V22.0474-001 Software Engineering
Spring 2008

Lecture 3

Clark Barrett, New York University

Review

Software Processes

- The Waterfall Model
- The Evolutionary Model
- Component-Based

Lessons

- Each process has advantages and disadvantages
- A hybrid approach may work best
- The best process varies with the circumstances

Outline

Software Processes

- The Rational Unified Process
- Extreme Programming

Sources for today's lecture:

Rational Unified Process: Best Practices for Software Development Teams, http://www-106.ibm.com/developerworks/rational/library/253.html

Rational Unified Process Trial Download, available from http://www.ibm.com/software/awdtools/rup/

Necula, George and Aiken, Alex. *CS 169: Software Engineering*. Lecture notes, Fall 2004.

Beck, Kent. *Extreme Programming Explained*, Second Edition.

Sommerville, Ian. *Software Engineering*, Eighth Edition, ch. 4, 17.

The Rational Unified Process

The Rational Unified Process is a software engineering process in that it describes a framework for activities leading to a software product.

However it is different from the other generic processes we have discussed.

Features of RUP

- It is a software process
- It is a product sold by IBM (formerly Rational Software)
- Supported by a large array of tools
- Configurable
- Regularly updated

The Rational Unified Process

Three Perspectives of RUP

- *Best Practices* A set of principles which are integrated into the process.
- *Dynamic Perspective* Phases, iterations, milestones of the process which change over time.
- *Static Perspective* The aspects of the process which are present in some form in all dynamic phases: activities, artifacts, roles, workflows.

Best Practices of RUP

RUP identifies six best practices which should be integrated into the software development process.

Best Practices

- *Develop software iteratively* RUP supports an iterative approach that reduces risk by having frequent releases that enable user involvement and feedback. An iterative approach also helps the development team stay focused on producing results and makes it easier to track progress and accommodate changes.
- *Manage requirements* RUP describes how to elicit, organize, and document requirements. *Use cases* are developed to describe specific flows of events through the system.
- *Use component-based architectures* RUP encourages architecture design that promotes component reuse. RUP provides a systematic approach to defining an architecture using new and existing components.

Best Practices of RUP

Best Practices (continued)

- *Visually model software* RUP uses the standard Unified Modeling Language (UML) to describe the structure and behavior of architectures and components. This is useful for visualization and communication.
- *Verify software quality* Quality should be reviewed with respect to the requirements based on reliability, functionality, and performance. Quality assessment of these criteria is built into the process, in all activities, involving all participants.
- *Control changes to software* RUP describes how to control, track, and monitor changes to enable successful iterative development. RUP has support for isolation and integration of individual workspaces.

The Rational Unified Process

The dynamic perspective identifies four *phases*.

Unlike the waterfall model, where phases are equated with activities, phases in the RUP chart the progress of many concurrent activities over time.

Phases in RUP

- *Inception*
- *Elaboration*
- *Construction*
- *Transition*

Iteration The RUP supports iteration within each phase as well as iteration of the entire process if desired.

Dynamic Phases of RUP

Inception

- Goals: establish a business case for the project, delimit the project scope, identify external entities with which the system will interact.
- Deliverables:
  - Vision document
  - Initial use-case model (10-20% complete)
  - Initial project glossary, business case, risk assessment, project plan, business model
  - One or more prototypes
- Evaluation criteria:
  - Stakeholder concurrence with scope and cost/schedule estimates
  - Requirements understanding as evidenced by use-cases
  - Credibility of cost/schedule estimates, priorities, risks, development process
  - Depth and breadth of any architectural prototype developed
  - Expenditures: actual vs planned

Dynamic Phases of RUP

Elaboration

- Goals: analyze problem domain, establish a sound architectural foundation, develop project plan, eliminate risk.
- Deliverables:
  - Use-case model capturing functional requirements
  - Supplementary non-functional requirements
  - Software architecture description
  - Executable architectural prototype
  - Revised business case, risk assessment, project plan, business model
- Evaluation criteria:
  - Are the vision and architecture stable?
  - Have major risks been addressed and resolved?
  - Is the plan realistic and sufficiently detailed?
  - Do stakeholders agree that the vision can be achieved with the current plan?
  - Expenditures: actual vs planned

