CITS5501 Software Testing and Quality Assurance
Risk, intro to formal methods

Unit coordinator: Arran Stewart

Risk

Motivation

Why do we look at risk?

- Because every project, even the simplest, involves risks

Motivation

Why do we look at risk?

- Because every project, even the simplest, involves risks
- Because people are usually not good at allowing for them

Motivation

Why do we look at risk?

- Because every project, even the simplest, involves risks
- Because people are usually not good at allowing for them
- Because risk assessment determines what other testing and quality assurance activities you undertake

Example

- What sort of requirements documentation should we require for, say, a website?
- Is there complicated or confusing terminology involved? Is there a risk of miscommunication or misunderstanding between us and our client? (e.g. perhaps for a website to be used internally by a financial services enterprise)
  We might try to reduce the risk of miscommunication by ensuring we have *glossaries* of terms, and that client representatives and developers agree on their understanding of these.

Example

- Is the domain, or the technologies we're likely to have to use, complicated? Is there a risk we have modelled it incorrectly? (Example: software to manage telephone exchange systems; systems which are inherently concurrent, as they are notoriously difficult to reason about.)
  We might reduce risk by putting additional effort into modelling the domain and/or the system – perhaps through a notation such as UML, perhaps even through formal (i.e. mathematical) specifications of the domain and/or system.

Example

- On the other hand, if this is a website for a local kids' football club, then many of these risks are unlikely to be relevant, or to cause significant problems if they do – so effort on those activities would be misplaced.

What is risk?

- Something that *might* happen, which would cause loss
  - If it will *definitely* happen, that's not a risk – it's usually called a "constraint"
  - If it doesn't result in some kind of unwanted consequence (i.e., a loss), it's also not a risk

What are some sorts of risk?

One categorization (Pressman):

- Project risks
  - Things that could affect the project plan or schedule
- Technical risks
  - Risks resulting from the problem being *harder to solve* than we thought
- Business risks
  - Things that threaten the viability, as a product, of the software to be built
  - (Could be commercial viability, but also applies to in-house or even open source software)

Sorts of risk – project risks

Project risks:

- Things that could affect the *project plan and/or schedule*, causing it to slip and costs to increase
- Examples: personnel move or resign, resources turn out to be more expensive/take longer to acquire than estimated, exchange rates shift

Sorts of risk – technical risks

Technical risks:

- Risks resulting from the problem being *harder to solve* than we thought
- They threaten the quality and timeliness of the software to be produced
- If they eventuate, implementation may become difficult or impossible
- Examples: design turns out to be infeasible, dependencies have bugs, language/framework turns out to be difficult to maintain

Sorts of risk – business risks

Business risks

- Things that threaten the viability, as a product, of the software to be built (even if there are no technical or project barriers to it being implemented on time and within budget)
- Examples: over-estimating the market for a product; being beaten to release by a competitor

What are some sorts of risk? (2)

Another (due to Robert N. Charette, 1989):

- Known risks
  - Those risks that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date)
- Predictable risks
  - Those risks that are extrapolated from past project experience (e.g., past turnover)
- Unpredictable risks
  - Those risks that can and do occur, but are extremely difficult to identify in advance

Reactive vs. proactive risk strategies

- *Reactive* risk strategies
  - "Don't worry, I'll think of something"
  - The majority of software teams and managers rely on this approach
  - Nothing is done about risks until something goes wrong
    - The team then flies into action in an attempt to correct the problem rapidly ("fire fighting")
- *Proactive* risk strategies
  - Steps for risk management are followed
  - Primary objective is to *reduce* risk and to have a contingency plan in place to handle unavoidable risks in a controlled and effective manner

Steps for risk management

1. Identify possible risks; recognize what can go wrong
2. Analyze each risk to estimate the probability that it will occur and the impact (i.e., damage) that it will do if it does occur
3. Rank the risks by probability and impact
   - Impact may be negligible, marginal, critical, and catastrophic
4. Develop a plan to manage (some of) those risks

