Dates
- Final exam
  - Monday, 12/13 8:00 - 9:50 (in this room)
  - cumulative
- Summary of Apache / Mozilla case study
  - due Wednesday 12/1 in class
- Test cases / beta test results
  - due Friday, 12/3 noon
- Acceptance Testing Code deadline
  - Monday, 12/6, 11am
  - bring your own laptop (one per group)
    - describe design
    - demonstrate functionality

Quality terminology
What are bugs?
- Error: human mistake
- Fault: result of mistake, evidenced in some development or maintenance product
- Failure: departure from the system’s required behavior
Garvin’s perspectives on quality

- Transcendental view: something we recognize but can’t define
- User view: fitness for purpose
- Manufacturing view: conformance to specification
- Product view: tied to inherent product characteristics
- Value-based view: depends on customer’s willingness to pay

McCall’s Quality Model

External quality criteria:
- Correctness
- Reliability
- Efficiency
- Integrity
- Usability
- Maintainability
- Testability
- Portability
- Reusability
- Interoperability

Systems approach to SE

- Identify activities and objects.
- Define the system boundary.
- Consider nested systems, system interrelationships.
Key factors altering software engineering practice (Wasserman)
- criticality of time-to-market for commercial products
- shifts in economics of computing (lower HW, higher development/maintenance costs)
- availability of powerful desktop computing
- extensive local- and wide-area networking
- availability and adoption of OO technology
- graphical user interfaces
- unpredictability of waterfall model of development

Wasserman’s basis for good software engineering
- Abstraction
  - Analysis and design methods and notations
  - User interface prototyping
  - Software architecture
  - Software process
- Reuse
- Measurement
- Tools and integrated environments

Capturing the requirements
- Requirement: a feature of the system or a description of something the system is capable of doing in order to fulfill the system’s purpose
- Three kinds of requirements:
  - those that absolutely must be met
  - those that are highly desirable but not necessary
  - those that are possible but could be eliminated
Requirements documents

- **Requirements definition**: complete listing of what the customer expects the system to do
- **Requirements specification**: restates the definition in technical terms so that the designer can start on the design
- **Configuration management**: supports direct correspondence between the two documents

Configuration management

- Set of procedures that track
  - requirements that define what the system should do
  - design modules that are generated from requirements
  - program code that implements the design
  - tests that verify the functionality of the system
  - documents that describe the system

Functional vs. non-functional requirements

- **Functional**: describes an interaction between the system and its environment.
  - Examples:
    - System shall communicate with external system X.
    - What conditions must be met for a message to be sent
- **Non-functional**: describes a restriction or constraint that limits our choices for constructing a solution
  - Examples:
    - Paychecks distributed no more than 4 hours after initial data are read.
    - System limits access to senior managers.
Types of requirements

- Physical environment
- Interfaces
- Users and human factors
- Functionality
- Documentation
- Data
- Resources
- Security
- Quality assurance

Characteristics of requirements

- Are they correct?
- Are they consistent?
- Are they complete?
- Are they realistic?
- Does each describe something the customer needs?
- Are they verifiable?
- Are they traceable?

Requirements Notations

- Dynamic vs. Static
- Examples?
Milestones and activities

- Activity: takes place over a period of time
- Milestone: completion of an activity -- a particular point in time
- Precursor: event or set of events that must occur in order for an activity to start
- Duration: length of time needed to complete an activity
- Due date: date by which an activity must be completed

Slack or float time

Slack time = available time - real time

= latest start time - earliest start time

Critical Path

- a path from start to finish in the activity graph for which the slack at every node is 0
  - typically multiple critical paths (not just one)
- Any delay of a node on a critical path delays the whole project
  - in contrast, nodes with slack can tolerate delays
- Analyzing your project in this way is called the Critical Path Method (CPM)
Effort estimation

- Expert judgment
  - analogy
  - proportion
  - Delphi technique
  - Wolverton model
- Algorithmic methods: \( E = (a + bS^c) \times m(X) \)
  - \( S \) size, \( X \) cost vector, \( m \) multiplier, \( a, b, c \) constants
  - Walston and Felix model: \( E = 5.25S^{0.91} \)
  - Bailey and Basili model: \( E = 5.5 + 0.73S^{1.16} \)

