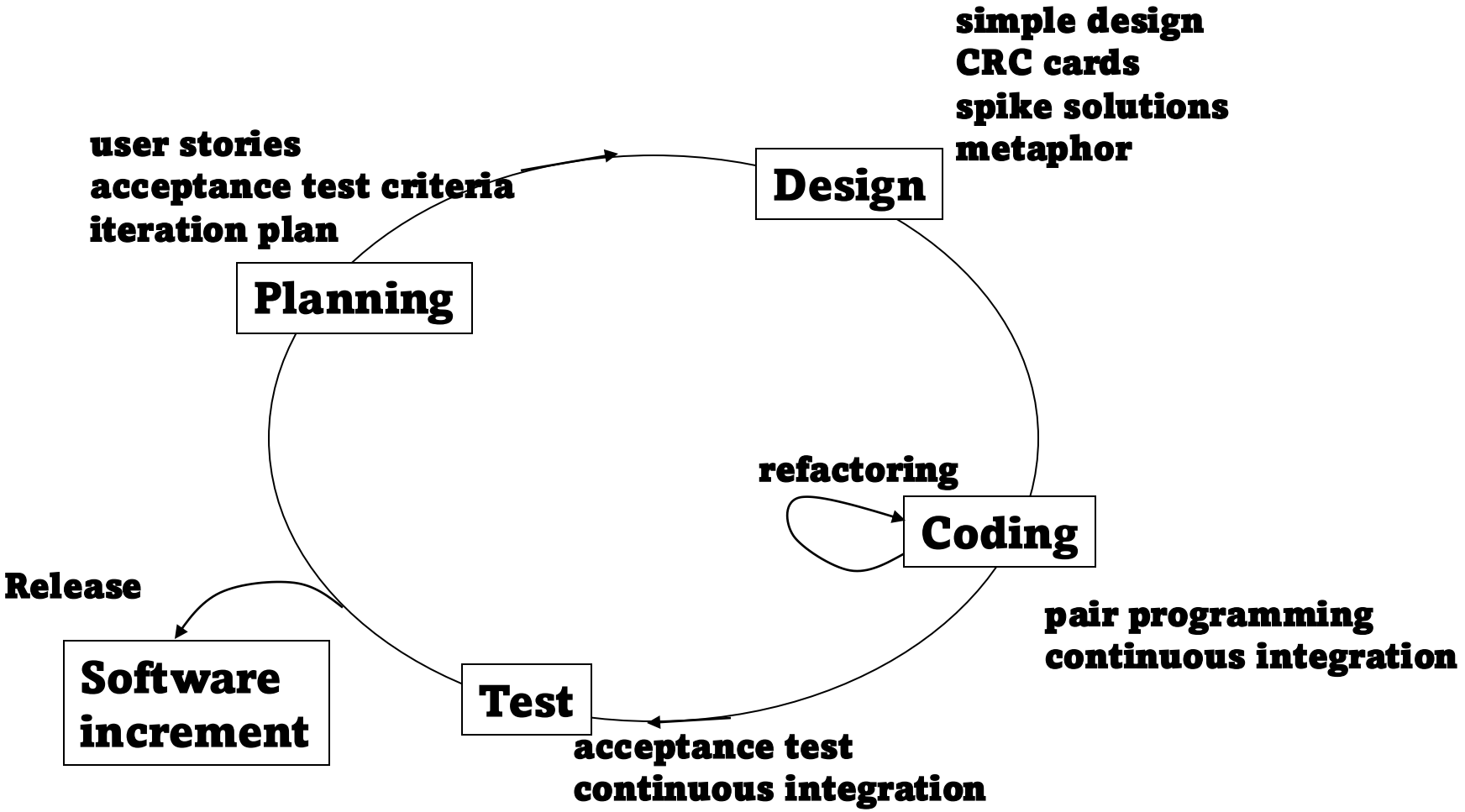
XP Layers: XP Team Practices

- On-the-spot design via pair programming
- Architecture is not a driving force but XP programs do have architecture
- The architecture is made up of metaphor, the design and development process