Risk identification

- A systematic attempt to identify risks to the project
- Because if they aren't identified, how can they be planned for?
- Two main sorts of risk:
  - Generic risks
    - Risks that are common to every software project
  - Product-specific risks
    - Risks that can be identified only by those a with a clear understanding of the technology, the people, and environment specific to the software that is to be built

Risk identification (2)

- One way of identifying risks – use a *risk checklist*
- Focuses on known and predictable risks in specific subcategories

Risk checklists

Some typical categories of items on risk checklists:

- Product size – risks associated with overall size of the software to be built
- Business impact – risks associated with constraints imposed by management or the marketplace
- Customer characteristics – risks associated with sophistication of the customer and the developer's ability to communicate with the customer in a timely manner
- Process definition – risks associated with the degree to which the software process has been defined and is followed
- Development environment – risks associated with availability and quality of the tools to be used to build the project
- Technology to be built – risks associated with complexity of the system to be built and the "newness" of the technology in the system
- Staff size and experience – risks associated with overall technical and project experience of the software engineers who will do the work

A project risk questionnaire

1. Have top software and customer managers formally committed to support the project?
2. Are end-users enthusiastically committed to the project and the system/product to be built?
3. Are requirements fully understood by the software engineering team and its customers?
4. Have customers been involved fully in the definition of requirements?
5. Do end-users have realistic expectations?
6. Is the project scope stable?
7. Does the software engineering team have the right mix of skills?
8. Are project requirements stable?
9. Does the project team have experience with the technology to be implemented?
10. Is the number of people on the project team adequate to do the job?
11. Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

Risk estimation

- Risk estimation attempts to rate, for each risk:
  - The probability that it eventuates
  - The magnitude of loss associated with the risk, should it eventuate
- The project planner, managers, and technical staff perform risk estimation so that risks can be prioritized, and limited resources be allocated where they will have the most impact

Risk tables

- A *risk table* provides a project manager with a simple technique for risk projection
- It consists of five columns
  - Risk summary – short description of the risk
  - Risk category – (see the slides on risk categorization)
  - Probability – estimation of risk occurrence based on group input
  - Categorization of impact – e.g. (1) catastrophic (2) critical (3) marginal (4) negligible
  - RMMM – pointer to a paragraph in the Risk Mitigation, Monitoring, and Management Plan

Developing a Risk Table

- List all risks in the first column (by way of the help of the risk item checklists)
- Mark the category of each risk
- Estimate the probability of each risk occurring
- Assess the impact of each risk based on an averaging of the four risk components to determine an overall impact value
- Sort the rows by probability and impact in descending order
- Draw a horizontal cutoff line in the table that indicates the risks that will be given further attention

Risk likelihood

How do we categorize likelihood?

Typically, organizations will divide "likelihood" into about 3 or 5 categories.

e.g. One division might be:

- unlikely; medium likelihood; highly likely

Another might categorize likelihoods as:

- rare, unlikely, possible, likely, highly likely

Any scheme which uses more than 5 categories might appear to offer very fine granularity, but this is probably illusory – humans are not good at making fine-grained estimations of likelihood.

Risk likelihood

Each level of likelihood is given a number - e.g. you might have:

- very low (1), low (2), medium (3), high (4), very high (5)

Risk impact

In addition to likelihood, we try to estimate the *impact* that an incident would have on the project (and perhaps the organization as a whole) if it eventuated.

As with likelihoods, impact is usually divided into 3 or 5 categories – for instance:

- low impact, medium impact, high impact.

Each of these comes with a description – e.g. "low impact" might mean "causes minor (less than 2-week) delays or budget overruns (less than $2000)".

"High impact" usually means something like "Would damage the project severely, making it unviable". It might include injury to life, or the organization incurring legal liabilities.

As with likelihoods, each level impact is given a number.

