CS189A - Capstone

Christopher Kruegel
Department of Computer Science
UC Santa Barbara
http://www.cs.ucsb.edu/~chris/

How Should We Build Software?

Let's look at an example

- Assume we asked our IT folks if they can do the following:
  - Every year all the PhD students in our department fill out a progress report that is evaluated by the graduate advisors. We want to make this online.

- After we told this to our IT manager, he said "OK, let's have a meeting so that you can explain us the functionality you want."

- We scheduled a meeting and at the meeting we went over
  - The questions that should be in the progress report
  - Type of answers for each question (is it a text field, a date, a number, etc?)
  - What type of users will access this system (students, faculty, staff)?
  - What will be the functionality available to each user?

Requirements Analysis and Specification

- Meeting where we discussed the functionality, input and output formats, types of users, etc. is called requirements analysis
  - during requirements analysis software developers try to figure out the functionality required by the client

- After the requirements analysis all these issues can be clarified in a **Software Requirements Specification (SRS)** document
  - maybe the IT folks who attended the requirements analysis meeting are not the ones who will develop the software, so the software developers will need a specification of what they are supposed to build.

- Writing precise requirements specifications can be challenging:
  - formal (mathematical) specifications are precise,
    but hard to read and write
  - English is easy to read and write, but ambiguous

Design

- After figuring out the requirements specifications, we have to build the software
- In our example, we assume that the IT folks are going to talk about the structure of this application first
  - there will be a backend database, the users will first login using an authorization module, etc.

Design

- Deciding on how to modularize the software is part of the **architectural design**
  - it is helpful (most of the time necessary, since one may be working in a team) to document the design architecture (i.e., modules and their interfaces) before starting the implementation

- After figuring out the modules, the next step is to figure out how to build those modules

- **Detailed design** involves writing a detailed description of the processing that will be done in each module before implementing it
  - generally written in some structured pseudo-code

Implementation and Testing

- Finally, the IT folks are going to pick an implementation language (PHP, Java Servlets, etc) and start writing code

- This is the **implementation** phase
  - implement the modules defined by the architectural design and the detailed design.

- After the implementation is finished the IT folks will need to check if the software does what it is supposed to do

- Use a set of inputs to **test** the program
  - when are they done with testing?
  - can they test parts of the program in isolation?

Maintenance

- After they finished the implementation, tested it, fixed all the bugs, are they done?

- No, we (client) may say, “I would like to add a new question to the PhD progress report” or “I found a bug when I was using it” or “You know, it would be nice if we can also do the MS progress reports online” etc.
  - The difficulty of changing the program may depend on how we designed and implemented it

