CE/CS/SE 3354 Software Engineering

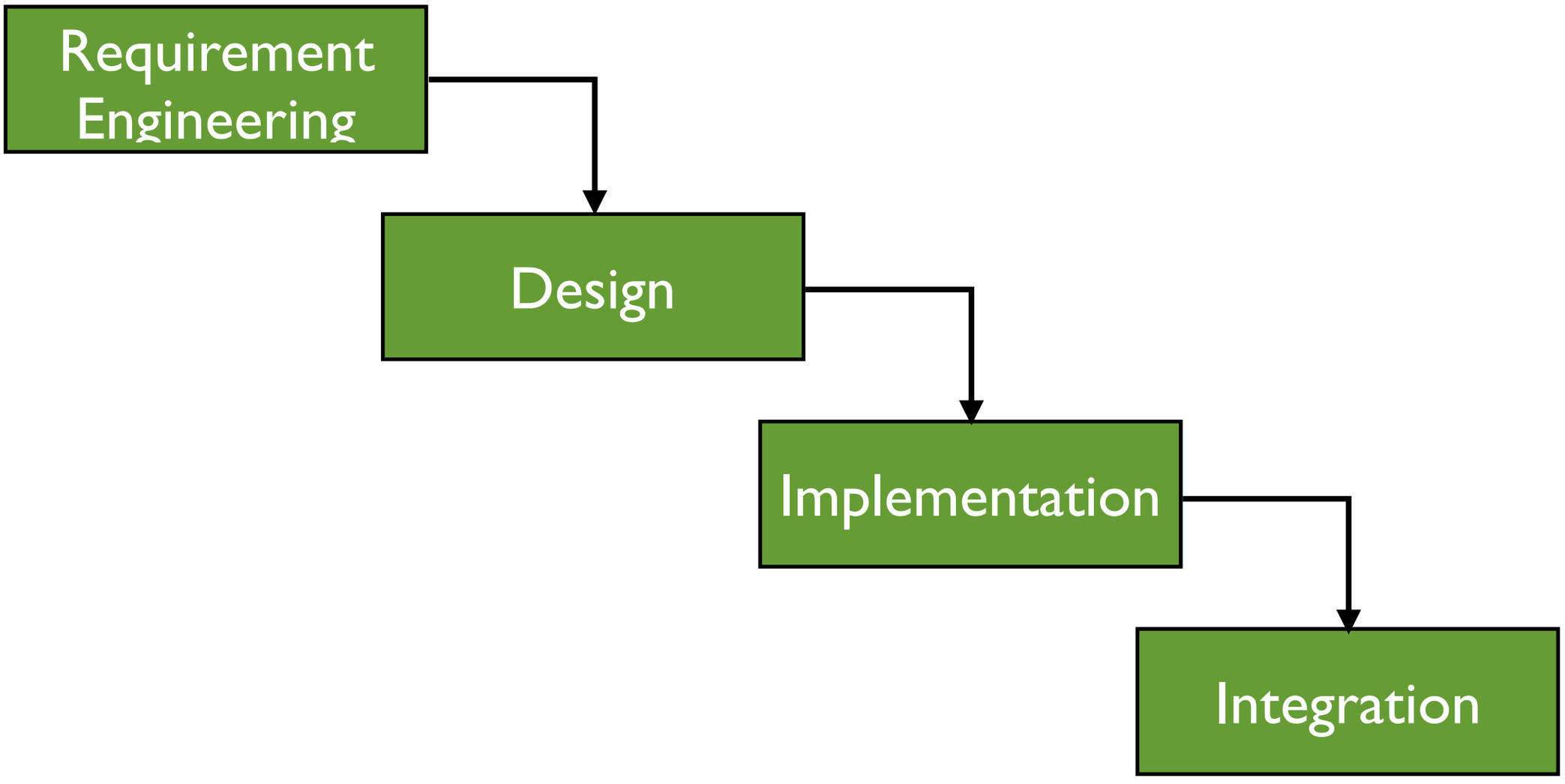
Software Process Models

Software Process Models

- A process model describes:
  - What steps you go through
  - Which development artifacts are produced, and when
  - How activities are coordinated

- Different process models
  - The waterfall model
  - The prototyping model
  - The iterative model
  - Agile development

The Waterfall Model



Requirement Engineering

- Figure out what the software is supposed to do…
- Collection
  - Talking to users, customers, etc. (customers != users)
  - Sometimes people are not sure about what they want
  - Some requirements can cost too much (but users/ customers do not know)

Requirement Engineering (Cont'd)

- Including functional & non-functional requirements
  - Functional requirements describe what a software system should do