Risk mitigation, monitoring, and management

An effective risk strategy for dealing with risk must consider three issues

- (Note: these are not mutually exclusive)
  - Risk mitigation (i.e., avoidance)
  - Risk monitoring
  - Risk management and contingency planning
- Risk mitigation (avoidance) is the primary strategy and is achieved through a plan

Example

Example: Risk of high staff turnover

- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market)
- Mitigate those causes that are under our control before the project starts
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave
- Organize project teams so that information about each development activity is widely dispersed
- Define documentation standards and establish mechanisms to ensure that documents are developed in a timely manner
- Conduct peer reviews of all work (so that more than one person is "up to speed")
- Assign a backup staff member for every critical technologist

Intro to formal methods

Sources

Some useful sources, for more information:

- Pressman, R., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 2005
- Huth and Ryan, *Logic in Computer Science*
- Pierce et al, *Software Foundations vol 1*

Overview

- When doing software engineering – specifying and developing software systems – the activities done can be done with varying levels of mathematical rigor.
- For instance, we could write a requirement
  - informally, just using natural language, and perhaps tables and diagrams. This is easy, but can be imprecise and ambiguous (and hard to spot when that has occurred)
  - semi-formally, perhaps using occasional mathematical formulas or bits of pseudocode to express what's required
  - mostly using mathematical notation, with a bit of English to clarify what the notation represents. This is typically a lot more work, and it can be harder to ensure the notation matches our intuitive idea of whhat the system should do, but has little or no vagueness or ambiguity.

Overview (2)

- Things towards the "more formal" side of this spectrum will tend to get called "lightweight formal methods" or "formal methods".
- We'll start with an example formal method (verification of programs using Hoare logic), then come back to the definition, and look at other sorts of formal methods.

A typical approach

Often, we'll apply formal methods in the following way:

- We'll have some specification – some property that we want our system to have
  - e.g., that it calculates the factorial of a natural number; or never gets deadlocked; or has certain security properties.
- And we'll have something representing the system – this is called a *model*
  - This could be actual code, or it could be annotated code, or it could be some more abstract model of the system (like state machines, which we have seen earlier)
- And we will try to show that the model meets the specification.

Rationale

- Why use formal methods?
- Building reliable software is hard.
  - Software systems can be hugely complex, and knowing exactly what a system is doing at any point of time is likewise hard.
- So computer scientists and software enginners have come up with all sorts of techniques for improving reliability (many of which we've seen) – testing, risk management, quality controls, maths-based techniques for reasoning about the properties of software
  - And this last sort of technique is what we call formal methods.

Rationale

- By *reasoning* about the properties of software – i.e., proving things about it – we can get much greater certainty that our programs are reliable and error-free, than we can through testing
- Testing is a sort of *empirical investigation* – we go out and check whether we can find something (bugs, in this case)
- But if we don't find it, that doesn't mean that whatever we were looking for doesn't exist – we may not have looked hard enough or in the right places.
  - (People once thought it was an eternal and obvious truth that there weren't such things as black swans, but it turned out they weren't looking in the right places.)

Example

- Consider a bit of code, in some Python-like language, for multiplying `i` by `j`
  - (We will look at more complex examples later)

```
n := 0
while j > 0:
  n := n + i
  j := j - 1
return n
```

How can we show that this code is correct? (Assuming it is.)

Example (2)

- We could try it on a number of inputs. (We might try using 0, values near 0, and values not especially near zero.)
- We could inspect the code and see if it conforms to our idea of multiplication
- We could try to *prove* that, once the code has finished executing, for *any* integers `i` and `j`, n will equal `i * j`

Program verification

- *Proofs of correctness* use techniques from formal logic to prove that if the starting state (i.e., "input" variables) of a program satisfies particular properties, than the end state after executing a program (i.e., "output" variables) satisfies some other properties.
- The first lot of properties are called *preconditions* (assertions that hold prior to execution of a piece of code), and the second lot are *postconditions* (assertions that hold after execution)

Program verification (2)

For instance, if our program P is the snippet of code from before –

```
n := 0
while j > 0:
  n := n + i
  j := j - 1
return n
```

– then our input variables are `i` and `j`, our output variable is `n`, and our precondition might be

- `i` and `j` are any integers

and our postcondition might be

- `n` equals (the original value of) `i` times (the original value of) `j`

Assertions

- Formally, these assertions are *predicates* that hold of variables. (i.e., they're things that are true or false, given variables as input).
- "is greater than 3" is a predicate.

