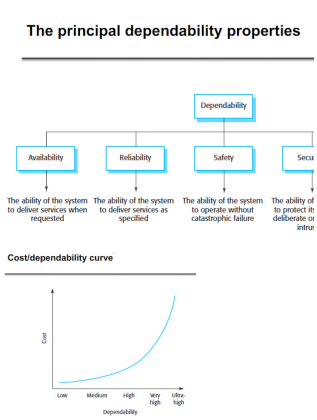
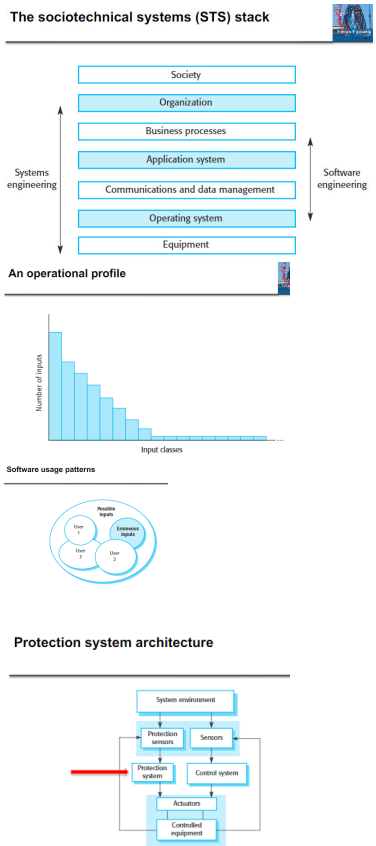
```
public static void main(String[] args) {
        ClassPool pool = ClassPool.getDefault();
        boolean useRuntimeClass = true;
        if (useRuntimeClass) {
            ClassClassPath classPath = new
ClassClassPath(new Rectangle().getClass());
            pool.insertClassPath(classPath);
        } else {
            String strClassPath = workDir +
"\\bin";
            pool.insertClassPath(strClassPath);
        }
        CtClass cc = pool.get("target.Rectangle");

cc.setSuperclass(pool.get("Target.Point"));//takes
a CtClass
        cc.writeFile(outputDir);//outputDir is a
string

         ClassPool pool = ClassPool.getDefault();
         boolean useRuntimeClass = true;
         if (useRuntimeClass) {
            ClassClassPath classPath = new
ClassClassPath(new Rectangle().getClass());
            pool.insertClassPath(classPath);
         } else {
            String strClassPath = workDir +
"\\bin";
            pool.insertClassPath(strClassPath);
         CtClass cc = pool.get("target.Rectangle");

curClass.setSuperclass(pool.getSuperClass());
        cc.writeFile(outputDir);

        ClassPool pool = ClassPool.getDefault();
        CtClass cc = pool.makeClass(newClassName);
        cc.writeFile(outputDir);
        CtClass ccInterface =
pool.makeInterface(newInterfaceName);
        ccInterface.writeFile(outputDir);

        ClassPool pool = ClassPool.getDefault();
      String strClassPath = outputDir;
      pool.insertClassPath(strClassPath);
        CtClass ccPoint2 =
pool.makeClass("Point2");
        ccPoint2.writeFile(outputDir);
        CtClass ccRectangle2 =
pool.makeClass("Rectangle2");
        ccRectangle2.writeFile(outputDir);
        // ccRectangle2.defrost(); //
modifications of the class definition will be
permitted.
        ccRectangle2.setSuperclass(ccPoint2);
        ccRectangle2.writeFile(outputDir);

       CtMethod m = cc.getDeclaredMethod("say");
        m.insertBefore("{
System.out.println(\"Hello.say:\"); }");
        Class<?> c = cc.toClass();
        Hello h = (Hello) c.newInstance();
        h.say();

   private static String       workDir       =
System.getProperty("user.dir");
   private static final String TARGET_POINT    =
"target.Point";
   private static final String TARGET_RECTANGLE =
"target.Rectangle";
        ClassPool cp = ClassPool.getDefault();
      String strClassPath = workDir +
File.separator + "classfiles";
      pool.insertClassPath(strClassPath);
        Loader cl = new Loader(cp);
        CtClass cc = cp.get(TARGET_RECTANGLE);
        cc.setSuperclass(cp.get(TARGET_POINT));
        Class<?> c =
cl.loadClass(TARGET_RECTANGLE);
        Object rect = c.newInstance();
        System.out.println("[DBG] rect object: " +
rect);
```

```
        Class<?> rectClass = rect.getClass();
        Method m =
rectClass.getDeclaredMethod("getVal", new Class[]
{});
        System.out.println("[DBG] method: " + m);
        System.out.println("[DBG] result: " +
m.invoke(rect, new Object[] {}));

   public static void main(String[] args) throws
Throwable {
      SampleLoader s = new SampleLoader();
      Class<?> c = s.loadClass("MyApp");
      c.getDeclaredMethod("main", new Class[] {
String[].class }).invoke(null, new Object[] { args
});
   }
   private ClassPool pool;
   public SampleLoader() throws NotFoundException {
      pool = new ClassPool();
      pool.insertClassPath(inputDir); //
MyApp.class must be there.
   }
public static void main(String[] args) throws
Throwable {
      SubstituteMethodBody s = new
SubstituteMethodBody();
      Class<?> c = s.loadClass(TARGET_MY_APP);
      Method mainMethod =
c.getDeclaredMethod("main", new Class[] {
String[].class });
         mainMethod.invoke(null, new Object[] { args
});
   }

   protected Class<?> findClass(String name) throws
ClassNotFoundException {
      CtClass cc = null;
      try {
         cc = pool.get(name);
         cc.instrument(new ExprEditor() {
            public void edit(MethodCall m) throws
CannotCompileException {
               String className = m.getClassName();
               String methodName =
m.getMethodName();

               if (className.equals(TARGET_MY_APP)
&& methodName.equals(DRAW_METHOD)) {
                  System.out.println("[Edited by
ClassLoader] method name: " + methodName + ", line:
" + m.getLineNumber());
                  m.replace("{"//
                        + "$proceed($$); "//
                        + "}");
               } else if
(className.equals(TARGET_MY_APP) &&
methodName.equals(MOVE_METHOD)) {
                  System.out.println("[Edited by
ClassLoader] method name: " + methodName + ", line:
" + m.getLineNumber());
                  m.replace("{" //
                        + "$1 = 0; " //
                        + "$proceed($$); " //
                        + "}");
               }
            }
         });
         byte[] b = cc.toBytecode();
         return defineClass(name, b, 0, b.length);

static String workDir   =
System.getProperty("user.dir");
ClassPool pool = ClassPool.getDefault();
        pool.insertClassPath(inputDir);
        CtClass cc = pool.get("target.Point");
        CtMethod m = cc.getDeclaredMethod("move");
        m.insertBefore("{