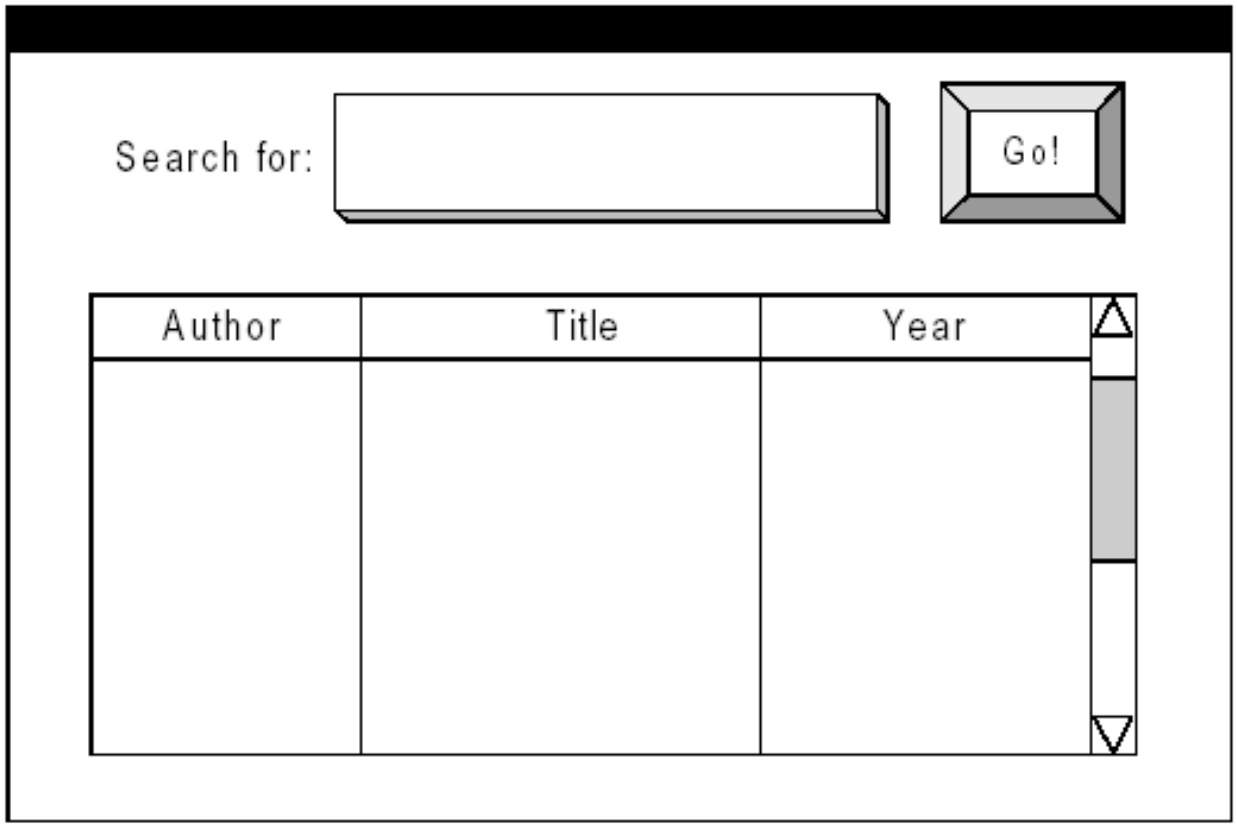
XP Layers: XP processes

- On-site customer – full-time involvement
- Testing
  - Acceptance tests by customers
  - Unit tests by developers
- Planning games
  - Iteration plan (1-3 weeks)
  - Release plan (weeks to months)

XP Process (brief)

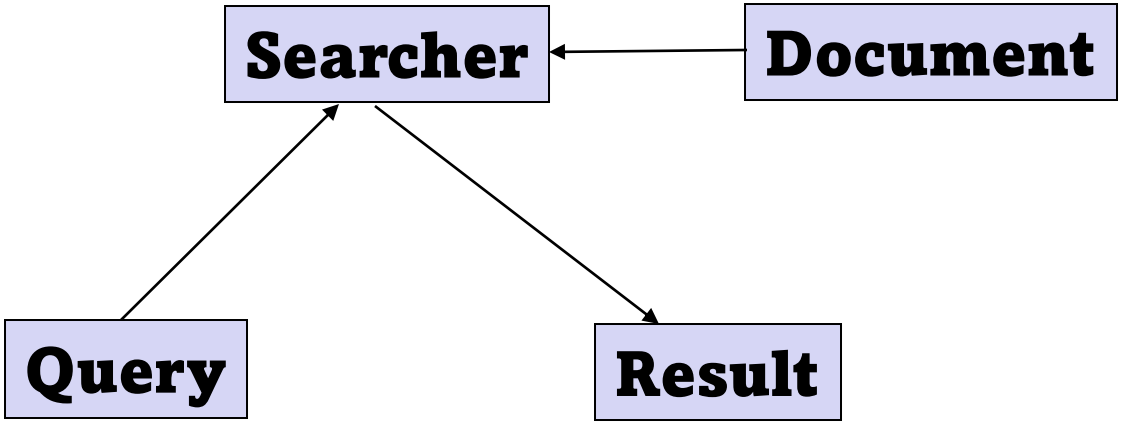


An Example (a small bibliographic system)



Adopted from XPExplores, by Wake

An Example (a small bibliographic system)



Programming in XP

- Program in small pieces (sometimes) a few lines at a time – *incremental*
- Write automated repeatable unit tests before writing code for it – *test first*

```
Public void testDocument () {
   Document d = new Document("a", "t", "y");
   assertEquals("a", d.getAuthor());
   assertEquals("t", d.getTitle());
   assertEquals("y", d.getYear());
   }
```

From W. Wake

Programming XP (test-code cycle)

1. Write a test
2. Compile
3. Implement just enough to compile
4. Run the test
5. Implement just enough to run
6. Run the test
7. Refactor to clarify and remove duplications
8. Goto 1

Implement just enough to compile

```
public class Document {
  public Document(String author,
     String title, String year){}
  public String getAuthor(){}
  public String getTitle(){}
  public String getYear(){}
}
```

From W. Wake

Implement just enough to run

```
public class Document {
   String author;
   String title;
   String year;
   public Document (){}
   public Document(String author, String title,
              String year){
              this.author = author;
              this.title = title;
              this.year = year;}
   public String getAuthor(){return author;}
   public String getTitle(){return title;}
   public String getYear(){return year;}
}
```

From W. Wake

Programming XP

- Check the empty result

```
public void testEmptyResult(){
    Result r = new Result();
    assertEquals(0, r.getCount());
    }
```

- Check non-empty result

```
public void testResultWithTwo Documents(){
        Document d1 = new Document("a1", "t1", "u1");
        Document d2 = new Document("a2", "t2", "u2");
        Result r = new Result (new Document[]{d1,d2});
        assertEquals(2, r.getCount());
        assert(r.getItem(0) == d1);
        assert(r.getItem(1) == d2);
        }
```

- The first version of Result class

```
public class Result {
   Document[] collection = new Document[0];
   public Result (){}
   public Result(Document[] collection){
        this.collection = collection;
   }
   public int getCount(){return collection.length;)}
   public Document getItem(int i) {return collection[i];}
}
```

Adopted from XPExplores, by Wake

Programming XP (more tests)

Query

```
    public void testSimpleQuery() {
      Query q = new Query("test");
      assertEquals("test", q.getValue());
   }
```

…

Just enough to run:

```
  public class Query {
      String query;
      public Query(String query) {
            this.query= query;}
      public String getValue(){return query;}
}
```

From W. Wake

Programming XP (more tests)

Searcher – empty case

```
public void testEmptyCollection() {
  Searcher searcher = new Searcher();
  Result r = searcher.find(new Query("any"));
  assertEquals(0, r.getCount());
}
```