    The correctness of the system
  - Non-functional requirements place constraints on how the system will do so

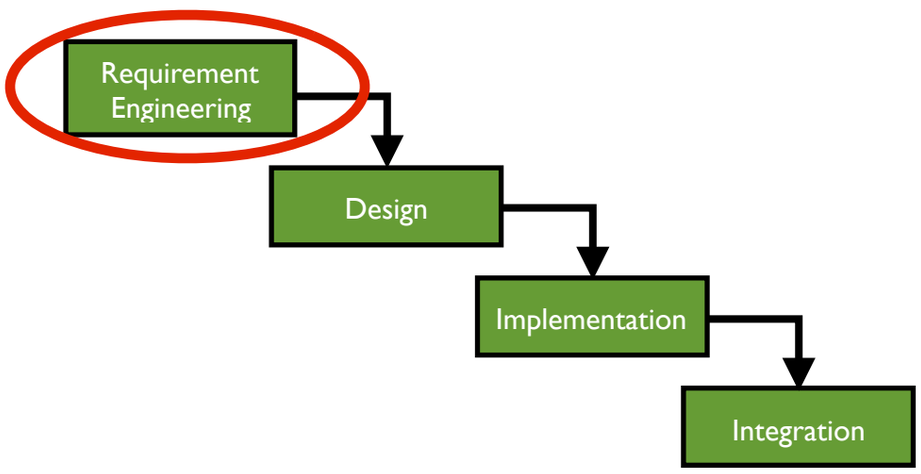
    Performance, usability, security, reliability, etc.

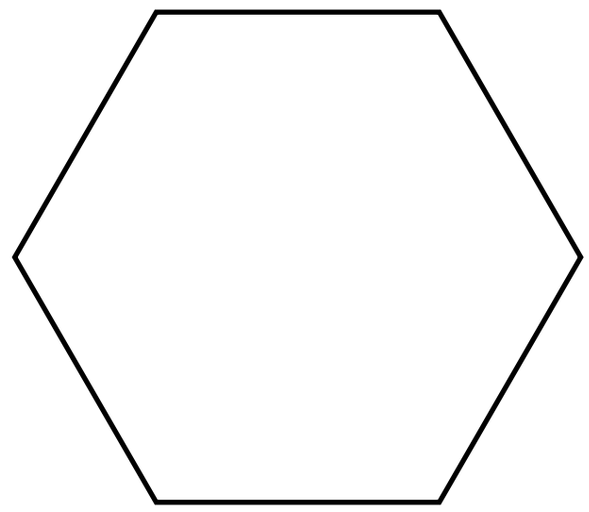
Functional requirement: The email must be sent when requested

Non-Functional requirement: The latency should not be more than one hour

Requirement Engineering

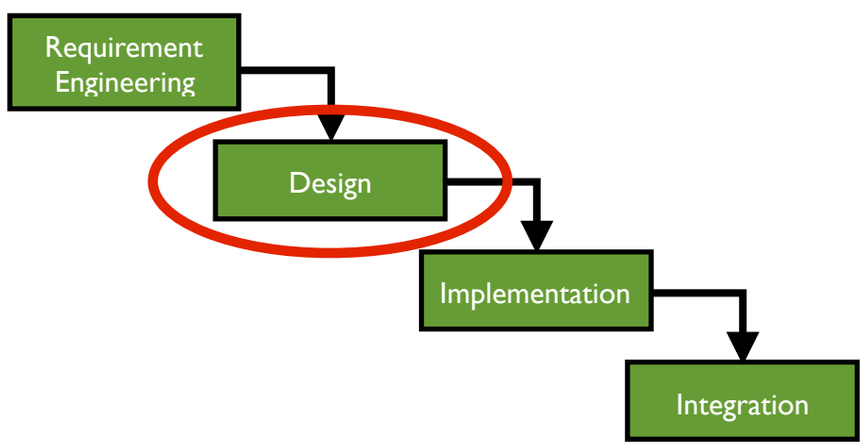
- Specification
  - A detailed document describes what the system does
  - Covers all situations
  - More precise than raw requirements
  - Can be formal or informal