System.out.println(\"[DBG] param1: \" + $1); " + //
                "System.out.println(\"[DBG] param2:
\" + $2); }");
```

```
        cc.writeFile(outputDir);
        System.out.println("[DBG] write output to:
" + outputDir);

        ClassPool defaultPool =
ClassPool.getDefault();
        defaultPool.insertClassPath(INPUT_PATH);
        CtClass cc = defaultPool.get(TARGET_MYAPP);
        CtMethod m =
cc.getDeclaredMethod(FACT_METHOD);
        m.useCflow(FACT_METHOD);
        m.insertBefore("if ($cflow(fact) == 0)" +
System.lineSeparator() + //
                "System.out.println(\"[MyAppFact
Inserted] fact \" + $1);");
        cc.writeFile(OUTPUT_PATH);
        InsertMethodBodyCflow s = new
InsertMethodBodyCflow();//pool = new
ClassPool();pool.insertClassPath(OUTPUT_PATH); //
TARGET must be there.
        Class<?> c = s.loadClass(TARGET_MYAPP);
        Method mainMethod =
c.getDeclaredMethod("main", new Class[] {
String[].class });
        mainMethod.invoke(null, new Object[] { args
});
        //findClass method:cc =
pool.get(name);byte[] b = cc.toBytecode();return
defineClass(name, b, 0, b.length);

        SubstituteMethodBody s = new
SubstituteMethodBody();// pool = new
ClassPool();pool.insertClassPath(new
ClassClassPath(new
java.lang.Object().getClass()));pool.insertClassPat
h(INPUT_PATH); // "target" must be there.
        Class<?> c = s.loadClass(TARGET_MY_APP);
        Method mainMethod =
c.getDeclaredMethod("main", new Class[] {
String[].class });
        mainMethod.invoke(null, new Object[] { args
});
        cc = pool.get(name);
        cc.instrument(new ExprEditor() {
            public void edit(MethodCall m) throws
CannotCompileException {
                if (
            byte[] b = cc.toBytecode();
            return defineClass(name, b, 0, b.length);

            FieldAcess s = new FieldAcess(); //pool
= new ClassPool();pool.insertClassPath(new
ClassClassPath(new
java.lang.Object().getClass()));pool.insertClassPat
h(INPUT_PATH); // TARGET must be there.
            Class<?> c =
s.loadClass(TARGET_MY_APP);
            Method mainMethod =
c.getDeclaredMethod("main", new Class[] {
String[].class });
            mainMethod.invoke(null, new Object[] {
args });

            NewExprAccess s = new NewExprAccess();
            Class<?> c =
s.loadClass(TARGET_MY_APP2);
            Method mainMethod =
c.getDeclaredMethod("main", new Class[] {
String[].class });
        mainMethod.invoke(null, new Object[] { args
});
            cc = pool.get(name);
         cc.instrument(new ExprEditor() {
               public void edit(NewExpr newExpr)
throws CannotCompileException {
                  StringBuilder code = new
StringBuilder();
                  code.append("\"y: \" + " +
"$_.y);\n }\n");
                  // System.out.println(code);
                  newExpr.replace(code.toString());
```

```
        String src = "public void xmove(int dx) { x
+= dx; }";
        CtMethod newMethod = CtNewMethod.make(src,
cc);
        cc.addMethod(newMethod);
        cc.writeFile(outputDir);

        CtMethod newMethod = CtNewMethod.make(src,
cc, "this", "move");
        cc.addMethod(newMethod);
        cc.writeFile(outputDir);

        ClassPool pool = ClassPool.getDefault();
        pool.insertClassPath(inputDir);
        CtMethod newMethod = new
CtMethod(CtClass.intType, "move", new CtClass[] {
CtClass.intType }, cc);
        cc.addMethod(newMethod);
        newMethod.setBody("{ x += $1; return x;}");
        cc.setModifiers(cc.getModifiers() &
~Modifier.ABSTRACT); cc.writeFile(outputDir);

ClassPool pool = ClassPool.getDefault();
        pool.insertClassPath(inputDir);
        CtClass cc = pool.get("target.Point");
        String src = "public void xmove(int dx) {
x += dx; }";
        CtMethod newMethod = CtNewMethod.make(src,
cc);
        cc.addMethod(newMethod);
        cc.writeFile(outputDir);

        ClassPool pool = ClassPool.getDefault();
        pool.insertClassPath(inputDir);
        CtClass cc = pool.get("target.Point");
        String src = "public void ymove(int dy) {
$proceed(0, dy); }";
        CtMethod newMethod = CtNewMethod.make(src,
cc, "this", "move");
        cc.addMethod(newMethod);
        cc.writeFile(outputDir);
        System.out.println("[DBG] write output to:
" + outputDir);

        ClassPool pool = ClassPool.getDefault();
        pool.insertClassPath(inputDir);
        CtClass cc = pool.get("target.Point");
        CtMethod newMethod = new
CtMethod(CtClass.intType, "move", new CtClass[] {
CtClass.intType }, cc);
        cc.addMethod(newMethod);
        newMethod.setBody("{ x += $1; return
x;}");
        cc.setModifiers(cc.getModifiers() &
~Modifier.ABSTRACT);
        cc.writeFile(outputDir);

        ClassPool pool = ClassPool.getDefault();
        pool.insertClassPath(inputDir);
        CtClass cc = pool.get("target.Point");
        CtMethod m = CtNewMethod.make("public
abstract int m(int i);", cc);
        CtMethod n = CtNewMethod.make("public
abstract int n(int i);", cc);
        cc.addMethod(m);
        cc.addMethod(n);
        m.setBody("{ return ($1 <= 0) ? 1 : (n($1
- 1) * $1); }");
        n.setBody("{ return m($1); }");
        cc.setModifiers(cc.getModifiers() &
~Modifier.ABSTRACT);

CtField f = new CtField(CtClass.intType, "z",
pointClass);
        pointClass.addField(f);
```