Example assertions

- Bounds on elements of the data:

  $$n \geq 0$$

- Ordering properties of the data:

  $$\text{for all } j : 0 \leq j < n - 1 : a_j \leq a_{j+1}$$

- "Finding the maximum"

  e.g. Asserting that p is the position of the maximum element in some array $a[0..n-1]$

  $$0 \leq p < n \vee (\text{for all } j : 0 \leq j < n : a_j \leq a_p)$$

Assertions in a program - multiplication

We'll distinguish our input variables from our working variables, by giving the, different names. We'll make a and b input.

```
{ pre : ⊤ } // no precons -- "top" or "true"
i := a
j := b
n := 0
while j > 0:
  n = n + i
  j = j - 1
{ post: n = a * b }
```

Assertions in a program - old values

- The problem here is that we want to refer to the *old* values of `i` and `j` in our postcondition
- Systems and languages for program verification often will have a special syntax for this, to avoid having to introduce new variables
- e.g. In the Eiffel language, you can refer to the value of `i` in the pre-state as just `old i`
- In languages or notations that don't have such syntax, we may have to manually do the work ourselves.
  - Often, we'll use prime marks (') to indicate a subsequent state of a variable
  - e.g. i' often means "the *next* value of i" (e.g. after a statement has executed, or a loop has executed, or we have transitioned to a new state)

Assertions in a program - square root example

```
# we want to find the square root of n
a := 0
b := n + 1
{ pre : n ≥ 0 }
while a + 1 != b:
  d := (a + b) div 2
  if d * d <= n:
    a := d
  else:
    b := d
{ post: a² ≤ n < (a + 1)² }
```

- Where we have a sequence [*preconditions*, *code fragment*, *postconditions*], we call this a **Hoare triple** (after logician and computer scientist Tony Hoare of Oxford, who also invented the Quicksort algorithm, amongst other things)

Invariants

Where we have an assertion which should hold before and after *every* iteration of a loop, we call this an *invariant*

```
a := 0
b := n + 1
{ pre : n ≥ 0 }
{ inv P: a < b ≤ n + 1 ∧ a² ≤ n < b² }
  # ^ an *invariant*
while a + 1 != b:
  d := (a + b) div 2
  if d * d <= n:
    a := d
  else:
    b := d
{ post: a² ≤ n < (a + 1)² }
```

Termination

- What if we have a loop that doesn't terminate? What can we conclude then?
- There are two ways of handling this:
  - *partial correctness* doesn't require a program to terminate
  - *total correctness* does

Partial correctness

- if we have a triple $\{ \phi \} P \{ \psi \}$, then we say that it is "satisfied under partial correctness" if
  - for all states which satisfy the preconditions, ϕ
  - the state resulting from executing P satisfies the postcondition, ψ, *if* P actually terminates.
- (If the pre- or post-conditions contain multiple assertions, we can always "and" them all together into just one assertion.)

Partial correctness

- So if our precondition is just $\top$, and our postcondition is $n = i * j$, then here is a program that has partial correctness:

```
while true:
  pass
```

- It *cannot* terminate, so partial correctness is satisfied.

Partial correctness

- So what use is that?
- Well, often it's handy to tackle a proof of correctness in two stages:
  1. Prove that *if* the program terminates, *then* it produces the results we want
  2. Prove that the program terminates
- And then we have *total* correctness.

Total correctness

- if we have a triple $\{ \phi \} P \{ \psi \}$, then we say that it is "satisfied under total correctness" if
  - for all states which satisfy the preconditions, ϕ
  - the state resulting from executing P satisfies the postcondition, ψ, and P is guaranteed to terminate.

Proving things

- So how do we actually build up our proof?
  - We need rules about how to combine triples together.
  - We cover these rules now

Composition

- The composition rule says:

  If we have $\{ a \}\ P_1\ \{ b \}$ and $\{ b \}\ P_2\ \{ c \}$

  then we can derive $\{ a \}\ P_1; P_2\ \{ c \}$

Assignment

- The assignment rule states that, after an assignment, any predicate that was previously true for the right-hand side of the assignment now holds for the variable on the left-hand side.
- Formally:

$$A[E/x] \;\; \mathrm{x} := \mathrm{E} \; \{ \; A \; \}$$

Here, A represents some (probably and'ed together) assertion about variables;
$A[E/x]$ means, "A, but wherever x appears, substitute for it the expression E instead"

- Which gives us the meaning we want: whatever *was* true of E is now true of x.