COCOMO model: stages of development

- application composition:
  - prototyping to resolve high-risk user interface issues
  - size estimates in object points
- early design:
  - to explore alternative architectures and concepts
  - size estimates in function points
- postarchitecture:
  - development has begun
  - size estimates in lines of code

Risk management requirements

- Risk impact: the loss associated with the event
- Risk probability: the likelihood that the event will occur
- Risk control: the degree to which we can change the outcome

Risk exposure = (risk probability) x (risk impact)
Three strategies for risk reduction

- avoiding the risk: change requirements for performance or functionality
- transferring the risk: transfer to other system, or buy insurance
- assuming the risk: accept and control it

Risk leverage = difference in risk exposure divided by cost of reducing the risk

Boehm’s top ten risk items

- Personnel shortfalls
- Unrealistic schedules and budgets
- Developing the wrong functions
- Developing the wrong user interfaces
- Gold-plating
- Continuing stream of requirements changes
- Shortfalls in externally-performed tasks
- Shortfalls in externally-furnished components
- Real-time performance shortfalls
- Straining computer science capabilities

Project plan contents

- project scope
- project schedule
- project team organization
- technical description of system
- project standards and procedures
- quality assurance plan
- configuration management plan
- documentation plan
- data management plan
- resource management plan
- test plan
- training plan
- security plan
- risk management plan
- maintenance plan
Conceptual design

- Tells the customer what the system will do
- Answers:
  - Where will the data come from?
  - What will happen to the data in the system?
  - What will the system look like to users?
  - What choices will be offered to users?
  - What is the timing of events?
  - What will the reports and screens look like?
- Characteristics of good conceptual design
  - in customer language with no technical jargon
  - describes system functions
  - independent of implementation
  - linked to requirements

Technical design

- Tells the programmers what the system will do
- Includes:
  - major hardware components and their function
  - hierarchy and function of software components
  - data structures
  - data flow

Five ways to create designs

- Modular decomposition
- Data-oriented decomposition
- Event-oriented decomposition
- Outside-in design
- Object-oriented design
Three design levels

- Architecture: associates system components with capabilities
- Code design: specifies algorithms and data structures for each component
- Executable design: lowest level of design, including memory allocation, data formats, bit patterns

Design styles

- Pipes and filters
- Object-oriented design
- Implicit invocation
- Layering
- Repositories
- Interpreters
- Process control
- Client-server

Characteristics of good design

- Component independence
  - coupling
    - degree of dependence among components
  - cohesion
    - degree of relatedness of internal parts of a component
- Exception identification and handling
- Fault prevention and tolerance
  - active
  - passive
**Coupling**
- highly coupled system = lot of dependence between components
  - via invocation
  - via data transfer
  - via control (eg control codes / flags)

**Content Coupling**
- one component modifies another
  - eg one component modifies internal data of another
  - very undesirable
  - can be turned into **common coupling**
    - use common data store
    - coupling still there but at least centralized

**Control Coupling**
- some parameter passed controls execution of a component
- can restrict coupling of each component executes only one function
  - localizes control to well-defined interface
**Stamp & Data Coupling**

- Data structure is passed from component to component to = **stamp coupling**
- Only individual data are passed = **data coupling**

---

**Types of Cohesion**

- **Coincidental**
  - parts are really unrelated and just got put together ("utils")
  - should be avoided
- **Logical**
  - logically related parts together (eg all file i/o)
  - no functional relation so still undesirable

---

**Types of Cohesion (2)**

- **Temporal**
  - functions only related by time sequence, eg. initialization
  - like logical cohesion type difficult to change
- **Procedural**
  - functions grouped because they have to be performed in certain order
- **Communicational**
  - data fetched at same time (eg from same file / disk)
Types of Cohesion (3)

- Sequential
  - output of one component used as input in another
  - better than others but still unconstrained in many ways
- Functional
  - every part is essential to a specific function and all essential elements are contained in one component
  - performs the function it is designed and no others