Dynamic Phases of RUP

Construction

- Goals: all components and features are developed and integrated, all features are thoroughly tested.
- Deliverables:
  - The software product
  - User manuals
  - Description of the current release
- Evaluation criteria:
  - Is the product stable and mature enough to be deployed?
  - Are all stakeholders ready for the transition into the user community
  - Expenditures: actual vs planned

Dynamic Phases of RUP

Transition

- Goals: transition the software product to the user community, correct problems, finish postponed features.
- Typical transition activities:
  - Beta testing
  - Parallel operation with legacy system being replaced
  - Conversion of legacy databases
  - Training of users and maintainers
  - Roll-out to marketing, distribution, sales
- Evaluation criteria:
  - Do all stakeholders agree that the vision has been realized?
  - Are the users satisfied and able to support themselves?
  - Expenditures: actual vs planned

Static Structure of RUP

Answers four questions: who, what, how, when.

Four components

- *Roles (or Workers):* Who. Defines the behavior and responsibilities of an individual or group. Roles participate in activities and are owners of artifacts.
- *Artifacts:* What. A piece of information produced, modified, or used by a process. Includes design documents, models, code, and executables.
- *Activities:* How. A unit of work assigned to a role. Usually expressed in terms of creating or updating some artifact.
- *Workflows:* When. A sequence of activities, involving one or more roles, producing an artifact.

Static Structure of RUP

The static components are further organized into nine *disciplines*: a collection of related activities related to a major area of concern within the overall project; an aid to understanding the project from a “traditional” waterfall perspective.