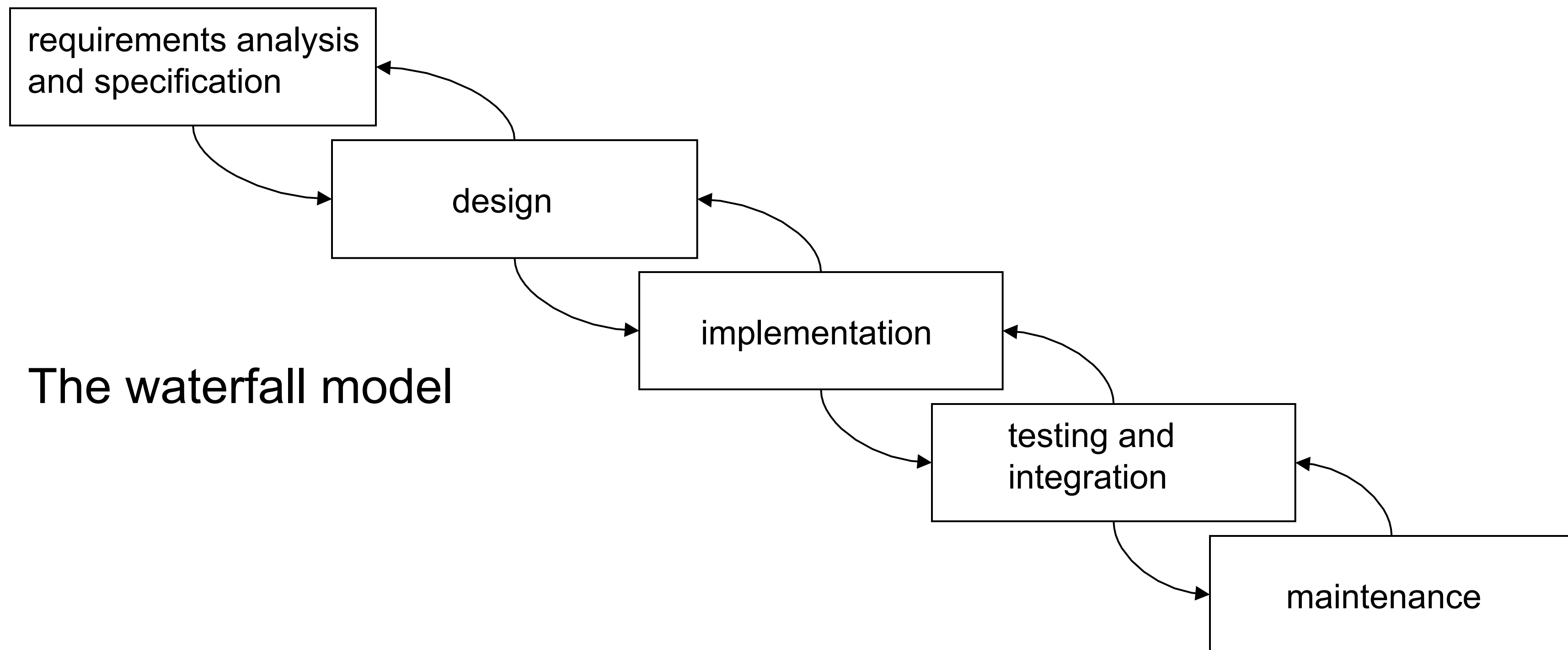
- This is called the **maintenance** phase where the software is continually modified to adopt to the changing needs of the customer and the environment.

Software Process Models

- Software process (software life-cycle) models
  - Determine the stages (and their order) involved in software development and evolution
  - Establish the transition criteria for progressing from one stage to the next

- Software process models answer the questions:
  - What shall we do next?
  - How long shall we continue to do it?

Waterfall Model



Software product is not only the executable file:
source code, test data, user manual, requirements specification, design specification

Waterfall Model

- Waterfall model is *document-driven*
- Documents
  - requirements specification, design specification, test-plan
  - these documents are crucial in achieving maintainability, traceability and visibility
- Feedback loops between different stages are confined to successive stages to minimize the expensive rework involved in feedback across many stages

Waterfall Model

Problems with waterfall model

- Because of the restricted feedback loops, waterfall model is essentially sequential
  - for example, the requirements have to be stated completely before the implementation starts.
  - it is often difficult for the customer to state all requirements explicitly
  - it is hard to handle changes in the requirements
- A working model of the software is not available until late in the project life-span
  - an undetected mistake can be very costly to fix
  - the delivered program may not meet the customer's needs
- For interactive, end-user applications document-driven approach may not be suitable.
  - for example, it is hard to document a GUI