Stub for Searcher class (for compilation)

```
public class Searcher {
  public Searcher() {}
  Result find(Query q) {}
}
```

Further concretization (for running)

```
public class Searcher {
  public Searcher() {}
  Result find(Query q) {return new Result();}
}
```

Programming XP (more tests)

- Searcher – one element collection

```
public void testOneElementCollection() {
   Document d = new Document("a", "Some title", "y");
   Searcher searcher = new Searcher(new Document[]{d});
   Query q1 = new Query("Some title");
   Result r1 = searcher.find(new Query(q1));
   assertEquals(1, r1.getCount());
   Query q2 = new Query("notThere");
   Result r2 = searcher.find(new Query(q2));
   assertEquals(0, r2.getCount());
}
```

A new constructor - for compilation

```
Document collection = new Document[0];
   public Searcher(Document[] docs) {
     this.collection = docs;
   }
```

Further concretization of **`find()`** (for running)

```
public Result find(Query q){
   Result result = new Result();
   for (int i = 0; i < collection.count; i++) {
     if (collection[i].matches(q)){
              result.add(collection[i]);
     }
   }
   return result;
}
```

Programming XP

- Lets do Document.matches()
- Test-first

```
public void testDocumentMatchingQuery() {
  Document d = new Document("1a", "t2t", "y3");
  assert(d.matches(new Query("1")));
  assert(d.matches(new Query("2")));
  assert(d.matches(new Query("3")));
  assert(!d.matches(new Query("4")));
}
```

Then program

```
public boolean matches(Query q) {
  String query = q.getValue();
  return
   author.indexOf(query) != -1 ||
   title.indexOf(query) != -1||
   year.indexOf(query) != -1;
}
```

Programming XP

Result add test:

```
public void testAddingToResult(){
    Document d1 = new Document("a1", "t1", "y1");
    Document d2 = new Document("a2", "t2", "y2");
    Result r = new Result();
    r.add(d1);
    r.add(d2);
    assertEquals(2, r.getCount());
    assertEquals (r.getItem(0),d1);
    assertEquals (r.getItem(1),d2);
}
```

Result update:

```
    …
 public class Result {
    Document[] collection = new Document[0];
    public int getCount() {return collection.size();}
    public Document getItem(int i) {
            return(Document)collection.elementAt(i); }
    public void add(Document d) {collection.addElement(d); }
}
```

From W. Wake

Programming XP (refactoring)

- Searcher, Result – refactor to convert array to Vector

```
public class Searcher {
   Vector collection = new Vector();
   public Searcher() {};
   public Searcher() {Document[] docs) {
     for (int i = 0; i < docs.length; i++) {
            collection.addElement(docs[i]);
     }
   }
   public Result find(Query q){
     Result result = new Result();
     for (int i = 0; i < collection.size(); i++) {
      Document doc = (Document) collection.elementAt(i);
      if (doc.matches(q)){
                result.add(doc);
       }
     }
     return result;
   }
}
…
public class Result {
   Vector collection = new Vector();
. . .
```

XP Team Practices (Who Owns Code?)

- Nobody
- Last one who touched
- One owner per class/package
- Sequential owners
- Ownership by layer
- Collective code ownership

XP Team Practices (Integration)

- Just before delivery
- Daily builds
- Continuous integration

XP Team Practices (Overtime)

- 40-hours week

  vs.

- Work overtime

XP Team Practices (Workspaces)

- Geographically separated Bad Practice
- Offices (1-2 persons) Bad Practice
- Cubicles Bad Practice
- Open workspace Good Practice

XP Team Practices (other considerations)

- Release scheduling
- Coding standard

Pair programming

- All production code is written by pairs of programmers working together at the same workstation
- One member of each pair drives the keyboard and type the code
- The other member of the pair watches the code being typed
- They interact intensively
- The roles change frequently
- Pair membership changes at least once per day so that every programmer works in different pairs each day

Pair programming

- Asking a partner for help (and receiving it)
- The partner helps with both strategic and detailed thinking
- The partner provides an ongoing quality check: review, refactoring, unit testing, new ideas.
- The partner provides permission to do the right thing, even if it's starting over.
- If one partner forgets something, the other can remind them.

Why pair programming works?

- The nature of brain
- The nature of the heart
- Pair programming experiences
  - Exhausting
  - Satisfying

What is architecture?

- Architecture shows up in spikes, the metaphor, the first iteration, and elsewhere:
  - Spike
  - Metaphor
  - First Iteration
  - Small Releases
  - Refactoring
  - Team Practices

What is architecture? (Spike)

- Quick throw-away explorations into the nature of a potential solution
- Based on the stories and the spikes, you decide on an approach to the system's structure
- The spikes you do in the exploration phase can guide you to a deployment architecture.

What is architecture? (Metaphor)

"The system metaphor is a story that everyone—customers, programmers, and managers—can tell about how the system works."

—Kent Beck

What is architecture? (Metaphor)

- Effective *metaphor* helps guide your solution.
- The metaphor acts partly as the logical design
- It identifies the key objects and their interactions.
- The metaphor may change over time as you learn more about the system, and as you get inspired in your understanding of it.

Why seek a metaphor?

- Common Vision
- Shared Vocabulary
- Generativity
- Architecture

Metaphor

- What if the metaphor is “wrong”?
- What if you can’t think up any good metaphors?

- What are some system metaphors that have been used?

  - Desktop metaphor for graphical user interfaces. [Ludolph et al., “The Story Behind the Lisa (and Macintosh) Interface.”]