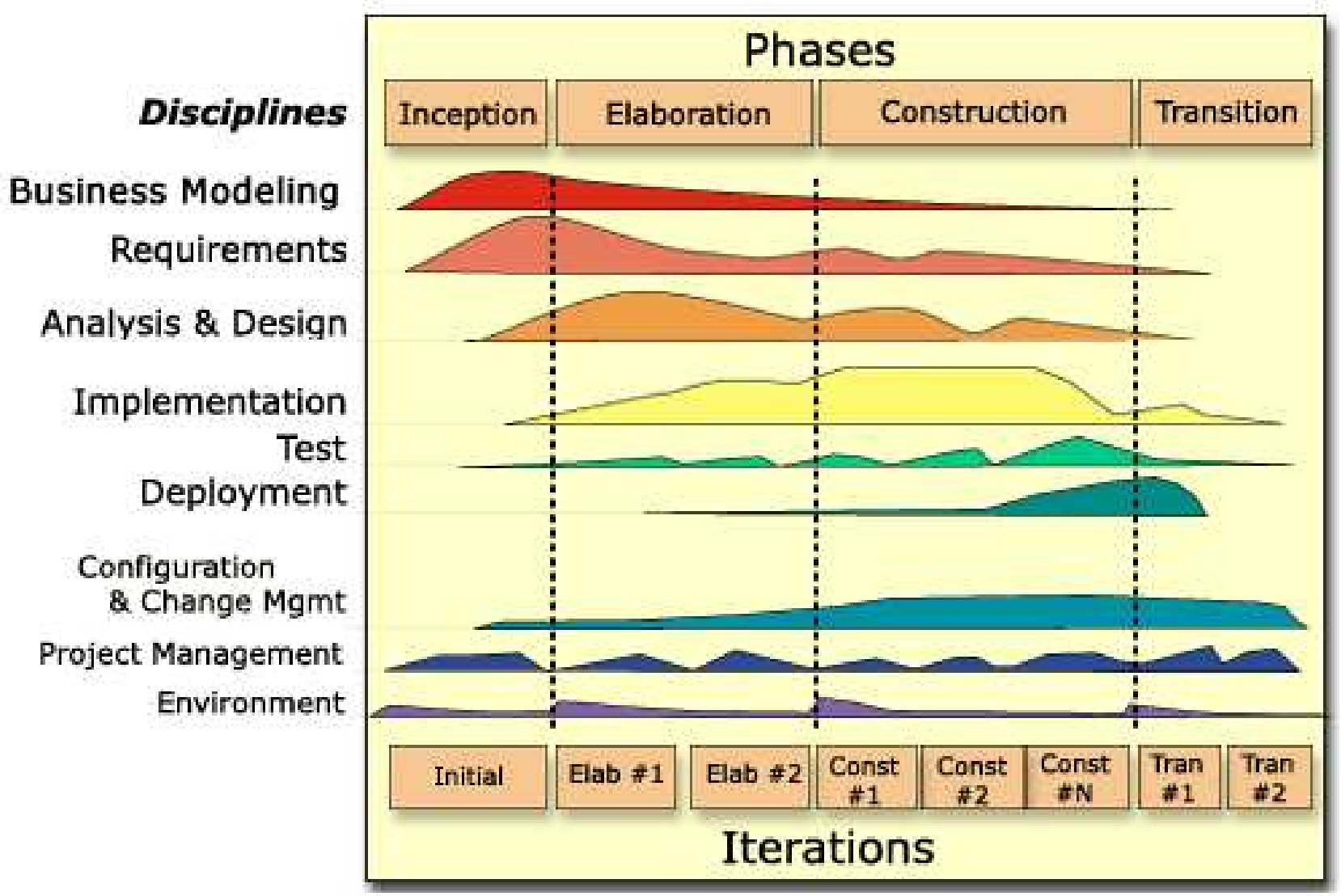
Engineering Disciplines

- Business Modeling
- Requirements
- Analysis and Design
- Implementation
- Test
- Deployment

Supporting Disciplines

- Configuration and Change Management
- Project management
- Software Development Environment

RUP Summary


Phases
Disciplines
Inception
Elaboration
Construction
Transition
Business Modeling
Requirements
Analysis & Design
Implementation
Test
Deployment
Configuration
& Change Mgmt
Project Management
Environment
Initial
Elab #1
Elab #2
Const #1
Const #2
Const #N
Tran #1
Tran #2
Iterations

RUP Summary

Discussion

What are some advantages and disadvantages of RUP?

RUP Summary

Discussion

What are some advantages and disadvantages of RUP?

- In many ways RUP represents an evolution of the traditional waterfall approach to accommodate software-specific best practices.
- RUP may be overkill for small to medium projects

Extreme Programming

Extreme Programming (XP) and other so-called *agile* methods focus on code development rather than design and documentation.

They are intended to deliver working software quickly through many short iterative cycles.

They are *revolutionary* rather than *evolutionary*.

XP includes a number of suggested practices which are different from or new to the traditional model.

XP Development Cycle

Short Cycle (2 weeks)

1. Meet with client to elicit requirements
   - User stories and acceptance tests
2. Planning game
   - Break stories into tasks, estimate cost
   - Client prioritizes stories to do first
3. Implementation
   - Write programmer tests
   - Simplest possible design
   - Refactor code often
   - Pair programming
4. Evaluate progress and repeat from step 1

XP Customer

Expert customer is part of the team

- On site, available at all times
- XP principles: communication and feedback
- Make sure we build what the customer wants

Actively involved in all stages

- Clarifies the requirements
- Negotiates with the team what to do next
- Writes and runs acceptance tests
- Constantly evaluates intermediate versions

User Stories

Write on index cards

- Meaningful title and short description
- Focus on “what” not the “why” or “how”

Uses client language

- Client must be able to determine if a story is successfully completed

Each iteration only includes a few stories

- Priority determined by customer

XP Example: Accounting Software

User stories

1. Create account

   “I can create named accounts”

2. List accounts

   “I can get a list of all accounts”

   - Question: How is the list ordered?

     “I can get an alphabetical list of all accounts”

3. Query account balance

   “I can query the account balance”

4. Delete account

   “I can delete a named account”

   - Question: Even if the balance is not zero?

     “I can delete a named account even if the balance is non-zero”

XP Customer Tests

Client must describe how user stories will be tested

- Concrete data examples
- Associated with (one or more) user stories

Customer can write and later (re)run tests

- Concrete expression of requirements

Tests should be automated

- Ensure they are run after each release

XP Example: Accounting Customer Tests

Tests are associated with (one or more) user stories

1. If I create an account *savings*, then another called *checking*, and I ask for the list of accounts I must obtain: *checking*, *savings*.
2. If I now try to create *checking* again, I get an error.
3. If I now query the balance of *checking*, I must get 0.
4. If I now try to delete *stocks*, I get an error.
5. If I now delete *checking*, it should not appear in the new listing of accounts.

XP Tasks

Each story is broken into *tasks* so that the work can be split up and the cost can be estimated.