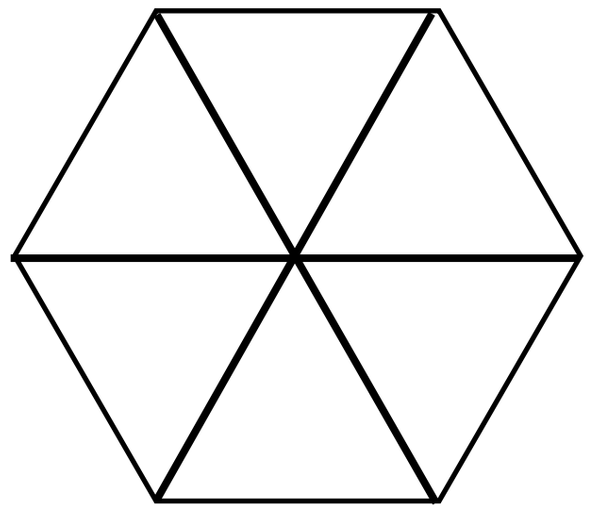




Design

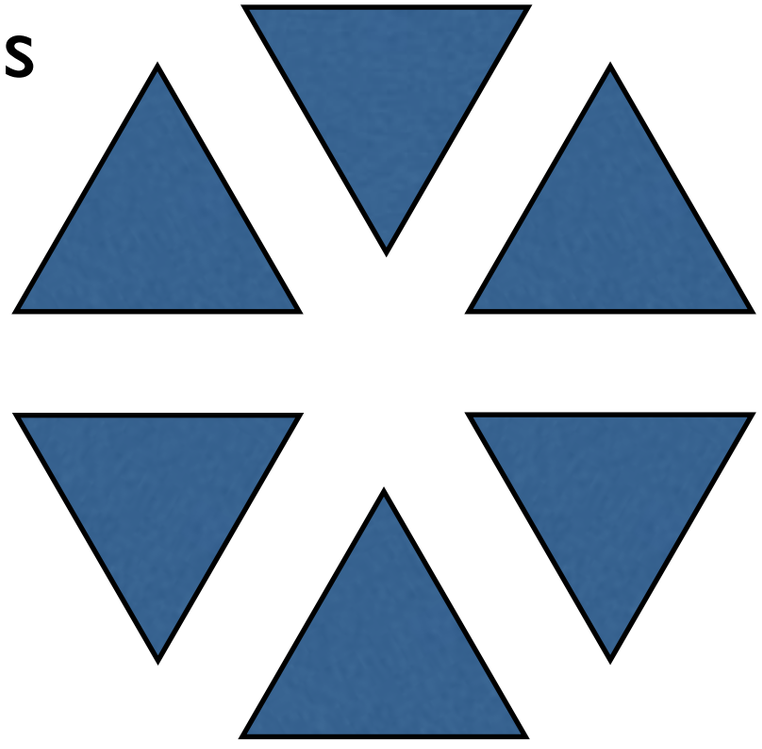
- The architecture
- How to decompose the software
- Define the interfaces between components