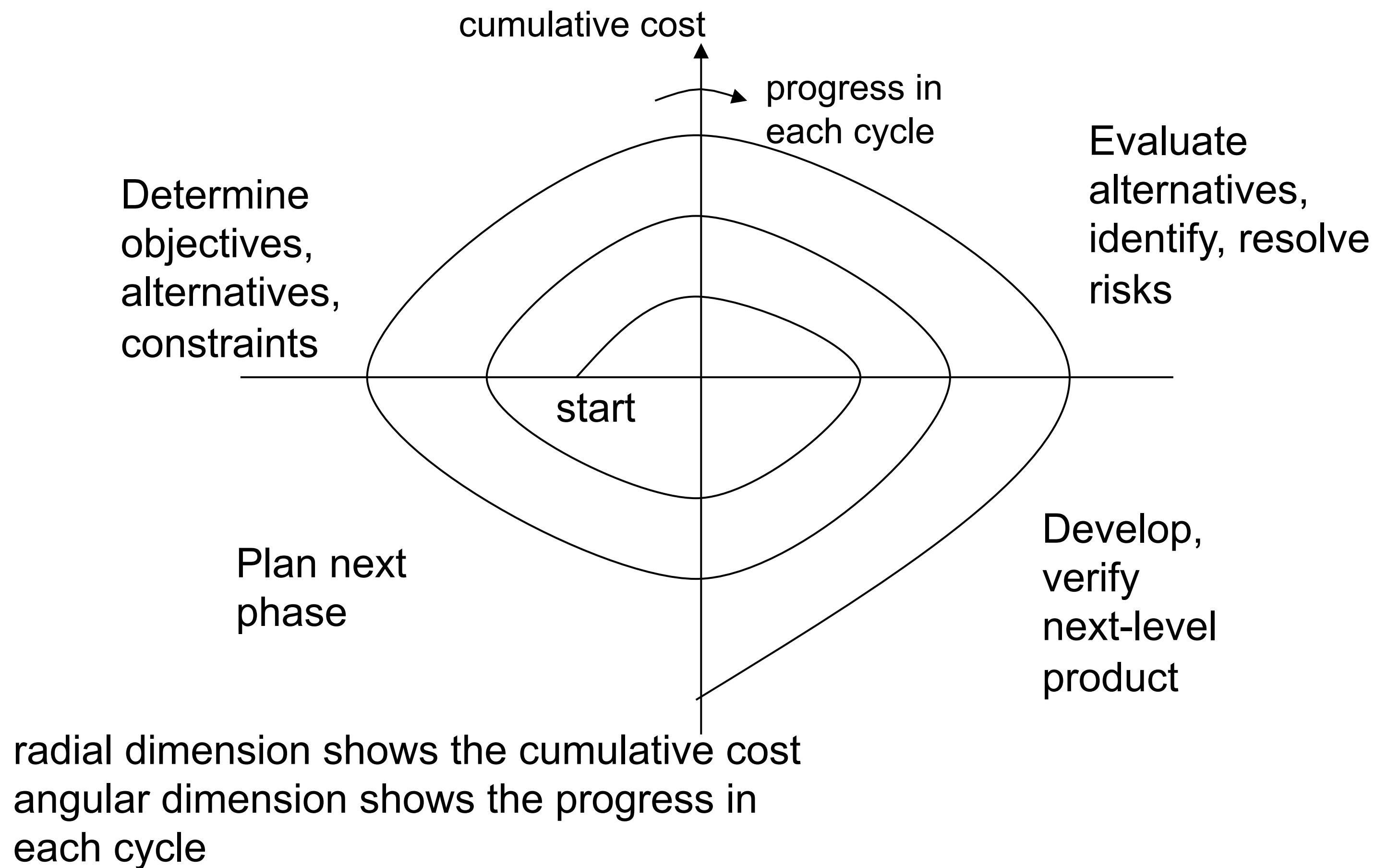
Rapid Prototyping

- After an initial requirements analysis, a quick design is developed
- This quick design should focus on aspects of the software that will be visible to the user such as input/output formats
- The quick design is used to construct a prototype
- The prototype is reviewed by the customer and/or user to refine the requirements for the software to be developed
- Prototype serves as a mechanism for identifying software requirements
- Especially useful for interactive applications

Rapid Prototyping

- Dangers of prototyping
  - The quick and possibly poor choices made in the design and the implementation of the prototype may influence the real product
  - After seeing a prototype customer may demand a working product fast
  - The prototype becomes the requirements specification. Since requirements specification serves as a contract between customer and developer, a prototype may not be a good contract

Spiral Model



radial dimension shows the cumulative cost
angular dimension shows the progress in each cycle

Spiral Model

- Spiral model consists of iterative cycles
- Spiral model can be considered a generalization of other process models
- Each cycle consists of four steps:

Step 1

- Identify the objectives (for example: performance, functionality, ability to accommodate change)
- Identify the alternative means of implementing this portion of the product (for example: different designs, reuse, buy)
- Identify the constraints imposed on the application of the alternatives (for example: cost, schedule)

Spiral Model

Step 2

- Evaluate the alternatives relative to objectives and constraints.
- Evaluate the risks involved with each alternative
- Resolve the risks using prototyping, simulation, benchmarking, requirements analysis, etc.
  - alternative: write a requirements specification
  - risk: customer may not be able to articulate the requirements precisely which may end up a costly redevelopment effort
  - risk resolution: develop a rapid prototype

Spiral Model

Step 3

- Develop and verify the product
- Product could be the software requirements specification, the design specification, etc.

Step 4

- Plan the next phase
- Depending on the next-phase this could be a requirements plan, an integration and test plan, etc.

Spiral Model

- The basic idea in spiral model is to evaluate and resolve risks at every step of the development

- Based on the risks involved in the development spiral model may become equivalent to other models (or a mixture of them)
  - If a project has a low risk in user-interface and performance requirements and has a high risk in budget and schedule predictability and control, based on these risks spiral model may turn into the waterfall model

- The challenges in using the spiral model
  - relies on risk assessment expertise
  - steps of the software life-cycle are not clearly defined

How Microsoft Builds Software

M. A. Cusumano and R. W. Selby, Communications of the ACM, 1997

- Microsoft 1996 numbers
  - big company
    - 20,500 employees (2010: 80-90K)
    - 250 products
    - $8.7 billion in revenues (2010: $62 billon)
  - big products
    - Windows 95: more than 11 million lines of code, a development team of 200 programmers and testers
    - Windows 7: 50+ million lines, 25 feature teams, 40 developers each

How Microsoft Builds Software

M. A. Cusumano and R. W. Selby, Communications of the ACM, 1997

- Microsoft philosophy for product development: to cultivate its roots as a highly flexible entrepreneurial company
  - do not adopt too many of the structured software-engineering practices
  - scale up the loosely structured (hacker) style of development

Challenges for Microsoft

Large complex software products

- Cannot be built by 2-3 person teams
- Team members need to create components that are interdependent
- Sequential life-cycle models such as waterfall model does not work very well
  - It is hard to define components accurately in the early stages of the development cycle
  - Requirements evolve during the development process

Microsoft's Approach

Microsoft's approach:

- Small parallel teams (three to eight developers each) or individual programmers
- Individual programmers and teams have freedom to evolve their designs and operate nearly autonomously
- Teams synchronize their changes frequently to make sure that the product components all work together

Synch-and-Stabilize Approach

- Continually **synchronize** what people are doing as individuals and as members of parallel teams and periodically **stabilize** the product in increments
  - milestone, daily build, nightly build, zero-defect
- **Build:** putting together partially completed pieces of a software product during the development process to see what functions work and what problems exist, usually by completely recompiling the source code and executing automated regression tests

Synch-and-Stabilize Approach

- Three phases
  - **Planning Phase:** Define product vision, specification and schedule
  - **Development Phase:** Feature development in 3 or 4 sequential subprojects that each results in a milestone release
  - **Stabilization Phase:** Comprehensive internal and external testing after each milestone, final product stabilization and ship

Planning Phase

- Planning Phase
  - **Vision Statement:** product managers (marketing specialists) and program managers (who specialize in writing functional specifications) use extensive customer input to identify priority-order product features
  - **Specification Document:** Based on the vision statement, program management and development group define feature functionality, architectural issues, and component interdependencies
  - **Schedule and Feature Team Formation:** Based on the specification document, program management coordinates schedule and arranges feature teams that each contain approximately 1 program manager, 3-8 developers and 3-8 testers (testers work in parallel, 1:1 with developers)