abstract class Base { — int f = 0; public int m1() { return 0; } } class ClassA extends Base { public int m1() { return f; } } class ClassB extends Base { public int m1() { return f; } } class ClassC extends ClassB { public int m1() { return f; } } 1: class Main { 2: Base x1; void thread1() { 3: x1 = new ClassA(); 4: System.out.println(x1.m1()); } void thread2() { 5: x1 = new ClassB(); 6: System.out.println(x1.m1()); } void thread3() { 7: x1 = new ClassC(); 8: System.out.println(x1.m1()); } } Inheritance when combined with dynamic binding can cause timing problems at runtime. Error•prone constructs • Aliasing • Unbounded arrays Buffer overflow failures can occur if no bound checking on arrays. • Default input processing Occur irrespective of the input. The default action changes the program control flow. Malicious inputs trigger a program failure.

Acceptance of formal methods -- • Formal methods limited in practical development: Hard to understand a formal specification Cannot assess if it is an accurate representation. Assess development costs but harder to assess the benefits. Unwilling to invest in formal methods. Unfamiliar with formal method approach. Difficulty in scaling up to large systems. Incompatibility with agile development methods.

agile -- Agile method are popular •> ? dependable software? A few conflicts: Require certifications (for dependable software) Must produce these certifications Analysis is also required in order to find these inconsistencies. These conflict with the general approach of development. We then develop these erqurierments while developing the system. Developers incorporate agile techniques... Iterative development, agile methods, test•first development. Pure agile is impractical.

An operational profile -- ... Number of inputs Input classes

ATM availability specification -- • Key reliability issues depends on mechanical reliability. • A lower level of software availability is acceptable. • The overall availability Specify availability with 0.999 each day.

ATM reliability specification -- • Key concerns ATMs conduct services as requested Record customer transactions ATM systems are available when required. • Database transaction mechanisms make a correction of transaction problems

Attributes of dependable processes -- Process characteristic Description Auditable The process should be understandable by people apart from process participants, who can check that process standards are being followed and make suggestions for process improvement. Diverse The process should include redundant and diverse verification and validation activities. Documentable The process should have a defined process model that sets out the activities in the process and the documentation that is to be produced during these activities. Robust The process should be able to recover from failures of individual process activities. Standardized A comprehensive set of software development standards covering software production and documentation should be available.

Availability and reliability -- • Reliability The probability of failure•free system operation. • Availability The probability that a system conducts requested services at a point in time.

Availability perception -- • Expressed as a percentage of the time Available to conduct services. • Two factors not considered: The number of users effected by unavailable systems. The length of system failed or unavailable period.

Availability specification -- Availability Explanation 0.9 The system is available for 90% of the time. This means that, in a 24•hour period (1,440 minutes), the system will be unavailable for Roughly, one minute per week.