  - Shopping carts for e-commerce sites. [Discussed in the Extreme-Programming group at egroups.com]

Metaphor: Example – Customer Service

- Naïve metaphor
- Assembly line
- Problem-solving chalkboard
- Subcontractors
- Workflow

Metaphor: some possible examples from XPExplained

- ***Mail***: letter, envelope, stamp, address, address book, mailbox, bank of mailboxes.
- ***Desktop***: file, folder, filing cabinet, trash can.
- ***Randomness***: dice, wheel, cards; sampling.
- ***Controls and gauges***: control panel, dashboard, knob, gauge, thermometer, probe points (special test points on a circuit board), TV remote, calibration.
- ***Broadcast***: radio tower, station/channel, tuning, program.
- ***Time***: Clock, calendar, timer.
- ***Data structures***: queue, stack, tree, table, directed graph, chain.
- ***Patterns***: decorator, facade, factory, state, strategy, visitor.
- ***Information objects***: Book, tombstone (final report on dead process), tape recorder, bill-of-materials, knitting needle cards (cards with holes punched indicating areas of interest, collected together by sticking knitting needles through them), money sorter, music score.
- ***Identity***: Passport, ticket, license plate, password, lock, key, keyring.
- ***Surfaces***: Window, scroll; canvas, pen, brush, palette; clipboard; overlaid transparencies; menu; typewritten page, style sheet; blackboard (AI system); map; mask, skin; mirror; form; file card.

Metaphor: some possible examples from XPExplained

- ***Dynamic objects***: outline, spreadsheet, Gantt chart, highway network.
- ***Connection***: plumbing, pipe, filter; bridge; standardized connectors, adapters; plug-in modules; breadboard (components and wires);
- ***Other Objects***: air mattress (compression); gate (monitor); checkpoint; glue (ala TeX word formatting); engine; toolbox; stacked camera lenses.
- ***Shopping***: shopping cart, checkout.
- ***Notification***: alert, alarm; lease; reservation, appointment; heartbeat.
- ***Processes***: walkthrough/drive-by/fly-by; slide show; auction, bid; registration; data mining; cooking, recipe (algorithm); voting; experiment; assembly line; double-entry bookkeeping, accounting; architecture, design; pruning; balance; garbage collection.
- ***People***: Agent, proxy, concierge (recommendations), servant, tutor, tour guide, subcontractor.

Metaphor

- Using Metaphors
  - determine metaphor during exploration phase
  - revise it during exploration
  - use it name for the “upper-most” classes
  - let it guide you during finding solution!
- Limits
  - metaphor may be a non-familiar for users
  - initial metaphor may be too weak (not powerful enough)
  - it may be too strong
  - not right metaphor may limit a conception of the system

What is architecture? (first iteration)

- The first iteration must be a functioning skeleton of the system as a whole
- The first iteration puts the architecture in place.
- You don't need all the features of any component in the first iteration
- Configure everything
- If the first iteration gets in trouble, drop any functionality you need to.
- One-Button Interface.

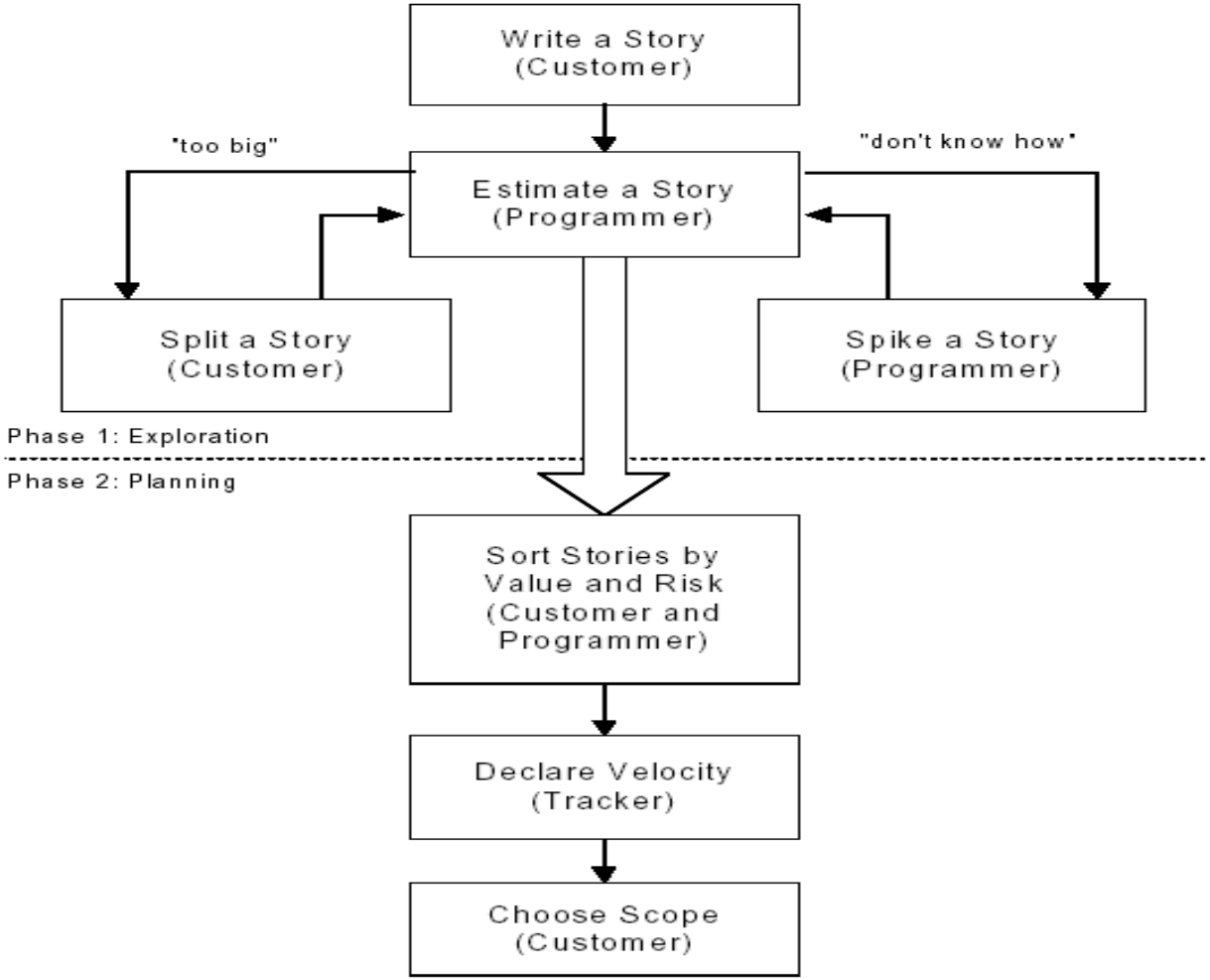
What is architecture?