Experience shows that initial feature set may change up to 30%

Development Phase

- Development Phase
  - Program managers coordinate the evolution of the specification. Developers design, code, and debug. Testers pair with developers for continuous testing
  - *Milestone 1: First 1/3 of features* (Most critical features and shared components
  - *Milestone 2: Second 1/3 of features*
  - *Milestone 3: Final 1/3 of features* (Least critical features)

All the feature teams go through a complete cycle of development, feature integration, testing and fixing problems in each milestone

Throughout the project the feature teams synchronize their work by building the product and by finding and fixing errors on a daily and weekly basis

At the end of a milestone the developers fix almost all the errors detected and stabilize the product

Development Phase: Milestones

Milestone 1 (first 1/3 features)

- –Development (design, coding, prototyping)
- –Usability lab
- –Private release testing
- –Daily builds
- –Feature debugging
- –Feature integration
- –Code stabilization
- –Buffer time (20-50%)

Milestone 2 (next 1/3 of the features)

- –Development
- –Usability lab
- –Private release testing
- –Daily builds
- –Feature Debugging
- –Feature Integration
- –Code stabilization
- –Buffer time

Milestone 3 (last set)

- –Development
- –Usability lab
- –Private release testing
- –Daily builds
- –Feature Debugging
- –Feature Integration
- –Feature Complete
- –Code Complete
- –Code stabilization
- –Buffer time
- –Zero-defect release
- –Release to Manufacturing

Stabilization Phase

- Stabilization Phase
  - Program managers coordinate with OEMs (Original Equipment Manufacturers) and ISVs (Independent Software Vendors) and monitor customer feedback. Developers perform final debugging and code stabilization. Testers recreate and isolate errors
  - Internal testing: Thorough testing of complete product
  - External testing: Thorough testing of complete product outside of company by “beta” sites such as OEMs or ISVs and end users
  - Release preparation: Prepare final release of “golden master” disk and documentation for manufacturing

Synch-and-Stabilize Approach: Requirements

- Vision statement and feature specifications to guide projects
  - leaves developers and program managers room to innovate or adapt to changed or unforeseen competitive opportunities and threats
- Base feature selection and priority order on user activities and data
  - Particularly for end-user applications there is a need for continuous observation and testing by real users during development

Synch-and-Stabilize Approach: Project Management

- Divide large projects into multiple milestone cycles with buffer time (20-50% of total project time) and no separate product maintenance group
  - buffer times give people time to respond to changes and unexpected difficulties and delays
- Control by individual commitments to small tasks and fixed project resources
  - Managers allow team members to set their own schedules but only after the developers have analyzed tasks in detail (for example half-day to three-day chunks). Team members are asked to personally commit to the schedules they set.
  - Managers then fix project resources by limiting the number of people they allocate to each project, they also try to limit the time spent on projects (teams can sometimes delete features if they fall too far behind)

Synch-and-Stabilize Approach: Design

- Evolve a modular design architecture mirroring the product structure in the project structure
  - Modular architecture allows teams to incrementally add or combine features

Synch-and-Stabilize Approach: Implementation

- Work in parallel teams but “synch up” and debug daily
  - Developers can check-out private copies of source code files from a centralized master version of the source code.
  - Developers implement their features by changing the private copies of the source code files.
  - Developers create private build of the product containing the new feature and test it
  - Then they check-in their private copy of the source code to the master version. Check-in process includes an automated regression test to make sure that there are no new errors.
  - Developers typically check-in their code twice a week

- Always have a product you can ship
  - versions for every major platform and market
  - designated developer (project build master) generates a complete build of the product on a daily basis

Synch-and-Stabilize Approach: Testing/Debugging

- Speak a "common language" on a single development site
  - Nearly all teams work on the same physical site with common development languages (C, C++) common coding styles, and standardized development tools
- Continuously test the product as you build it
  - Product teams test the product as they build them including usability tests on users
- Metric data to determine milestone completion and product release
  - By monitoring metrics such as how many bugs are newly opened, resolved, fixed, and active, the managers decide when to move forward in the project