Availability -- • The time that a software system is available Repair and restart time considered • Availability of 0.001 • Continuously running systems E.g., railway signalling systems.

avoiding failures -- 4. Diverse hardware systems on each channel To avoid errors on same channels 5. Diverse software applications in each channel 6. Several self•checking systems in parallel high•availability is required for such things Flight control system. There is 1. Channel 1 and 2. Channel 2 that get run through the comparator 5 seaprate computers in airbus. Each system can control the software Extensive diverse systems between primary and secondary systems e. Different processors f. Different chipsets from different manufacturers g. Different complexity h. Different programming langauges (by different teams)

bad practices -- Dynamic binding with inheritance can be crazy. Asliasing : more than 1 name to refer to the same state variable Unbounded arrays: buffer overflow failures Default input processing – malicious inputs trigger failures Provide restart capabilities – after long transactions Developers often write programs which address memory locations outside of array bounds Name all constants that represent real•world values Do not use numeric values, always refer to them by name Reduce mistakes for wrong values by using a name rather than a value

Benefits of formal specification -- • Developing a formal specification Analyze system requirements in detail. Detect problems, inconsistencies and incompleteness. • Specification expressed in a formal language Discover inconsistencies and incompleteness. • A formal method correctly transforms a formal specification into a program. • Reduce program testing costs Verify a program formally against its specification.

Benefits of reliability specification -- • Help to clarify stakeholders' needs. • Provide a measurement basis for system tests. • Improve the reliability by different design strategies. • Evidence of including required reliability implementations.

Causes of failure -- • Hardware failure Design and manufacturing errors. • Software failure Errors in its implementation. • Operational failure Human operators make mistakes.

Check array bounds -- • Address a memory location outside of the range of an array declaration. • Bounded buffer vulnerability E.g., Writing executable code into memory by deliberately writing beyond the top element in an array. • Bound checking for an array access Within the bounds of the array.

checks program -- 1. Range checks 2. Size checks min or max input sizes 3. Representation checks Does the input contain invalid pieces, e.g. names or numbers that are invalid 4. Reasonableness checks check the logical information. Is this input reasonable? Provide a handler for all exceptions 1. Report exceptions and provide the related information 2. Conduct alternative processes relatd info to recover from the problem 3. Pass control to a run•time support system 4. A mechanism to provide fault tolerance a. Recover from fault•cuased errors b. Eliminating faults

Classes of error -- • Specification and design errors and omissions. Reveal errors and omissions in requirements. Models generated automatically from source code. Analysis by using model checking find undesirable faults. • Inconsistences between a specification and a program. Refinement methods Programmer mistakes of inconsistencies with specification Discover inconsistencies between programs and specifications.

Code section -- Exception handling code Normal flow of control Exception detected Normal exit Exception processing Method where error occurred Method with an exception handler The main method Throws exception Forwards exception Catches exception Looking for appropriate handler Method without an exception handler Searching the call stack for the exception handler Exception handling • Three possible exception handling strategies Report exceptions and provide the related information. Conduct alternative processes • The related information required to recover from the problem. Pass control to a run•time support system. • A mechanism to provide fault tolerance Recovering from fault•caused errors Eliminating faults

Common Errors in Bounded Buffer Vulnerability -- • Writing data past the end of allocated memory can be detected by OS Generate a segmentation fault error that terminates the process.

Common Errors in Default Input Processing -- • Unexpectedly execute a call to method foo(int) with default input '0', instead of a call to method foo(int, int). Method overloading. Integer '0' is a default value for argument 'b' in the overloaded method foo(int).

Common Mistake in Inheritance and Dynamic -- Binding

Common Mistakes in Dynamic Memory -- Allocation • No matter how much we try, it is very difficult to free all dynamically allocated memory. Even if we can do that, it is often not safe from exceptions. • If an exception is thrown, the ":a" object is never deleted. • Detect memory leaks by Valgrind

Constant Interface Pattern -- • Use final class for Constants • Declare public static final and static import all constants

Constants Best Practice -- values • Use constants reflecting real•world values names Do not use numeric values and always refer to them by name. • Reduce mistakes for wrong values by using a name rather than a value. • Changing constant values Easy to maintain and localize edit locations to make the change.

Dependability achievement -- • Correct system configuration. • Capabilities to resist cyberattacks. • Service recovery mechanisms after a failure

Dependability attribute dependencies -- • Depend on the system's availability and reliability. • Corrupted data by an external attack. • Unavailable to conduct denial of service attacks on a system. • Malicious system virus infection and damage

Dependability costs -- • Dependability costs increase exponentially. • There are two reasons for this Expensive development techniques and hardware for higher levels of dependability. Increased testing and system validation for system dients and regulators.

Dependability economics -- • Accepting untrustworthy systems and pay for failure social and political factors. • Depends on system types that need modest levels of dependability.

Dependable bslash programming bslash guidelines -- ProcessBuilder pb = new ProcessBuilder("external•program"); Timer t = new Timer(); Process p = pb.start(); TimerTask killer = new TimeoutProcessKiller(p); t.schedule(killer, 5000); class TimeoutProcessKiller extends TimerTask { Process p; public TimeoutProcessKiller(Process p) { this.p = p; } @Override public void run() { p.destroy(); } } (1) Limit the visibility of information in a program • Limited access to data for their implementation. • Reduce possibilities of accidental corruption of program state by other components • Control visibility by using abstract data types Private data representation. Limited access to data through predefined operations.

Dependable process activities -- • Design and program inspections Inspection and checking for systems by different people. • Static analysis Automated inspection on the program source code. • Test planning and management Design system test suites. Manage to provide enough coverage of system requirements.

Dependable process characteristics -- • Explicitly defined A defined process model to drive the production process. Data must be collected during the process to prove that the development follows process models. • Repeatable Not rely on individual judgment. Can be repeated across projects and with different team members.

Dependable processes and agility -- • Agile process iterative development, test•first development and user involvement in the development team. • Agile team follows agile process, actions, and agile methods • However, 'pure agile' is impractical for dependable systems.