Stories vs Tasks

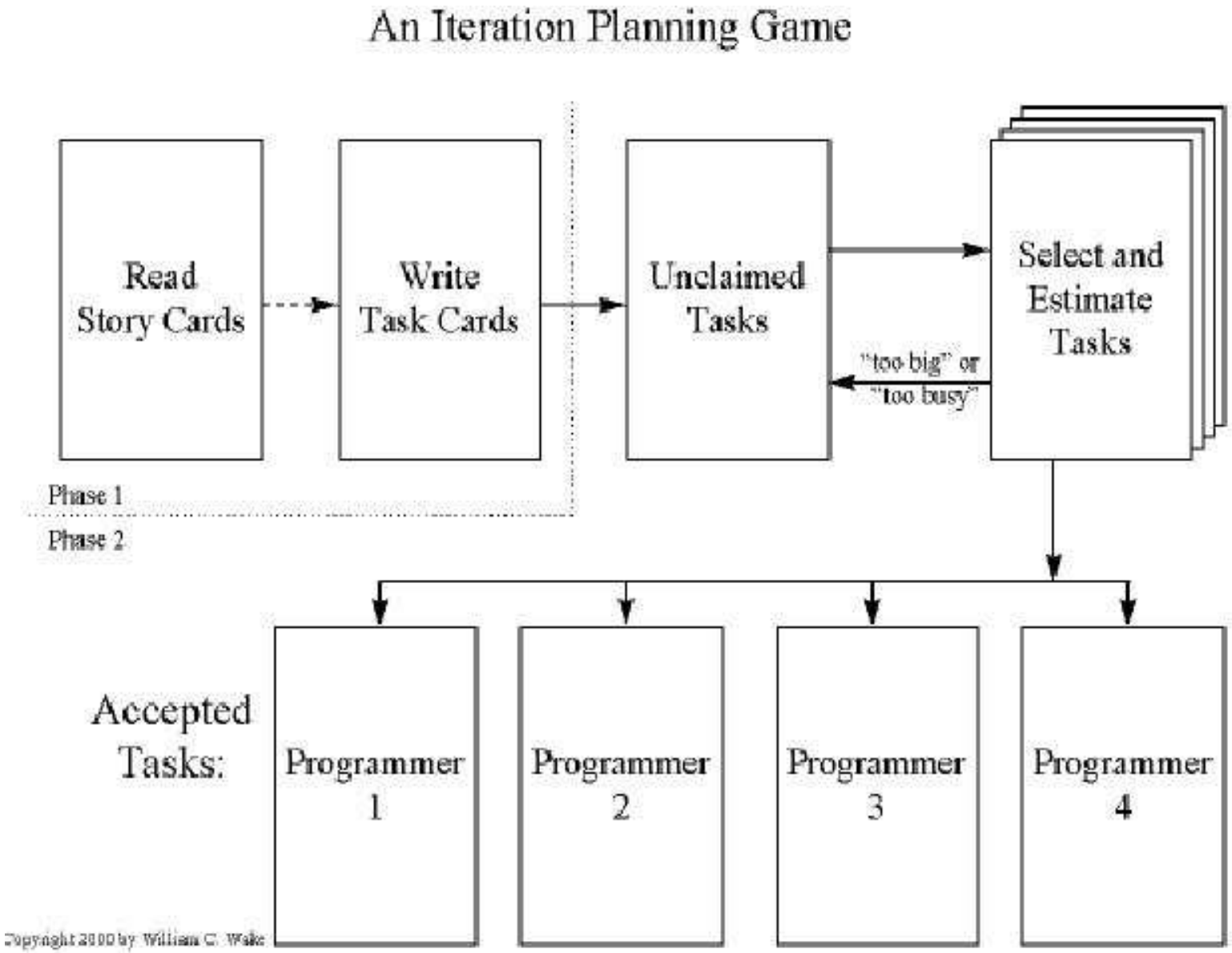
- A story is a customer-centered description: *what to do*
- A task is a developer-centered description: *how to do it*
- Example:
  - Story: "I can create named accounts"
  - Tasks: "Ask the user the name of the account"
    "Check to see if the account already exists"
    "Create an empty account"

Validate the breakdown of stories into tasks

- Check with customer
- If there are too many tasks, split story into multiple stories

Team assigns a cost to each task

XP: Play the Planning Game



XP: Play the Planning Game

Customer chooses important stories for the next release

- Selection of stories plus relative priority of stories

Development team creates and estimates cost for tasks

- Cost may be different depending on developer
- Developers can “bid” on each task

Tasks vs completion date

- If you pick a set of tasks, you can compute the completion date
- XP encourages choosing a completion date first and then picking stories/tasks until the time budget is full

XP: Test-Driven Development

Write tests *before* implementing tasks

Unit tests

- Break acceptance tests into units
- Write unit tests
- Test both good and bad behavior

Example

```
addAccount(‘‘checking’’);
if (balance(‘‘checking’’) != 0) throw...
try {
  addAccount(‘‘checking’’);
  throw...
} catch (DuplicateAccount e) {}
```

XP: Why Write Tests First?