Synch-and-Stabilize Approach: Summary

- Summary of Microsoft's approach
  - Small parallel teams (three to eight developers each) which have freedom to evolve their designs and operate nearly autonomously
  - Daily synchronization through product builds, periodic milestone stabilization, and continuous testing
- Coordinates teams in a flexible manner
- Suitable for fast changing demands

Sync-and-Stabilize vs. Waterfall

Sync-and-Stabilize Process Model
(Evolutionary Development)

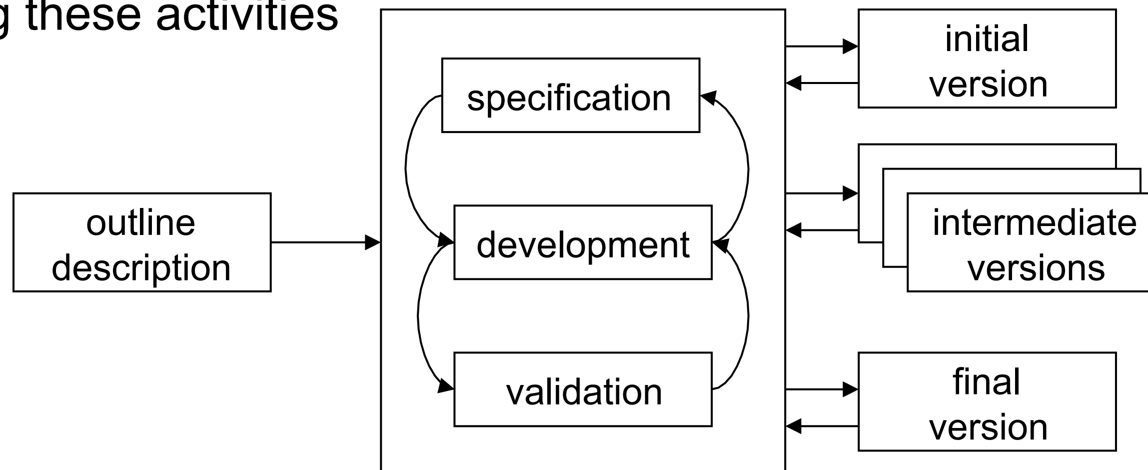
- Product development and testing done in parallel
- Vision statement and evolving specification
- Features are prioritized and built in 3 or 4 milestone subprojects
- Frequent synchronizations (daily builds) and intermediate stabilizations (milestones)
- "Fixed" release and ship dates and multiple release cycles
- Customer feedback continuous in the development process
- Product and process is designed so that large teams work like small teams

Waterfall Process Model
(Sequential Development)

- Separate phases done in sequence
- Complete "frozen" specification and detailed design before building the product
- Trying to build all pieces of a product simultaneously
- One late and large integration and system test phase at the project's end
- Aiming for feature and product "perfection" in each project cycle
- Feedback primarily after development as inputs for future projects
- Working primarily as a large group of individuals in a separate functional department

Evolutionary Software Development

- Software is built iteratively and incrementally by first providing an initial version and then improving/extending it based on the user feedback until an adequate system has been developed
  - Scrum, extreme programming, agile software development
- As opposed to the sequential nature of the waterfall model, in the evolutionary software development specification, development and validation activities are executed concurrently with fast feedback among these activities



Agile Software Development

Manifesto for Agile Software Development

available at: http://agilemanifesto.org/

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

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"

Principles of Agile Software Development

- 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.
- 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.

Principles of Agile Software Development

- 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.
- 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

- Extreme programming (XP) is a type of agile software development process proposed by Kent Beck
  - "Embracing Change with Extreme Programming," by Kent Beck, IEEE Computer, October 1999, pp. 70-77

Extreme Programming

- XP follows the agile software development principles as follows
  - Software is built ***iteratively***, with ***frequent releases***
  - Each release implements the set of ***most valuable features/use-cases/stories*** that are chosen by the customer
  - Each release is implemented in a ***series of iterations***, each iteration adds more features/use-cases/stories
  - Programmers turn the stories into ***smaller-grained tasks***, which they individually accept responsibility for
  - The programmer turns a task into a set of ***test cases*** that will demonstrate that the task is finished
  - Working as ***pairs***, the programmers make the test cases run, evolving the design in the meantime to maintain the simplest possible design for the system as a whole