- Small releases
- Refactoring
- Team practices

XP process (Release planning)

- A release is a version of a system with enough new features that it can be delivered to users outside the development group
- The goal of release planning is
  - to help the customer identify the features of the software they want
  - to give the programmers a chance to explore the technology and make estimates
  - to provide a sense of the overall schedule for everybody
- The release planning process is called the "Release Planning Game."
  - It is a cooperative game

Cooperative game



Adopted from XPExplored, by Wake

Exploration phase

- ***Goal***: Understand what the system is to do
- ***Method***: The Customer writes and manages stories. The Programmer estimates stories (spiking when necessary). Continue until all stories wanted for the planning phase are estimated.
- ***Result***: A working set of estimated stories.

Exploration phase (activities)

- Customer Writes a Story.
- Programmers Estimate a Story.
- Customer Splits a Story.
- Programmers do a Spike.

Planning phase

- ***Goal***: Plan the next release
- ***Input****:* Set of estimated stories.
- ***Method***: Do Actions
  - *Customer Sorts Stories by Value*
  - *Programmers Classify Stories by Risk*
  - *Programmers Declare the Velocity (The first time, a reasonable guess is 1/3 story point per programmer per week; after that, determine velocity empirically)*
  - *Customer Chooses Scope*
- ***Result****:* A prioritized list of the stories currently planned to be included in the next release

Planning phase

- If the Customer doesn’t like the resulting schedule:
  - change the stories,
  - release with fewer stories,
  - accept a new date,
  - change the team,
  - work with a different team entirely.

An Example of release planning

- **Objective**: Produce a system able to search an electronic library card catalog.
- **Exploration**:
  - Story: Query=>Details. “Given a query, return a list of all matching items. Look at the details of any items.” **– too big**

- **Split the story to smaller stories**:
  - **Query** “Query by author, title, or keyword.”
  - **Z39.50** “Use Z39.50 [the ANSI standard for searching library catalogs].”
  - **MARC** “Support Z39.2 MARC document format [ANSI standard].”
  - **SUTRS** “Support simple text document format (SUTRS) [From Z39.50].”

An Example of release planning
(more stories)

- **GUI** "Use a graphical user interface."
- **Sorting** "Allow sorting of results (e.g., by author or title).
- **Drill-down** "Z39.50 systems can either let you look at quick information (e.g., title, author, year), or the whole record. Instead of always showing the whole record, drill down from quick results to individual items."
- **No-drill for 1** "Don't make me drill down if the result has only one item."
- **Save** Results "Save search results to a file."
- **Print** "Print whole list of results or individual items."
- **Save** Query "Allow saving of queries."
- **Booleans** "Support Boolean queries"
- **Config** "Configure what library we're working with."
- **Portable** "System can run on a PC. Nice if it could run on Unix and Mac too."
- **Performance** "Typical query is answered in 10 seconds or less."

Sorting and Selecting
(customer sorts by value- programmer by risk)

	High Value	Medium Value	Low Value
High Risk	Z39.50 (3)	Boolean (3) Sorting (2)	
Medium Risk	Query (2) MARC (2) GUI (2)	Drill-down (2)	Print (2)
Low Risk	Performance (1)	SUTRS (1) Config (2) Portable (1)	No-drill (1) Save Result (1) Save Query (1)

Adopted from XPExplored, by Wake

Selecting stories for the next release

(F/E velocity is 5 – story points per iteration)

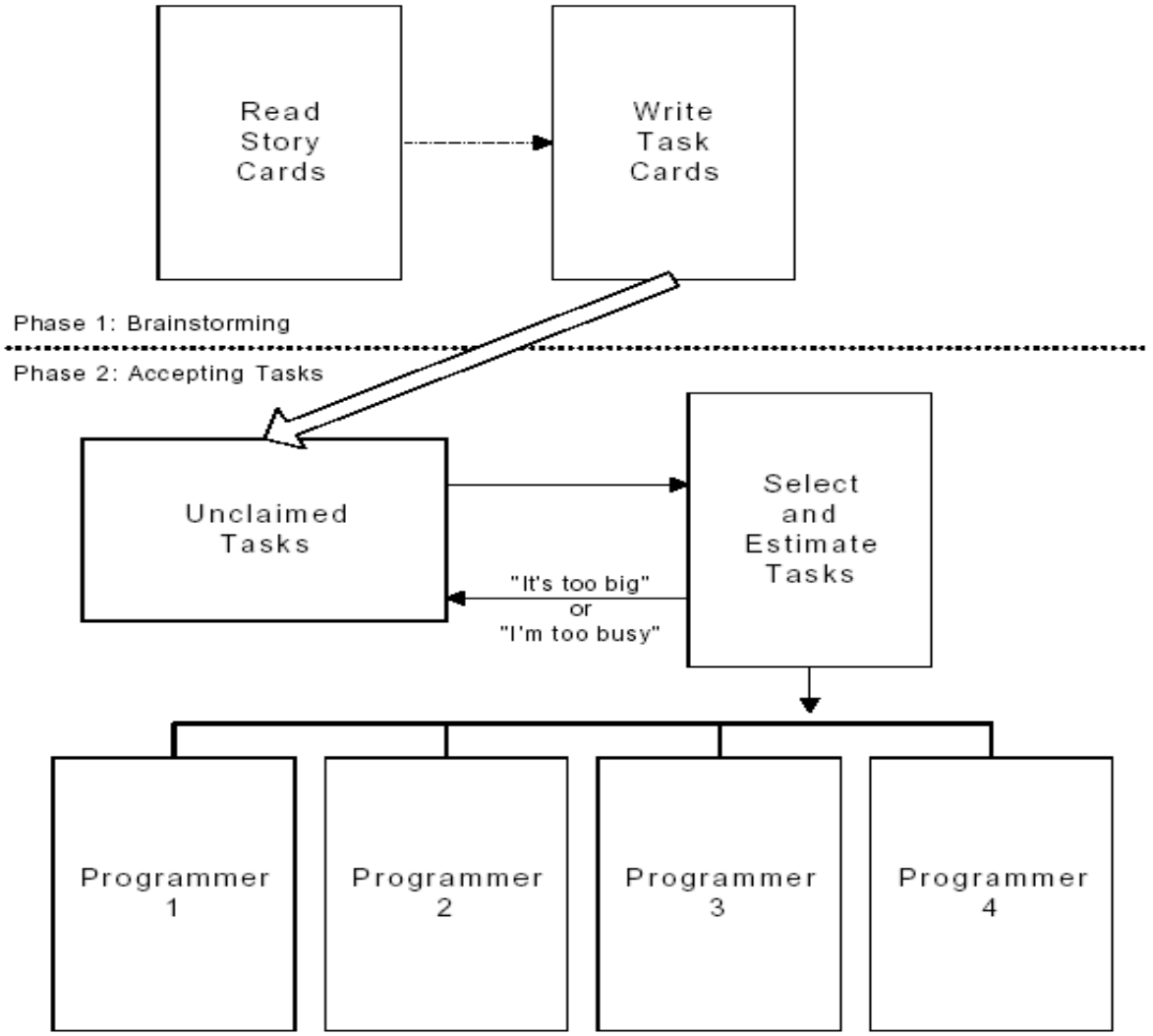
Planned Release	
Story Points	**Story name**
3	Z39.50
2	Query
2	MARC
2	GUI
1	Performance
1	SUTRS
2	Config
1	Portable