Dependable processes -- • A well•defined, repeatable software process to reduce faults. • A well•defined repeatable process Not depend on individual skills. • Check whether to use software engineering practice. • Verification and validation (V&V) activities for fault detection.

Dependable programming -- • Standard programming practices Reduce program fault introduction rate. • Support fault avoidance, detection and tolerance

dependable software -- 1. Limit the visibility of information in a program 2. Check al inputs for validity 3. Provide a handler for all exceptions 4. Minimize the use of error•prone constructs 5. Provide restart capabilities 6. Check array bounds 7. Include timeouts when calling external components 8. Name all constants that represent real•world values

Diversity and redundancy examples -- • Redundancy. Backup servers to switch, when failure occurs. • Diversity. Different servers running on different operating systems.

Error Avoidance -- 1. Fault avoidance Programmer created program to avoid errors and minimize faults. 2. Fault detection Developers use verification techinques to remove faults in the system. 3. Developers design software systems that do not cause system failure

Error-prone constructs (2) -- • Human error of misunderstanding or losing track of the relationships between the different parts of the system • Error•prone constructs in programming languages Inherently complex • Avoid or minimize the use of error•prone constructs.

Error-prone constructs (2)3 -- 1. Unconditional branch goto statements 2. Floating point numbers (inherently imprecise with invalid comparisons) 3. Pointers referencing the wrong memory can cause corrupt data 4. Dyamic memory allocation might cause memory overflow Object may not be deleted if you throw an exception. 5. Parallelism •> can result in subtle timing errors due to unforeseen interaction between parallel processes 6. Recursion •> can cause memory overflow as the stack fills up 7. Interrupts – signal that might cause a critical operation to be interrupted 8. Code is not localized. This can result in unexpected behavior.

Error-prone constructs -- • Unconditional branch (goto) statements • Floating•point numbers comparisons. • Pointers Pointers referring to the wrong memory areas can corrupt data. Aliasing can make programs difficult to understand and change. • Dynamic memory allocation Run•time allocation can cause memory overflow.

Examples of functional reliability requirements -- RR1: A pre•defined range for all operator inputs shall be defined and the system shall check that all operator inputs fall within this pre•defined range. (Checking) RR2: Copies of the patient database shall be maintained on two separate servers that are not housed in the same building. (Recovery, redundancy) RR3: N•version programming shall be used to implement the braking control system. (Redundancy) RR4: The system must be implemented in a safe subset of Ada and checked using static analysis. (Process)

failure channels -- 1. Diverse hardware systems on each channel To avoid errors on same channels 2. Diverse software applications in each channel 3. Several self•checking systems in parallel high•availability is required for such things Flight control system. There is 1. Channel 1 and 2. Channel 2 that get run through the comparator a. Different processors b. Different chipsets from different manufacturers c. Different complexity d. Different programming langauges (by different teams)

Failures -- 1. Operators are not familiar with the specification a. How is the system supposed to behave? i. Important to gain knowledge about reliability & use 2. System does not always reflect the user's reliability of perception a. User has certain pre•assumptions about the environment i. Office environment is different from university environment

Fault management -- • Fault avoidance Avoid human errors to minimize system faults. Organize development processes to detect and repair faults. • Fault detection Verification and validation techniques to remove faults. • Fault tolerance Design systems that faults do not cause failures.

Fault tolerance -- • Fault tolerant in critical situations. • Fault tolerance required High availability requirements Failure costs are very high. • Fault tolerance Able to continue in operation despite software failures. • Fault tolerant required against incorrect validation or specification errors, although a system is proved to conform to its specification

Fault-tolerant system architectures -- • Fault•tolerant systems architectures Fault tolerance is essential based on redundancy and diversity. • Examples of situations for dependable architectures: Flight control systems for safety of passengers Reactor systems for a chemical or nuclear

emergency Telecommunication systemsfor 24 slash 7 availability.

Faults and failures -- • Failures Results of system errors resulted from faults in the system • However, faults do not necessarily result in system errors Transient and 'corrected' before an error arises. Never be executed. • Errors do not necessarily lead to system failures Corrected by detection and recovery Protected by protection facilities.

Faults, errors and failures -- Term Description Human error or mistake Human behavior that results in the introduction of faults into a system. For example, in the wilderness weather system, a programmer might decide that the midnight (midnight is 00.00 in the 24•hour clock). System fault A characteristic of a software system that can lead to a system error. The fault is without a check if the time is greater than or equal to 23.00. System error An erroneous system state that can lead to system behavior that is unexpected by system users. The value of transmission time is set incorrectly (to 24.XX rather than 00.XX) when the faulty code is executed. System failure An event that occurs at some point in time when the system does not deliver a service as expected by its users. No weather data is transmitted because the time is invalid.

Formal approaches -- • Verification•based approaches Different representations of a software system are proved to be equivalent. Demonstrate the absence of errors. • Refinement•based approaches A system representation is transformed into a lower•level representation. Correct transformation results in equivalent representations.

formal benefits -- Develop formal Formal language for all of this Which therein finds inconsistencies Hopefully turn this into a program 1. Hard to understand 2. Hard to apply to a large system 3. Cannot assess if accurate

Formal methods and dependability -- You define a formal model of the software system. Formal methods are based on the mathe Formal methods are cost•effective for s/w eng. Considerably reduce certain tyeps of software error. 1. Verification based approaches. a. Different representations of the software systems b. Demonstrates the absence of errors? Verification based approaches 2. Refine•based approaches a. Low•level representations. Correct transformations result in equivalent representations.