Extreme Programming Practices

- **Planning:** Customers decide the scope and timing of releases based on the estimates provided by programmers. Programmers implement only the functionality demanded by the features/use-cases/stories in that iteration.
- **Small Releases:** The system is put into production in a few months, before solving the whole problem. New releases are made often—anywhere from daily to monthly
- **Simple Design:** At every moment, the design runs all the tests, communicates everything the programmers want to communicate, contains no duplicate code, and has the fewest possible classes and methods.

Extreme Programming Practices

- **Tests:** Programmers write unit tests minute by minute. These tests are collected and they must all run correctly. Customers write functional tests for the features/use-cases/stories in an iteration.
- **Refactoring:** The design of the system is evolved through transformations of the existing design that keep all the tests running.
- **Pair programming:** All production code is written by pairs of programmers, each pair uses a single computer.
- **Continuous integration:** New code is integrated with the current system after no more than a few hours.

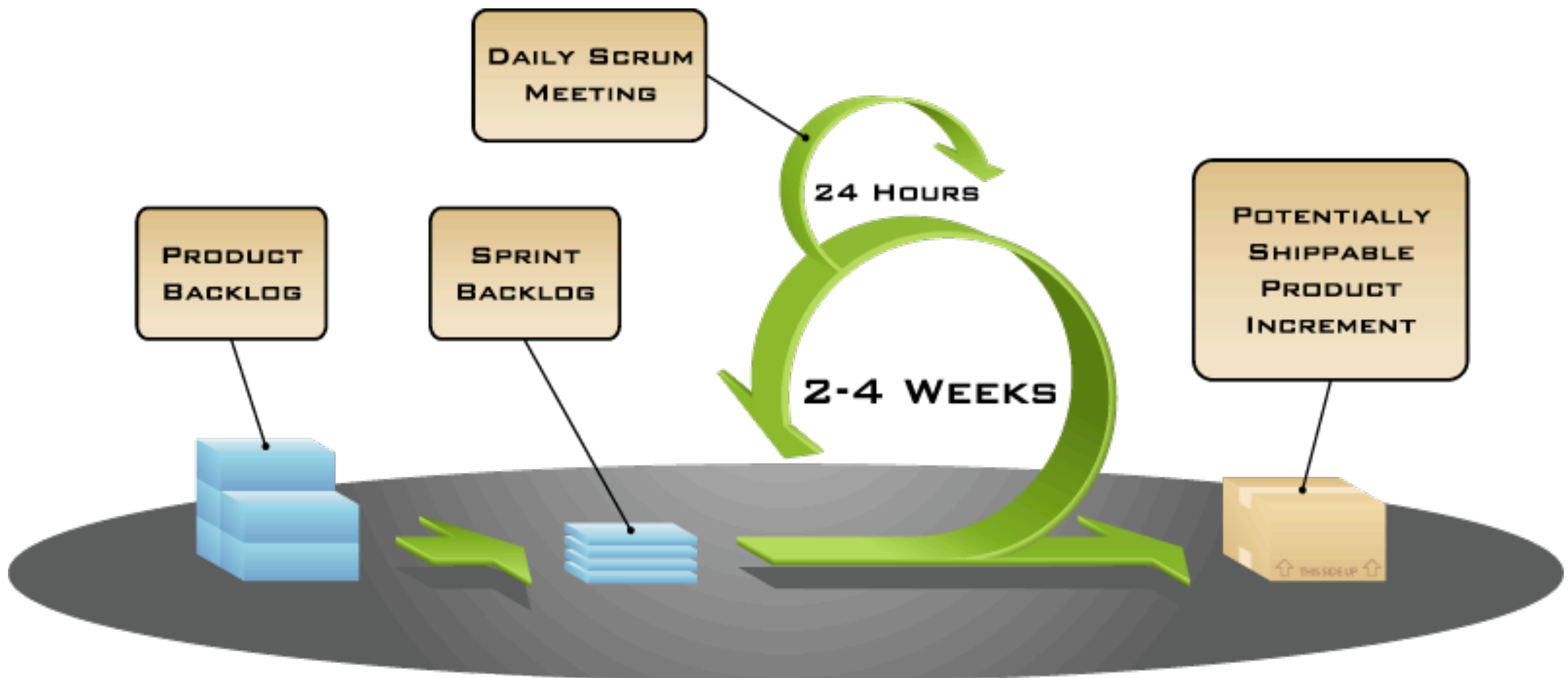
Extreme Programming Practices

- **Collective ownership:** Every programmer improves any code anywhere in the system at any time if they see the opportunity.
- **On-site customer:** A customer sits with the team full-time.
- **Open workspace:** The team works in a large room with small cubicles around the periphery. Pair programmers work on computers set up in the center.

Scrum

- An evolutionary/iterative/incremental/agile software process
- The main roles in Scrum are:
  - Scrum team: Team of software developers
  - Scrum master : Project manager
  - Product owner: Client
- Characteristics of Scrum:
  - Self-organizing teams
  - Product development in two to four week sprints
  - Requirements are captures as items in a list of product backlog

Scrum Overview



Scrum Roles

- Product owner
  - Defines the features of the product
  - Decides on release date and content
  - Prioritize features according to market value
  - Adjust features and priority every iteration as needed
  - Accepts or rejects work results

- Scrum Master
  - Represents management of the project
  - Responsible for following the Scrum process
  - Ensures that the team is fully functional and productive
  - Shields the team from external influences

Scrum Roles

- Scrum Team
  - Typically 5 to 9 people
