* Reliability (different from safety)
  + Correctness: Does what it is supposed to do, and only that
  + Robustness: Acts acceptably when it can’t do what it is supposed to do
* Safety
  + An absence of accidents (an unplanned and unacceptable loss)
  + Reliable but unsafe: mars polar lander
  + Safe but unreliable: human operators (procedure deviation)
* Contracts
  + Bertram Meyer said that “Defining a precondition and a postcondition for a routine is a way to define a contract that binds the routine and its callers…”
  + Meyer’s contracts specify
    - Obligations required of the client
    - Conditions that must be true before the service will be requested
    - Obligations required of the service provider
    - Conditions that must be true during and after the execution of the service
    - Guarantees of services
    - Defined for each method or service call
  + For one of our milestone we had to do a method contract here is an example of one
  + Method Contract (Chapter 4):
    - **Operation Signature**:
    - + push( x : object) : bool   where push is a public instance method of Stack class and x is any instance or subclass of Object.  method returns true if successful, false if x not added to stack
    - **Preconditions**: true
    - **Postconditions**:  size’ = size + 1.  Original contents unchanged, x is new top of stack.
    - **Invariant:** size >=0 (Invariant is what is guaranteed true)
    - **Frame Condition:**size is changed, stack is changed, original stack elements not changed
    - It is important to remember that contracts are the EXTERNAL view of the object.  Within a method, we may temporarily violate something, but upon return, everything has to be consistent.
  + Exceptions vs Errors
    - Exceptions are deviations from the norm that we expect
      * Recoverable: The system can find an alternative work around and user can continue their original task (such as FileNotFound, Format Problems, and Checked vs Unchecked)
      * UnRecoverable: The system continues running, but user cannot complete original task (such as IOexception)
    - Strategies for dealing with Exceptions/Errors
      * Rollback - try to perform, but on failure undo everything
      * Retry - repeatedly attempt, after recovering from failed attempts
      * Appeal to higher authority - allow user to steer application
      * Resign - minimize damage, write to log, signal definite and safe failure
    - Errors are when things go unexpectedly wrong and little can be done to recover

REPAIR: R – Runtime : (do nothing/let it die)  
E – Exception (catch and handle an exception at throw point)  
P – Propagation (allow exceptions to move up system to point where you will handle them)  
A – Atomic (transactions – ensure something either happens or it doesn’t)  
I – Isolation (keep other processes from seeing changes.  i.e. until transaction commits, nothing has happened)  
R – Resumption (retry, code tries to “heal” itself)

**GRASP and Object Design**

* GRASP stands for guidelines for assigning responsibility to classes and objects inobject-oriented design**.**
  + Controller: assigns responsibility of dealing with system events to a non-UI class that represents the overall system or use case scenario. THIS IS A NON-USER INTERFACE OBJECT THAT HANDLES A SYSTEM EVENT
  + Creator: classes handling creation of objects.
    - Creating classes are responsible for:
      * recording instances of the new objects
      * Closely using instances of the new objects
      * Have initialized info for instances of new objects and pass it on creation
  + High Cohesion: Evaluative Pattern that tries to keep objects focused, manageable and understandable. Used to help with low coupling. Responsibilities of a certain element are extremely related and focused. To increase cohesion, break programs into different systems and classes. Low cohesion means there are too many unrelated responsibilities. This means it is hard to understand. Objects should not do many unrelated things.
  + Indirection: Supports low coupling and reuse potential between two elements. Ex. model-view-controller pattern to mediate between data and representation
  + Information Expert: Determine where to delegate responsibilities (methods, computed fields, etc). Look at a responsibility, determine what is needed to make it work, and where it needs to be stored.
  + Low Coupling: Measures how strongly an element is connected to another element (knowledge/reliance of).
    - Have low dependency between classes
    - Changing something in one class, does not change much in another class
    - High reuse potential
  + Polymorphism: responsibility of defining the variation of behaviors based on type is assigned to the type for which this variation happens. use polymorphic operations instead of explicit branching based on type.
  + Protected Variations: protects elements from variation of other elements. Done so  by wrapping focus of instability with an interface and use polymorphism to create different implementations of the interface
  + Pure Fabrication: A class that does not represent a concept of problem domain. Made for low coupling, high cohesion and reuse. AKA Domain-Driven Design
* SOLID stands for single responsibility, open-closed, Liskov substitution, interface segregation, and dependency inversion
  + The idea is that if you apply the SOLID principles your system will be easier to maintain and extend over time.

**User Evaluations**

* Techniques for Evaluation
  + Formative vs Summative
  + With/Without end-users
  + Using a variety
* Specifically look at Heuristic evaluation, Cognitive Walkthrough, Think Aloud Observation
* Look at Usability NOT FUNCTIONALITY
* Heuristic Eval
  + Small set of evaluators look at interface and judge compliance
  + Formative & Qualitative
  + Can be used with Written Prototypes
  + Created by Jakob Nielsen
* Eval Procedures
  + Gather Input
  + Independent Evaluations
  + Debrief: Consolidate Found Errors
  + Severity Ratings: based off frequency, impact, persistence and market impact (0-4 rating scale)
* Usability Bug
  + Anything that evaluator thinks is. JUST IDENTIFY, DO NOT SOLVE
* Cognitive Walkthrough
  + Evaluate design for ease of learning
  + Needs detailed description of prototype
  + Define Inputs (user goals and interaction tasks)
  + Walk through Action Sequence (address step of task sequence)
  + Record good info
* Observing Users
  + Best way to gather feedback
  + Watch, listen and learn
  + Think aloud
  + Identifying errors are hard to catch

Design

* Singleton
  + Problem: Limit the application to only one instance of a particular class, but with global access
  + Used to control access to key resources
  + Solution: make static accessor method, private constructor
  + Problems: anyone can get a reference, subclassing, default public constructors, introduces coupling, limits reuse