Formal specification -- • Formal methods Development approaches based on mathematical analysis. • Formal methods include Formal specification; Specification analysis and proof; Transformational development; Program verification. • Reduce programming errors and cost for dependable systems.

formal -- 1. Reduce # of errors 2. Reduce faults or runtime errors 3. Dependable systems engineering 4. Use of formal methods in certain areas (high system failure cost reducation).

Functional reliability requirements -- • Checking requirements Identify incorrect data before it leads to a failure. • Recovery requirements Help the system recover from a failure. • Redundancy requirements Specify redundant system features. • Process requirements Specify software development processes. 31

Functional reliability requirements2 -- 1. Checking requirements a. Identify incorrect data before it leads to a failure 2. Recovery requirements a. Help the system recover from a failure 3. Redundancy requirements a. Specify reduantnat system features 4. Process requirements a. Specify software development processes

Good practice guidelines for dependable -- programming

Hardware fault tolerance -- • Triple•modular redundancy (TMR). • Three replicated identical components Receive the same input and their outputs are compared. • One different output Ignored based on the assumption of component failure. • Most faults caused by component failures A low probability of simultaneous component failure.

Holistic system design -- • Interactions and dependencies between system layers Example: regulation changes causes changes in applications. • For dependability, a systems perspective is essential Software failures within the enclosing layers. Failures in adjacent layers affects software systems.

Importance of dependability -- • The costs of system failure is high if the failure leads to economic losses.

Improvements in practice -- • Multi•version programming leads to significant improvements in reliability and availability. Diversity and independence • In practice, observed improvements are much less significant • Considerable costs to develop multi•versions of systems.

Include timeouts when calling external components -- No indication of a failure. Failure of a remote computer can be 'silent' In a distributed system. • Set timeouts on all calls to external components. • Assume failure and take actions to recover from errors After a defined time period without a response.

Input Check for Validity -- • Taking inputs from their environment based on assumptions about the inputs. • Program specifications rarely defined Inconsistent inputs with the assumptions. • Unpredictable program behavior Unusual inputs Threats to the security of the system. • Check program inputs before processing Considering the assumptions about the inputs.

Input Output Mapping -- 1. number of users affected by available services Failure in the middle of the night is less important 2.Length of system failure

Insulin pump reliability specification -- • Probability of failure (POFOD) metric. • Transient failures Repaired by user actions, such as, recalibration of the machine. demands. • Permanent failures Re•installed by the manufacturer. Occur no more than once per year. POFOD < 0.00002.

Key points -- • Software diversity is difficult to achieve because it is practically impossible to ensure that each version of the software is truly independent. • Dependable programming relies on including redundancy in a program as checks on the validity of inputs and the values of program variables. • Statistical testing is used to estimate software reliability. It relies on testing the system with test data that matches an operational profile, which reflects the distribution of inputs to the software when it is in use.

Layers in the STS stack -- • Business processes Processes involving people and systems • Organizations Business activities for system operations • Society Laws, regulation and culture

N-version programming -- • Multiple versions of a program execute computations. Odd number of computers involved, e.g., three versions. • The results are compared using a voting system. • The correct result is determine by the majority result. • The notion of triple•modular redundancy, as used in hardware systems.

Non-functional reliability requirements -- • Non•functional reliability requirements Reliability specifications using one of the reliability metrics (POFOD, ROCOF or AVAIL). • Used for many years in safety•critical systems Uncommon for business critical systems. • Need precise measurement about reliability and availability expectations.

Non-functional reliability requirements2 -- a. Specs using reliability metrics b. POFOD probability of fault on demand c. ROCOF rate of occurrence of fault d. AVAIL: availibity

Operational profile generation -- • Automatic data generation, if possible. Difficult for interactive systems. Easy for 'normal' inputs • Difficult to generate 'unlikely' inputs (anomalies) and test data for these anomalies. • Unknown usage pattern of new systems. • Changeable operational profiles Non•static but dynamic E.g., learn about new systems, changing usage patterns.

Operational profiles -- • A set of test data Frequency matches the actual frequency from 'normal' usage of the system. The number of times the failure event occurred. • A close match with actual usage The measured reliability Reflect the actual usage of the system. • Generate from real data Collect from an existing system Assumption of the usage pattern of a system

Operational profiles2 -- 1. A set of test data freque. Matches normal usage & number of times the failure event occurred 2. Close match with actual usage reflect actual usage 3. Generated from real data Think geometric distrubtion 1. Automatic data generation 2. Difficult to generate unlikely inputs and anomalies and test for these 3. Unknown usage pattern of a new system 4. Changeable operational profiles a. Non•static, but dynamic b. Learn about new systems, changing usage patterns

Other dependability properties -- • Repairability Capability of being repaired in the event of a failure • Maintainability Capability of being adapted to new requirements • Error tolerance Capability to tolerate failures due to user input errors

Output -- selector N software versions Agreed result Fault manager Input N•version programming • The different versions of a system Designed and implemented by different teams. • Assuming a low probability of making same mistakes Different algorithms used • Empirical evidence Commonly misinterpret specifications Use same algorithms in different systems.

Perceptions of reliability -- • Not always reflect the user's reliability perception The assumptions about environments for a system are incorrect • Different usage of a system between in an office environment and in a university environment. The consequences of system failures affects the perception of reliability.