  - Cross-functional team that does the software development including designing, programming and testing
  - Co-location and verbal communication among team members
  - Teams are self-organizing, no titles
  - Team membership should not change during a sprint

Scrum Meetings

- **Sprint Planning** (at most 8 hours)
  - This is done at the beginning of every sprint cycle (2 to 4 weeks)
  - Team selects items from the product backlog they can commit to completing
  - Sprint backlog is created
    - Tasks for this sprint are identified and each is estimated (1 to 16 hours). This is done collaboratively, not by ScrumMaster
  - High-level design is discussed

- **Daily Scrum** (at most 15 minutes)
  - Daily, stand-up meeting
  - Not for problem solving
  - Every team member answers three questions:
    - What did you do yesterday?
    - What will you do today?
    - Is anything in your way? (ScrumMaster is responsible for following up and resolving the impediments)

Scrum Meetings

- **Sprint Review** (at most 4 hours)
  - Team presents what it accomplished during the sprint
    - Typically a demo of new features or underlying architecture
    - Incomplete work should not be demonstrated
  - Informal meeting, no slides
  - Whole team participates
  - Open to everybody

Scrum Meetings

- **Sprint Retrospective** (at most 3 hours)
  - Periodically take a look at what is and is not working
  - Done after every sprint
  - ScrumMaster, Product owner, Team and possibly customers and others can participate
  - One way of doing sprint retrospective is to ask everyone what they would like to
    - 1) Start doing, 2) Stop doing, 3) Continue doing

Scrum Artifacts

- **Product Backlog**
  - These are the requirements
  - A list of all desired work on the project
  - Prioritized by the product owner
    - Reprioritized at the start of each sprint
  - Each backlog item also has an estimated time it will take to complete it

Scrum Artifacts

- **Sprint Backlog**
  - Team members sign up for work of their own choosing
  - Estimated work remaining is updated daily
  - Any team member can add, delete or change the sprint backlog
  - Each sprint backlog item has daily estimates for the amount of time that will be spent on that item each day

- **Burn Down Chart**
  - A daily updated chart displaying the remaining cumulative work on the sprint backlog. It gives a simple view of the sprint progress.

More on Scrum

- More information about Scrum process is available at:
  - www.mountaingoatsoftware.com/scrum
  - www.scrumalliance.org
  - www.controlchaos.com