XP: Why Write Tests First?

Testing first clarifies the task at hand

- Forces you to think in concrete terms
- Helps identify and focus on corner cases
- Ambiguous stories/tasks are caught before time is wasted on implementation

Testing forces simplicity

- Your only goal is to pass the test
- Implement simplest code that fulfills the goal
- Fight premature optimization/generalization

Useful documentation

- Exposes completely the programmer's intent

Ensures that you think about testability

- How will you know when you are done?
- In what order do you need to test components?
- What infrastructure is needed?

XP: When Tests Fail

Fail a unit test

- Fix code to pass unit test

Fail an acceptance test (from a user story)

- Not enough unit tests
- Add a unit test, then fix the code

Fail on beta-testing

- Create tests to capture failure
- Fix code to pass tests

The automated set of tests always grows, never decreases. This captures the increasing functionality of the system.

XP: Refactoring

Refactoring means changing “how” something is done, not “what” is done.

Why Refactor?

XP: Refactoring

Refactoring means changing “how” something is done, not “what” is done.

Why Refactor?

- Want code that is easy to understand and maintain
- Incremental extension outgrows initial design
- Lack of extensive early design
- Unavoidable, so just plan for it

XP: Refactoring

Refactoring means changing “how” something is done, not “what” is done.

Why Refactor?

- Want code that is easy to understand and maintain
- Incremental extension outgrows initial design
- Lack of extensive early design
- Unavoidable, so just plan for it

Refactoring and regression testing

- Comprehensive suite needed for fearless refactoring
- Plan refactoring to allow frequent regression testing
- Only refactor working code
  - Much easier to isolate problems
  - Do not underestimate the importance of small incremental steps

XP: Refactoring Examples

Repeated code

- In all conditional branches: move outside conditionals
- In several methods: create new methods
- Almost duplicate code: create new parameterized methods

Changing names

- A name should suggest what the method does and how it should be used
- Method 1: rename, then fix compiler errors
- Method 2: rename, have old method call new method, slowly update references

XP: Pair Programming

Pilot and copilot metaphor

- Pilot types, copilot monitors
- Disagreements point early to design problems
- Pairs are periodically shuffled: everyone gets to know all the code

Benefits

- Better code: instant code review
- Reduces risk: collective understanding of design and code
- Improves focus: instant feedback and advice
- Knowledge and skill migration: good habits spread

XP: Resistance to Pair Programming

XP: Resistance to Pair Programming

From programmers

- “It will slow me down”
- Stressful to relate to people all the time
- Need time alone to figure things out
- Afraid to show you are not a genius

From management

- Myth: inefficient use of personnel
  - That would be true if the most time consuming part were typing
  - Example:
    - 2 individuals: 60 loc/h each, 1 bug/33 loc
    - 1 team: 80 loc/h, 1 bug/40 loc
    - 1 bug fix costs 5 hours
    - 60 kloc program: 2 individuals: 500 development + 9000 bug fix
    - 60 kloc program: 1 team: 750 development + 7500 bug fix
- Resistance from developers: ask them to try it, find those who like it

Discussion

What are the relative advantages and disadvantages of RUP and XP?

Discussion

What are the relative advantages and disadvantages of RUP and XP?

What can apply to your projects this semester?

Discussion

What are the relative advantages and disadvantages of RUP and XP?

What can apply to your projects this semester?

- You are encouraged to adopt techniques from one or both of these processes.

Homework

Proposal Selection

- By the end of the day, I will have posted the best proposals
- Follow the link on the web page to vote on which proposal you like best
- Your vote must be in by tomorrow at midnight.
- Note: this is the easiest assignment of the semester, so be sure to do it!

Next Assignment

- Teams will be assigned on Thursday
- Your first assignment is a Requirements Document
- The Requirements Document will be due two weeks from today.