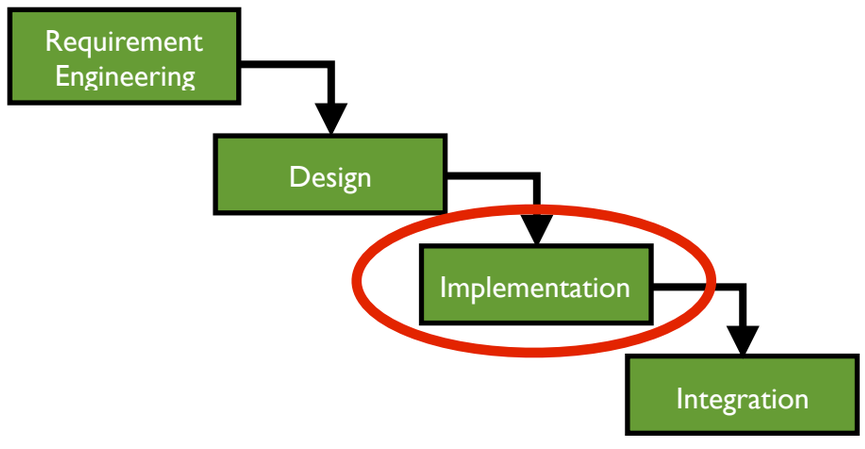




Implementation

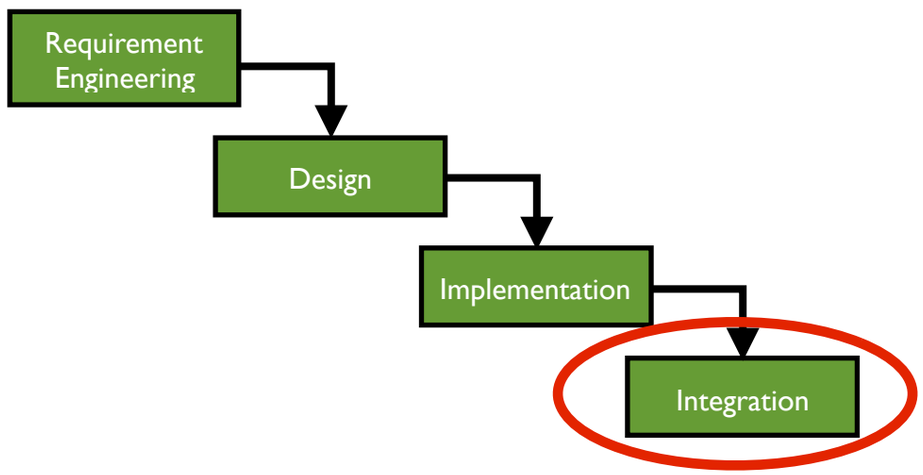
- Code each module
- Sequence of implementing modules
  - Priority
  - Testability / Dependence
- Unit Testing

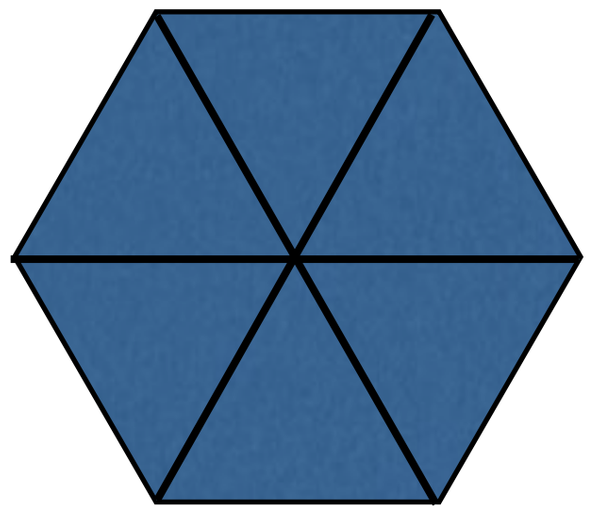




Integration

- Put things together
- Test the whole system



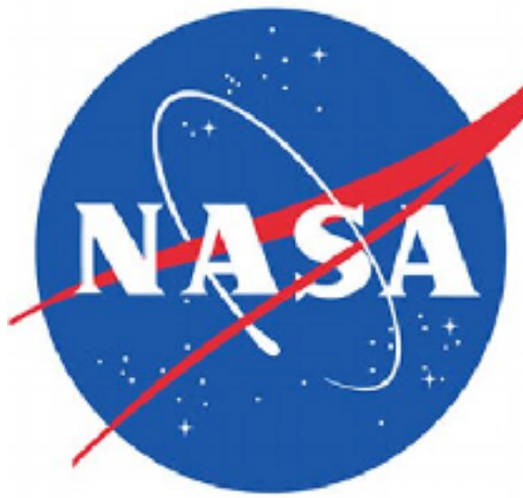


Waterfall Model

- A standard software process model
- Testing or validation after each step
  - move to a phase only when its preceding phase is reviewed and verified
- No iterations (some variants allow feedback between steps)

NASA Example

- Each execution handles $4Billion equipments, human lives, dreams
- 420k lines of code, 17 errors in 11 versions
- Commercial equivalent would have 5000 bugs

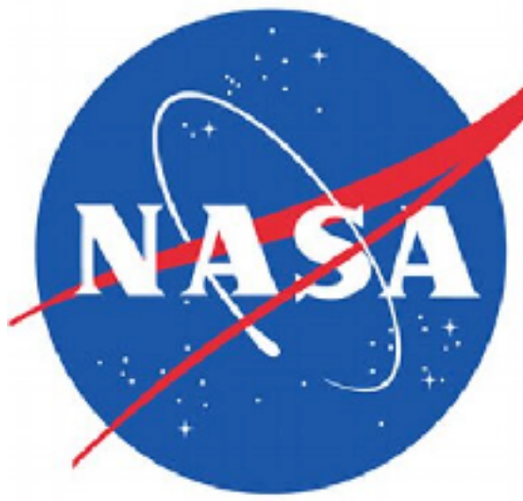


NASA Example (Cont'd)

- A third of the effort before coding starts
- Long specifications, written down, fully discussed
  - 40k pages of specification (longer than the code)
  - Change to add GPS support (2500 pages more specification)
  - Specification is almost pseudo code
- Validation and review at all levels
  - 85% of bugs revealed before testing

NASA Example (Cont'd)

- Cost
  - 260 people
  - $32 million
  - A year
- TOO EXPENSIVE!!!
- Overkill for normal software



In practice

- Very little software is built with waterfall
- What are the main risks?
- What are the good things about the waterfall model?