Patterns & Frameworks

- Patterns support reuse of software architecture and design
  - patterns capture the static and dynamic structures and collaborations of successful solutions to problems that arise when building applications in a particular domain
- Frameworks support reuse of detailed design and code
  - a framework is an integrated set of components that collaborate to provide a reusable architecture for a family of related applications
  - Both improve development time and reuse

Design Patterns

- Represent solutions to problems that arise when developing software in a particular context
  - i.e., pattern = problem / solution pair in context
- Capture static & dynamic structure and collaboration among key participants in software design
  - particular useful in articulating how and why to resolve non-functional forces
- Facilitate reuse of successful software architectures and design
Design Pattern Descriptions
- Name and intent
- Problem and context
- Force(s) addressed
- Abstract description of structure and collaborations
- Positive and negative consequences of use
- Implementation guidelines and sample code
- Known uses and related patterns
- Pattern descriptions are often independent of programming language or implementation details
- contrast with frameworks

Frameworks
- are semi-complete applications
  - complete applications are developed by inheriting from and instantiating parameterized frameworks
  - provide domain-specific functionality
    - e.g., business or telecommunication applications, window systems etc.
  - exhibit inversion of control at run time
    - = framework determines which objects and methods to invoke in response to events

Class Libraries / Frameworks & Patterns
- Class libraries
  - self-contained, plugable ADTs
- Frameworks
  - reusable, semi-complete applications
- Patterns
  - problem, solution, context
Design Pattern Space

- Creational Patterns
  - initializing and configuring classes and objects
- Structural patterns
  - decoupling interface and implementation of classes and objects
- Behavioral patterns
  - dynamic interaction among groups of classes and objects

Creational Patterns

- Factory method
  - method in a derived class creates associates
- Abstract factory
  - factory for building related objects
- Builder
  - factory for building complex objects incrementally
- Prototype
  - factory for cloning new instances from prototype
- Singleton
  - factory for a singleton (sole) instance

Structural Patterns

- Adapter
  - translator adapts a server interface for a client
- Bridge
  - abstraction for binding one of many implementations
- Composite
  - structure for building recursive aggregations
- Decorator
  - extends an object transparently
- Facade
  - simplifies the interface for a subsystem
- Flyweight
  - many, fine-grained objects shared efficiently
- Proxy
  - one object approximates another (intermediary)
Behavioral Patterns (1)

- Chain of responsibility
  - request delegated to designated service provider
- Command
  - request as first class object
- Interpreter
  - language interpreter for small grammar
- Iterator
  - aggregate elements are accessed sequentially

Behavioral Patterns (2)

- Mediator
  - mediator coordinates actions between its associates
- Memento
  - snapshot captures and restores object states privately
- Observer
  - dependents update automatically when subject changes
- State
  - object whose behavior depends on its state

Behavioral Patterns (3)

- Strategy
  - abstraction for selecting one of many algorithms
- Template method
  - algorithm with some step supplied by a derived class
- Visitor
  - operations adapted to elements of a heterogeneous object structure
Writing high-quality code

- Good code must be based on good design
  - choose right data & control structures
  - choose appropriate programming language(s)
  - program into a language not in it
    - use your own conventions, standards & class libraries if language does not provide what you need
  - most programming principles are independent of the specific language but depend on how you use it

Good Construction Practices

- Start with a good design
  - decide how much (little!) to do at keyboard
  - set coding conventions for names, comments, layout
  - define how to handle errors, security, class interface standards, reuse standards, performance
  - program into not in

OO: Designing Classes

- Foundations: ADTs
- Interfaces
- Implementation issues
- When to create a class?
- Language dependency
- Packages
<table>
<thead>
<tr>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>hide implementation details</td>
</tr>
<tr>
<td>changes will be localized</td>
</tr>
<tr>
<td>interface can be more informative (eg not currentFont.size = 16 but use an ADT size in pixels or points)</td>
</tr>
<tr>
<td>easier to improve performance</td>
</tr>
<tr>
<td>correctness is more obvious</td>
</tr>
<tr>
<td>more self-documenting</td>
</tr>
<tr>
<td>less coupling (no need to pass data everywhere)</td>
</tr>
<tr>
<td>deal with high-level (real world) entities not low implementation detail</td>
</tr>
</tbody>
</table>

CS 1530: Software Engineering Fall 2004