Adopted from XPExplored, by Wake

Iteration planning

- Can be thought as a board game
- Activities
  - take stories to be implemented in the iteration,
  - break them into smaller tasks
  - assign task to programmers
- Iterations are fixed length and time-boxed

Iteration Planning



Adopted from XPExplored, by Wake

Iteration planning (Setup)

- The Tracker reminds the team how many story points (“weeks”) were completed last iteration.
- The Customer selects whichever unfinished stories they want, provided the total story points fits that limit.
- The Tracker reminds each Programmer how many task points (“days”) they completed last iteration. (On the first iteration, assume 1/2 task point per programmer per day in the iteration.)
- Each Programmer puts that many coins in “their” Accepted Tasks box.

Iteration planning game (Phase 1)

- The Customer picks a story card and reads it aloud.
- The Programmers brainstorm the tasks required to implement that story. They write a task card for each task identified.
- Continue until each story card has been read.
- Move the stack of task cards to the Unclaimed Tasks pile.

Iteration planning game (Phase 2)

- Mostly the Programmers' job.
- Customer may be involved in this phase for two reasons:
  - If the Programmers can't fit in all the tasks, they'll ask to defer or split a story.
  - If they have more time than tasks, they may ask for another story.
- For each task, some Programmer will select it, estimate it in task points ("days"), and write their estimate
- If the task is bigger than 3 days ("too big"), the team can split the task and put the new tasks on the Unclaimed Tasks pile.
- If the Programmer has fewer coins than their estimate ("too busy"), they erase their estimate and return the card to the Unclaimed Tasks pile.
- Otherwise, the Programmer can accept the task, by removing as many coins as their estimate, and adding the card to their stack.
- This all happens in parallel as various Programmers claim various tasks.
- Continue until the team gets stuck, needs more stories, or wins.

Adopted from XPExplored, by Wake

Iteration planning game (finishing)

- Team Gets Stuck:
  - The Customer picks a story to split or defer.
  - Remove the corresponding tasks. Brainstorm new tasks for a split story.
  - Try Phase 2 again.
- Team Needs More Stories:
  - There are Programmers who haven't used up all their coins, but the tasks are all claimed.
  - The Customer can add in another story and see if the team can fit it in.
- Team Wins:
  - All tasks are accepted.
  - The workload is reasonably balanced (everybody has approximately 0 coins left).
- By the end of the game, the team has developed an iteration plan that should enable them to implement customer's chosen stories.
  - The Manager may record the iteration plan.
  - Customers start writing acceptance tests.
  - Programmers implement tasks.