Risks of waterfall

- Relies on precise and stable requirements
- Users cannot involve much (specifications are difficult to understand)
- Takes too long to finish
- Small errors (or requirement changes) at the beginning steps are unaffordable

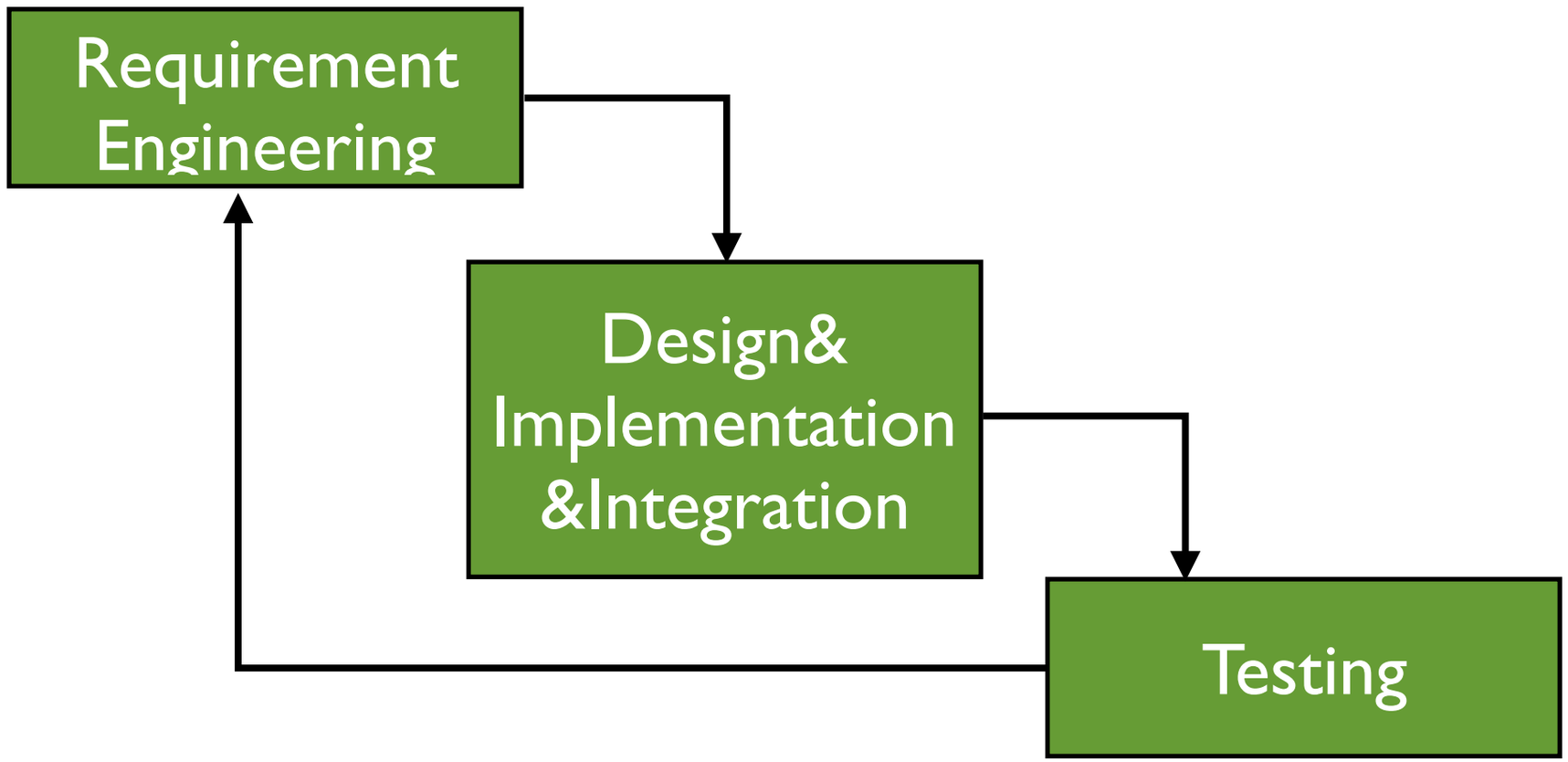
- Suitable: projects for specific critical software, no competition, enough resources

Good things of waterfall

- Defines all the basic activities for a software process
- Emphasis on documents and specifications to support high-quality software

The Iterative Model

- Solve the risks of waterfall by infinite weak cycles



The Iterative Model

- Plan for change
- Use the similar steps as waterfall
- But expect to iterate the whole process continually, and spend less effort on each iteration (weak cycles)