Principal properties -- • Security Capability of resisting accidental or deliberate intrusions. • Resilience A judgment of how well a system can maintain the continuity of its critical services.

Probability of failure on demand (POFOD) -- • The probability of the system failure when a service request is made. Useful when demands for service are relatively infrequent. • Implement appropriate protection systems Demand services occasionally. Serious consequence due to failed services. • Develop for safety•critical systems E.g., emergency shutdown system in a chemical plant.

Problems with design diversity -- • Tend to solve problems using same methods • Characteristic errors Different teams making same mistakes. Making mistakes in same parts. Specification errors propagated to all implementations

Problems with redundancy and diversity -- • Adding diversity and redundancy increases complexity. • Increase the chances of error Unanticipated interactions between redundant components. • Advocate simplicity to decrease software dependability. E.g., an Airbus product is redundant slash diverse; a Boeing product has no software diversity

Process diversity and redundancy -- • Process activities Not depend on a single approach, such as testing. • Redundant and diverse process activities. • Multiple process activities complement each other Cross•checking techniques avoid process errors

Protection systems -- • A specialized system Associated with other control system. Take emergency action to deal with failures. E.g., System to stop a train or system to shut down a reactor • Monitor the controlled system and the environment. • Take emergency action to shut down the system and avoid a catastrophe.

protection systems2 -- Specialized system a. Associated with controls system b. Action to deal with failures c. E.g. system to stop a train or to shut down a reactor Monitor the controlled system and the environment Take action to shut down the system and avoid a catastrophe Developers apply redundancy for protection systems •> regarding monitoring and control capabilities. 1. Diverse 2. Protection system for redundancy

Protection -- sensors System environment Actuators Controlled equipment Control system Protection system Sensors Protection system functionality • Protection systems for redundancy Control capabilities to replicate in the control software. • Diverse protection systems Different technology used in the control software. • Need to expend in validation and dependability assurance. • A low probability of failure for the protection system.

public final class Constants { -- private Constants() { } public static final double PLANCK_CONSTANT = 6.62606896e•34; } import static math.Constants.PLANCK_CONSTANT; import static math.Constants.PI; class Calculations { public double getReducedPlanckConstant() { return PLANCK_CONSTANT slash (2 * PI); } } Reliability measurement

Rate of fault occurrence (ROCOF) -- • System failure occurrence rate • ROCOF of 0.002 • Reliable systems needed Systems perform a number of similar requests in a short time E.g., credit card processing system. • Reciprocal of ROCOF is Mean time to Failure (MTTF) Systems with long transactions System processing takes a long time. MTTF is longer than expected transaction length.

Redundancy and diversity -- • Redundancy Keep more than a single version. • Diversity Provide the same functionality in different mechanism. • Redundant and diverse components should be performed independently E.g., software written in different programming languages.

Regulated systems -- • Critical systems are regulated systems Approved by an external regulator. E.g., nuclear systems and air traffic control systems • A safety and dependability case Approved by the regulator. Create the evidence for systems' dependability, safety and security.

Regulation and compliance -- • The general model of economic organization Universal in the world. Offer goods and services and make a profit. • Ensure the safety of their citizens Follow standards to ensure that products are safe and secure.

Reliability achievement -- • Fault avoidance Development technique to minimise the possibility of mistakes or reveal mistakes. • Fault detection and removal Verification and validation techniques to increase the probability of correcting errors. • Fault tolerance Run•time techniques to ensure that faults do not cause errors.

Reliability and specifications -- • Reliability Defined formally w.r.t. a system specification A deviation from a specification. • Incomplete or incorrect specifications • Unfamiliar with specifications Unaware how the system is supposed to behave. • Perceptions of reliability

Reliability in use -- • Reliability not improved by X% by removing faults with X% • Program defects rarely executed Not encountered by users. Not affect the perceived reliability. • Users' operation patterns to avoid system features. • Software systems with known faults Considered reliable systems by users.

Reliability measurement problems -- • Operational profile uncertainty Inaccurate operational profile that does not reflect the real use of the system. • High costs of test data generation Expensive costs to generate test datasets for the system. • Statistical uncertainty A statistically significant number of failures for computation. Highly reliable systems rarely fail. • Recognizing failure Conflicting interpretations of a specification about unobvious failures.

Reliability measurement -- Compute observed reliability Apply tests to system Prepare test data set Identify operational profiles

Reliability metrics -- • Units of measurement of system reliability. • Counting the number of operational failures and the period length that the system has been operational. • Assess the reliability (e.g., critical systems) Long•term measurement techniques • Metrics Probability of failure on demand Rate of occurrence of failures

reliability problems -- 1. Operational profile uncertainty a. Inaccurate operational profile that does not reflect the real system 2. High costs of test data generation 3. Statistical uncertainty, highly reliable systems rarely fail(so hard to test) 4. Recognizing failure – conflicton interpretations of a specification

about unobvious failures

reliability systems concerns -- A. ATMS CONDUCTED services as requested B. Record customer transactions C. Atm systems are available when required 1. Need to identify system services in order to identify system reliability

Reliability testing -- • Reliability testing (Statistical testing) Assess whether a system reaches the required level of reliability. • Not part of a defect testing process Datasets for defect testing do not include actual usage data. • Data sets required to replicate actual inputs to be processed.

reliability testing2 -- 1. Identify operational profiles 2. Prepare test data set 3. Apply tests to system 4. Compute observed reliability

runtime evolution -- 1. Build time 2. Hire time 3. Deploye time

Safety regulation -- • Regulation and compliance applies to the sociotechnical system. • Safety•related systems Certified as safe by the regulator. • Produce safety cases to show systems follow rules and regulations. • Expensive to document certification.