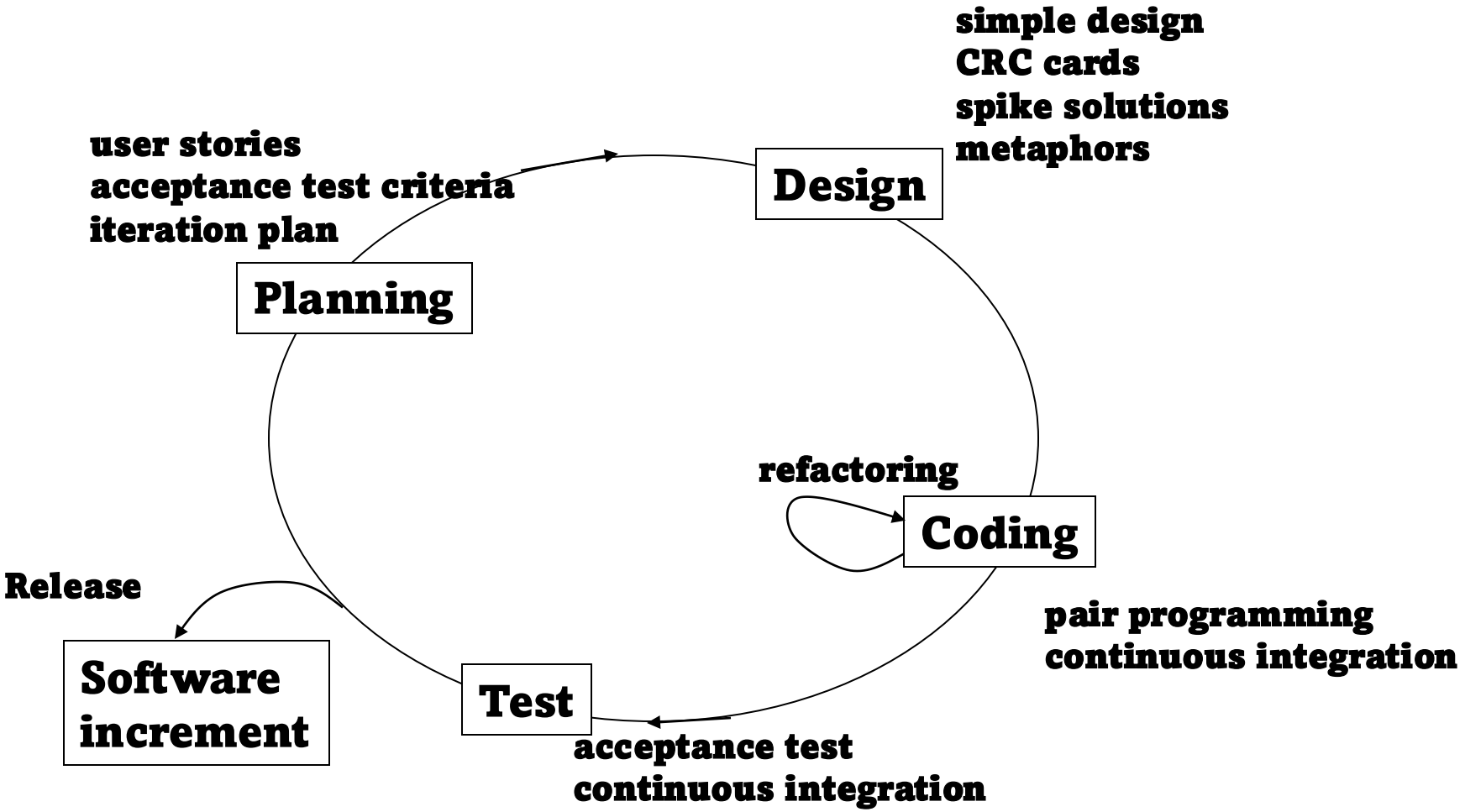
Key Ideas

- The team brainstorms tasks, but the individual programmer estimates and accepts them.
- Use the “Yesterday’s Weather” rule for both story points and task points.

Practices of XP

- User stories
  - a short description of some part/piece of the desired functionality for the software system
  - Should not be long
  - They may evolve
  - There should be clear how to test for the successful completion of the story
  - The developer writes an estimates on the card

XP Process (repeat)



Some Limitations of eXtreme Programming?

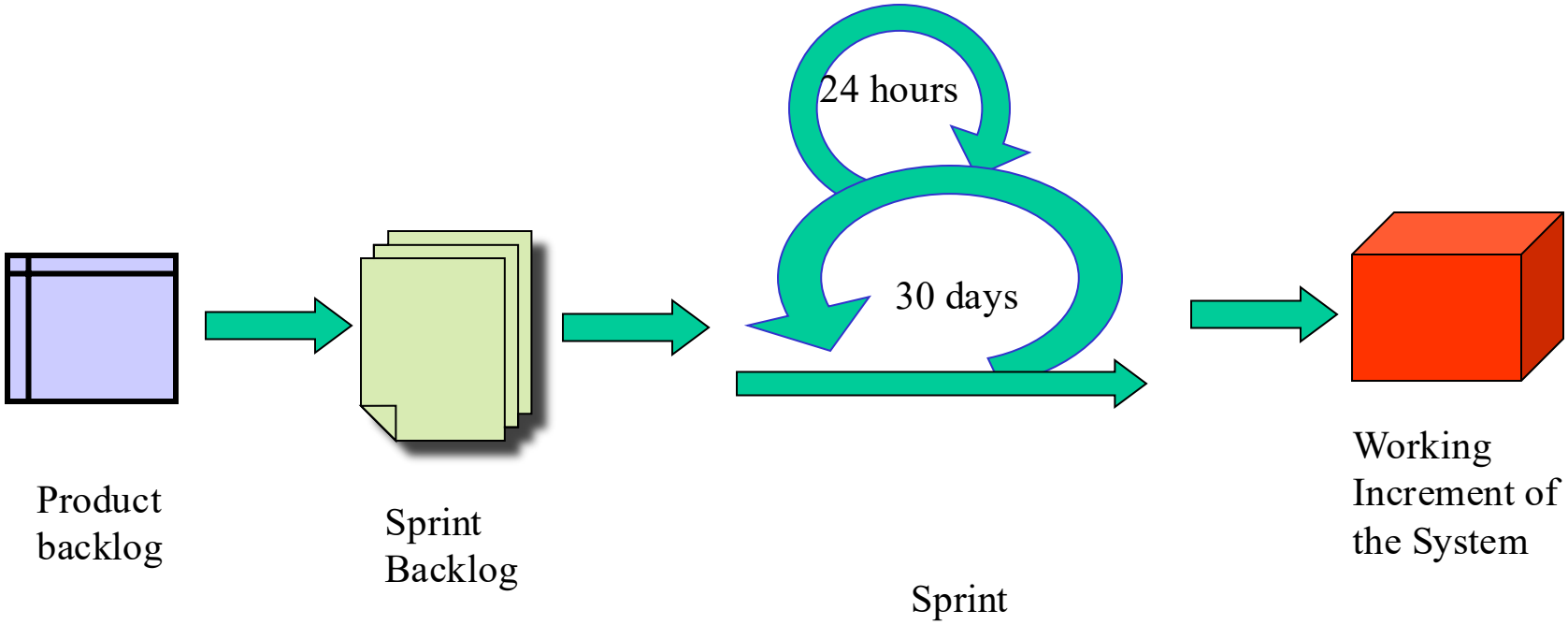
- XP only works well with small teams, six or seven is probably a reasonable limit, although it may work with up to twelve.
- XP only works with the very best software engineers. A typical team will mostly hold PhD's in computing or have equivalent industrial experience.
- XP is extremely demanding and requires very effective teamwork. Not every engineer can cope with the approach.