Example: Ant Project

- Building tool for Java (also works for C/C++)
- First stable version 1.1 released in June 2000
  - Small developer group with 3-4 people
  - First prototype version released in one month
  - First stable version released in about half a year

Example: Ant Project

- The Project evolves for 13 years
  - 5770 issue (bugs & new features) reports from users, 2581 reports handled
  - 7 major versions in 13 years
  - Code commits 12,000 +
  - File modifications 50,000 +
  - Lines of code from 25.6k to 260k

Advantages

- Users involve earlier
  - User can give feedback after the first version released
- Cheaper to get a working software
  - Get the first version very fast
- The software always work, though not perfect
  - Important in many cases, e.g., in competitions
- Keep refining requirements, and accommodates changes
  - Cost for requirement changes/errors are small

Disadvantages

- More bugs, sometimes may cause severe loss

420k lines of code, 17 errors

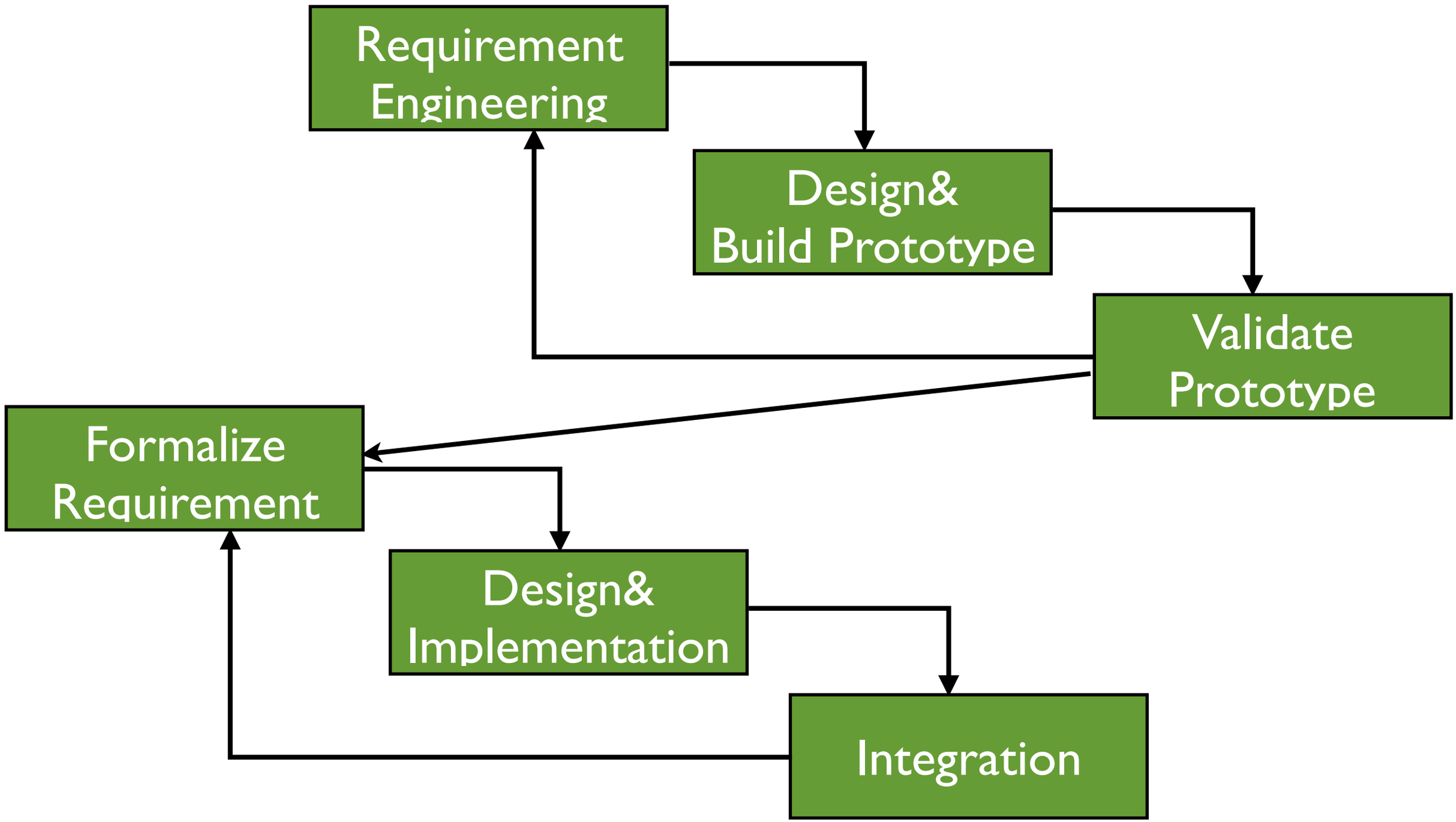
260k lines of code, 5770 bug reports

- Design is critical to ensure that a change does not affect the whole system
  - Ant has 12,000+ code commits