Self-monitoring architectures -- • Multi•channel architectures System monitoring its own operations Take action if inconsistencies are discovered • The same computation is carried out on each channel Compare the results Producing identical results assumes correct system operation A failure exception is reported when different results arise.

Software diversity strategies -- 1. Different languages (programming) 2. Different design methods and tools 3. Different algorithms in the implementations Diverse version of systes hsould have no dependence (so they fail in completely different ways) •> problem similarity since ppl are not intellectually diverse (many people make similar mistakes in similar locations due to software complexity.).

Software diversity -- • Fault tolerance depend on software diversity • Assume that different implementations fail differently Independent implementations Uncommon errors • Strategies Different programming languages Different design methods and tools Explicit specification of different algorithms

Software diversity2 -- Fault tolerant • > depends on software diversity. Assume that different implementations fail differently 1. Uncommon errors 2. Independent implementations

Software reliability -- • Dependable software is expected • However, some system failures are accepted. • Software systems have high reliability requirements E.g., critical software systems • Software engineering techniques for reliability requirements. E.g., medical systems and aerospace systems

Software usage patterns -- Possible inputs User User User Erroneous inputs

Specification dependency -- • Software redundancy susceptible to specification errors. Incorrect specification causes system failures. • Complex software specifications Hard to perform validation and verification. • Developing separate software specifications.

Specification dependency2 -- Software redundancy susceptible to specification errors incorrect specification Complex software specifications Hard to perform comparisons

Specifying reliability requirements -- • Availability and reliability requirements for different types of failure. Low probability of high•cost failures • Availability and reliability requirements for different types of system service. Tolerate failures in less critical services. • A high level of reliability required. Other mechanisms for reliable system services.

Splitter -- Comparator Input value Output Status Splitter Comparator Output Status Input Filter Filter Filter Filter Filter Status Status Status Output Output Output Output Airbus architecture discussion Each system is able to perform the control software. • Extensive diverse systems between primary and secondary systems: Different processors Different chipsets from different manufacturers. Different complexity •• only critical functionality in secondary. Different programming languages by different teams.

Statistical testing -- • Testing software for reliability rather than fault detection. • Measuring the number of errors Predict reliability of the software More errors, compared to the one in the specification. • An acceptable level of reliability Test software systems and repair or improve them until systems reach the level of reliability.

Strategies -- 4. Different languages (programming) 5. Different design methods and tools 6. Different algorithms in the implementations

System dependability -- • The most important system property is the dependability. • Reflect the user's degree of trust in that system. • Reflect the extent of the user's confidence that it will operate as users expect. • Cover the related attributes: reliability, availability and security.

system input output mapping -- Ie Input set Oe Output set Program Inputs causing erroneous outputs Erroneous outputs

System reliability requirements -- • Functional reliability requirements Define system and software functions Avoid, detect or tolerate faults Not lead to system failure. • Software reliability requirements Cope with hardware failure or operator error. • Non•functional reliability requirements • A measurable system attribute specified quantitatively. • E.g., the number of failures and the available time.

Systems and software -- • Software engineering is part of system engineering process. • Software systems are are essential components of systems based on organizational purposes. • Example The wilderness weather system is part of forecasting systems Hardware and software, forecasting processes, the organizations, etc.

The increasing costs of residual fault removal -- Cost per error detected Few Number of residual errors Many Very few

The principal dependability properties -- Dependability Availability Reliability Security Safety Resilience The ability of the system to protect itself against deliberate or accidental intrusion The ability of the system to resist and recover from damaging events The ability of the system to operate without catastrophic failure The ability of the system to deliver services as specified The ability of the system to deliver services when requested

The sociotechnical systems (STS) stack -- Equipment Operating system Communications and data management Application system Business processes Organization Society Systems engineering Software engineering

try { -- System.out.println ("a[" + i + "]=" + a[i] + "\n"); } } catch (Exception e) { System.out.println ("error = " + e); } catch (ArrayIndexOutOfBoundsException e) { System.out.println ("ArrayIndexOutOfBoundsException"); } A.java: error: exception ArrayIndexOutOfBoundsException has already been caught catch (ArrayIndexOutOfBoundsException e) (3) Provide a handler for all exceptions • An error or some unexpected event E.g., a power failure. • Exception handling constructs Responding and handling exception events Change the program execution flow • Using normal control constructs to handle exceptions? A number of additional statements Significant overhead Tedious and error•prone

Use of formal methods -- • The principal benefits Reduce faults or runtime errors. • Applicable main area: Dependable systems engineering. • Cost•effective formal methods Reduce high system failure costs

Validity checks -- • Range checks Check the input's range. • Size checks Check the input's maximum or minimum size. • Representation checks Check the input's expression for the representation E.g. names do not include numerals. • Reasonableness checks Check the input's logical information. E.g., reasonable rather than an extreme value.

var str year -- var str year value 'b' 'o' 'u' 'n' 'd' 'l' 'e' 's' ???? Common Errors in Bounded Buffer Vulnerability • Return a heap•allocated copy of the string with all uppercase letters. • No bounds checking on its input. • Overflow buf with the unbounded call to strcpy().