SCRUM

From http://en.wikipedia.org/wiki/Scrum_(rugby)

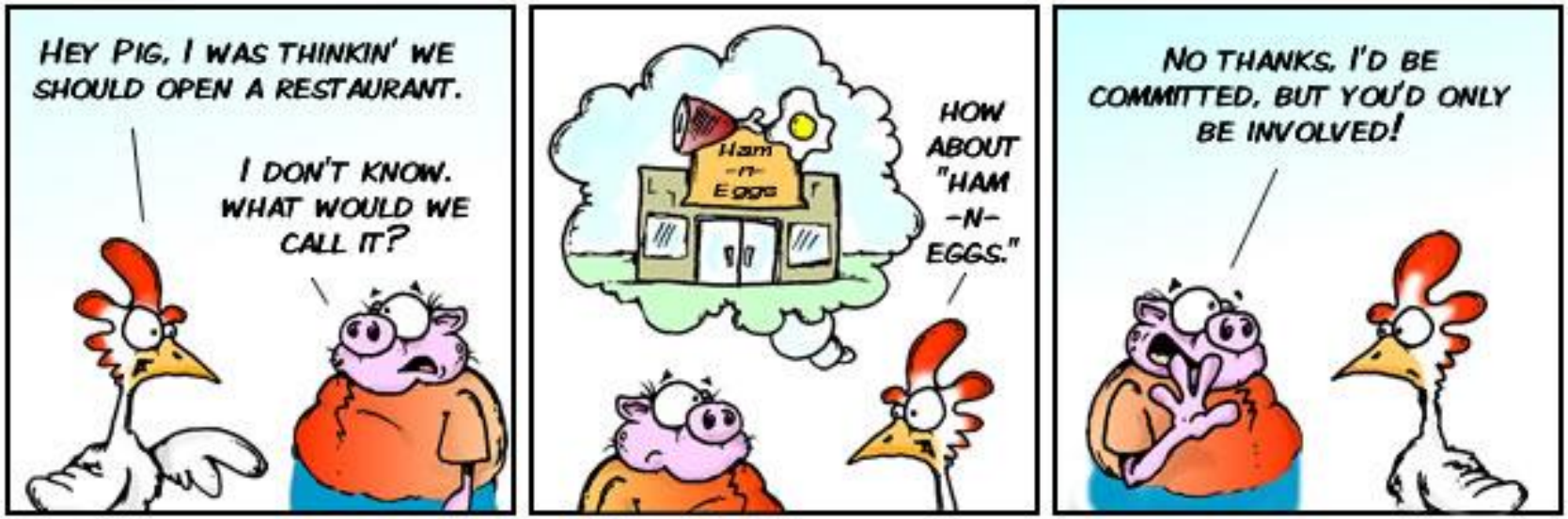
Rugby student William Webb Ellis, 17, inaugurates a new game whose rules will be codified in 1839. Playing soccer for the 256-year-old college in East Warwickshire, *Ellis sees that the clock is running out with his team behind so he scoops up the ball and runs with it in defiance of the rules*.

SCRUM: an Agile Method



Adopted from XPExplored, by Wake

Roles: Chicken and Pig



Roles

"Pig" roles

- Product Owner
- ScrumMaster (or Facilitator)
- Team

"Chicken" roles

- Stakeholders (customers, vendors)
- Managers

Meetings

- Sprint Planning Meeting
  - At the beginning of the sprint cycle (every 15–30 days), a "Sprint Planning Meeting" is held.
    - Select what work is to be done
    - Prepare the Sprint Backlog that details the time it will take to do that work, with the entire team
    - Identify and communicate how much of the work is likely to be done during the current sprint
    - Eight hours limit
- At the end of a sprint cycle, two meetings are held: the "Sprint Review Meeting" and the "Sprint Retrospective"
- Sprint Review Meeting
    - Review the work that was completed and not completed
    - Present the completed work to the stakeholders (a.k.a. "the demo")
    - Incomplete work cannot be demonstrated
    - Four hours time limit
- Sprint Retrospective
    - All team members reflect on the past sprint.
    - Make continuous process improvement.
    - Two main questions are asked in the sprint retrospective: What went well during the sprint? What could be improved in the next sprint?
    - Three hours time limit

From http://en.wikipedia.org/wiki/Scrum_(development)

Meetings

- Daily Scrum
  - The meeting starts precisely on time.
  - All are welcome, but only "pigs" may speak
  - The meeting is timeboxed to 15 minutes
  - The meeting should happen at the same location and same time every day
  - During the meeting, each team member answers three questions:
    - What have you done since yesterday?
    - What are you planning to do by today?
    - Do you have any problems preventing you from accomplishing your goal?
- Scrum of scrums
  - Each day normally after the daily scrum.
    - These meetings allow clusters of teams to discuss their work, focusing especially on areas of overlap and integration.
    - A designated person from each team attends.
  - The agenda will be the same as the Daily Scrum, plus the following four questions:
    - What has your team done since we last met?
    - What will your team do before we meet again?
    - Is anything slowing your team down or getting in their way?
    - Are you about to put something in another team's way?

From http://en.wikipedia.org/wiki/Scrum_(development)

Summary

- No explicit analysis/design models
  - Agile programming goal: Minimize the amount of documentation produced beside the code.
  - The assumption is that smaller deliverables reduce the amount of work and duplication of issues.
- Models are only communicated among participants
  - The client is seen as a “walking specification”
- Source Code is the only external model
  - The system design is made visible in the source code by using explicit names and by decomposing methods into many simpler ones (each with a descriptive name).
  - Refactoring is used to improve the source code.
  - Coding standards are used to help developers communicate using only the source code.