In Practice

- Most existing software projects use this model
  - Daily builds
  - Releases
- Not suitable for systems that are costly for testing or very critical in quality
  - NASA programs
  - Military / Scientific / …

The Prototype Model



The Prototype Model

- Looks like a two-cycle waterfall model
  - Actually not, it is weak + strong
- Different from waterfall
  - Good: users involve more (by using prototype), reveal small errors in requirements / design
  - Not solved: still can not handle frequent requirement changes
  - Bad: prototype is discarded (waste of some effort, sometimes can be even more expensive than waterfall)

When to use prototype model

- The desired system needs to have a lot of interaction with the end users
  - e.g., online systems and GUI applications

    Useful for requirement collection
- The desired system needs efforts to implement but its effectiveness is uncertain
  - e.g., novel research ideas

    Useful for quick evaluation

Agile Manifesto

- http://agilemanifesto.org

Agile Manifesto

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

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

Agile Manifesto

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Agile Manifesto

- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Agile Manifesto

- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.

Agile Manifesto

- 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.

From Manifesto to Process

- The Agile Manifesto does not provide concrete steps.
- Organizations usually seek more specific methods within the Agile movement
  - Extreme Programming
  - Feature Driven Development
  - Scrum, and others

Extreme Programming

- Extreme Programming (XP) is a (very) lightweight incremental software development process.
- It involves a high-degree of discipline from the development team
- Popularized by K. Beck (late 90's)
- Comprised of 12 core practices
- Most novel aspect of XP (as a process) is the use of pair programming

XP Core Practices

- Planning
- Small Releases
- System Metaphor
- Simple Design
- Continuous Testing
- Refactoring
- Collective Code Ownership
- Continuous Integration
- 40-hour Work Week
- On-site Customer
- Coding Standards
- Pair Programming

Planning

- Business (customers) and development (programmers) cooperate to produce the maximum business value as rapidly as possible.
  - Business lists desired features (user stories) for the system.
  - Development estimates how much effort each story will take.
  - Business then decides which stories to implement in what order, as well as when and how often to produce a production releases of the system.

Planning - User Stories

Title: Create Account
Description: I can create a named account

Title: List Accounts
Description: I can get a list of all accounts.

Title: Query Account Balance
Description: I can query account balance.

Title: Delete Account
Description: I can delete a named account

Small Releases

- Start with the smallest useful feature set.
- Release early and often, adding a few features each time.
- Each iteration ends in a release.

System Metaphor

- Each project has an organizing metaphor, which provides an easy to remember naming convention.
- The names should be derived from the vocabulary of the problem and solution domains.

Simple Design

- Just in time
  - Design and implement what you know now, not worry too much about future: future is unpredictable

- No optimization for future
  - It is common that the optimization becomes unnecessary

    Example: optimization to handle large input data, but later found the input changed (e.g., smaller format, or available in a summarized form)

Continuous Testing

- Always keeps code working
  - Test before/after any major changes (Continuous testing)
- Plan coding to allow frequent tests
  - Do not do too comprehensive changes, break to phases

  Example: Add a product query feature for shopping software
    - Add list all products first
    - Add text query
    - Add filtering conditions one by one
    - Add sorting
    - …

Refactoring

- Refactor out any duplicate code generated in a coding session.
- You can do this with confidence that you didn't break anything because you have the tests.

Collective Code Ownership

- No single person "owns" a module.
- Any developer is expect to be able to work on any part of the code base at any time.
- Improvement of existing code can happen at anytime by any pair
- Size???

Continuous Integration

- All changes are integrated into the code base at least daily.
- The tests have to run 100% both before and after integration.

40-Hour Work Week

- Programmers go home on time. In crunch mode, up to one week of overtime is allowed.
- Multiple consecutive weeks of overtime are treated as a sign that something is very wrong with the process.

On-site Customer

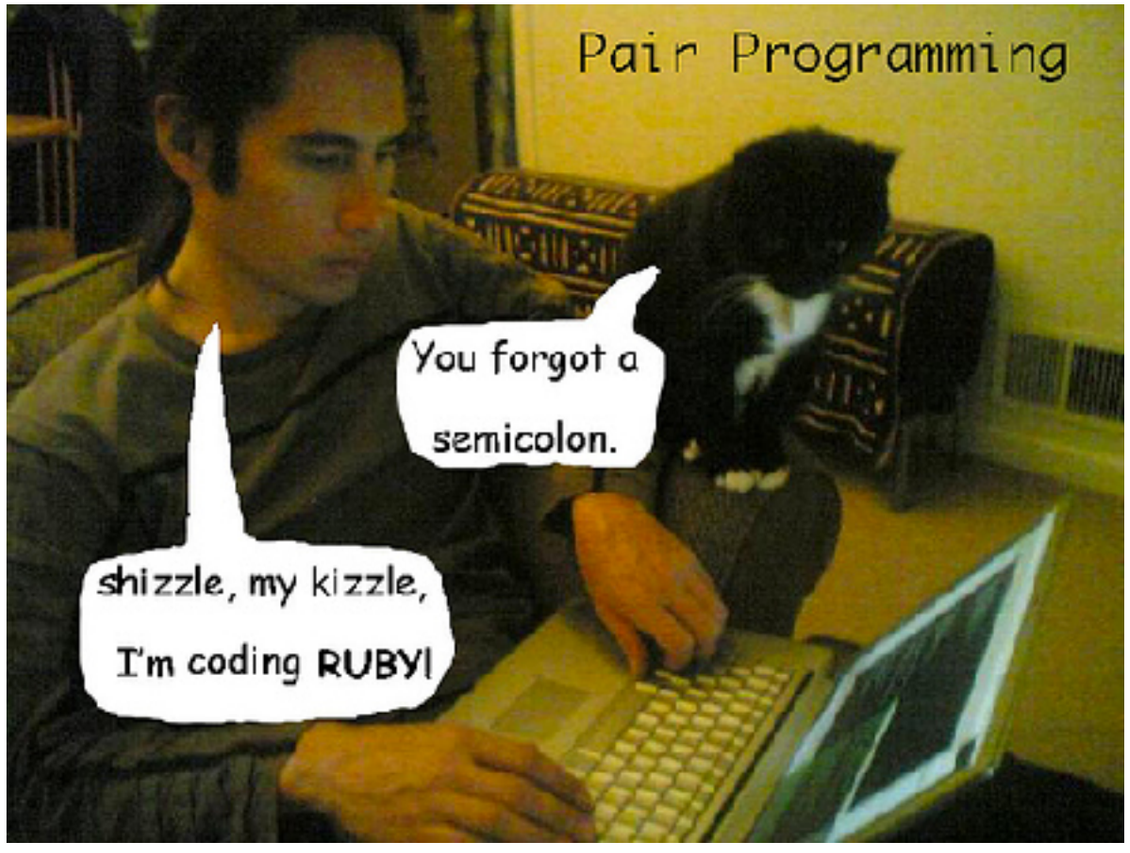
- Development team has continuous access to a real live customer, that is, someone who will actually be using the system.
- For commercial software with lots of customers, a customer proxy (usually the product manager) is used instead.

Coding Standards

- Everyone codes to the same standards.
- Ideally, you shouldn't be able to tell by looking at it who on the team has touched a specific piece of code.

Pair Programming

- Programmer and Monitor
  - Pilot and Copilot
  - Or Driver and Navigator
- Programmer types, monitor think about high-level issues
- Disagreement points to design decisions
- Pairs are shuffled



Pair Programming - Advantages

- Results in better code
  - Instant code review
  - Monitor always captures the big picture
- Results in better developer
  - More social atmosphere
  - Instant knowledge/idea exchange
  - More opportunities to learn new skills and better coding styles
- Reduces Risk
  - More people understand the code

Pair Programming - Why people don't like

- Stressful to relate to people all the time
- Slows the programming
  - But save for time in maintenance
- Waste of personnel
  - But fewer bugs, and more people can work on it, once bugs are revealed

Extreme Programming - Scalability

- XP works well with teams up to 12-15 developers.
- It tends to degrade with teams sizes past 20
- Work has been done in splitting large projects/ teams into smaller groups and applying XP within each group.

When to use extreme programming

- Requirement prone to change
- Easy to get testable requirements (often true in the maintenance phase on a software)
- Need quick delivery (business competition)

- In practice, frequently used in start-up companies

When **not** to use extreme programming

- Quality critical software (e.g., military, NASA projects)
- Large group for large project (still can be used for components)
- No highly-involved customers

Summary: Software Process Models

- Traditional models
  - Waterfall
  - Iterative
  - Prototyping
- Agile development
  - Extreme programming