Conditional

- We represent "if" using the conditional rule
- The rule is:

  If we have

  $$\{\ a \wedge C\ \}\ P_1\ \{\ b\ \},\ \text{and we have}\ \{\ a \wedge \neg C\ \}\ P_2\ \{\ b\ \}$$

  then we can derive

  $$\{\ a\ \}\ \text{if}\ C\ \text{then}\ P_1\ \text{else}\ P2\ \{\ b\ \}$$

- (i.e., If we know that we end up with b holding regardless of whether C is true or not, then we know it must be true after the `if` statement)

"Partial while"

- The rule for partial correctness of while loops is:

  If we have a triple $\{ A \wedge B \}\ P\ \{ A \}$

  then we can derive:

  $$\{ A \}\ \text{while } B \text{ do } P \text{ done } \{ \neg B \wedge A \}$$

- Here, A is what is called the **loop invariant** – it is something that should be *preserved by* (i.e. is true before and true after) the loop body.
- After the loop is done, A should still hold, no matter how many times the loop executed.
- *if* we got out of the loop, it must be the case afterwards that $\neg B$ (since otherwise, "while B" would've continued)

"Total while"

- If we can prove partial correctness for a while loop;
  and we can prove the loop terminates;
  then we've proved total correctness.
  - (There is syntax for this in Hoare logic, but we won't go into it.)

Worked example

- program fragment:

```
{ pre : b ≥ 0 }
i := a
j := b
n := 0
while j > 0:
  n = n + i
  j = j - 1
{ post: n = a * b }
```

- . . . we try to work out a loop invariant . . .
- invariant: $(j * i) + n = a * b$
  i.e., if we add our "result so far" (n), which is increasing, to the product of j and i (where j is decreasing), that should remain constant, and is equal to $a * b$.

Worked example

- let our loop invariant $\boldsymbol{L}$ be $(j * i) + n = a * b$

```
{ pre : b ≥ 0 }
i := a
j := b
n := 0
{ j > 0 ∧ L }
while j > 0:
  n = n + i
  j = j - 1
{ j ≤ 0 ∧ L }
{ post: n = a * b }
```

- The "partial while" rule lets us put assertions round the while loop.
- So one thing we know now is that *if* the loop finishes, then $\boldsymbol{L}$ will still hold.
- If j ended up being 0, then that would be handy, because

  $$(j = 0) \wedge (j * i + n = a * b)$$

  implies

  $$n = a * b$$

  which is what we would like to prove.

Worked example

- So if we can prove that
  - at the end of the loop, $j = 0$,
  - and that the loop terminates,
  - and that the initialization statements and the program preconditions satisfy the loop precondition

  . . . then we've proved that the program does what we want. (Assuming $b \geq 0$, which is a precondition for the whole program.)

Worked example

- We won't work through the rest of the proof in detail, but hopefully you have an idea of how it will go.
- Roughly, to prove that the loop ends, and that j is 0 at the end of it:
  Since j is some finite number, and we are subtracting one off it each time, some math and logic tells us that we must end sometime, and that j will end up equaling zero.

Worked example

- Coming up with the loop invariant is usually the hard part; the other rules can be applied in a more automatic kind of way.
- Edsger Dijkstra said that to properly understand a while statement is to understand its invariant.

Sorts of formal methods

Back to formal methods

- So here, our *specifications* were assertions about variable values before and after the program executed, written as mathematical formulas.
- We used a method that was largely *manual* – putting assertions around fragments of code.
- Some bits of that could be partly automated – the rules for composition and assignment could be done by machine
- The loop invariant, however, requires ingenuity to come up with
- Our *model* of the system was, in fact, the code itself.
  - (The code is still just a *model*, a simplification, of the actual running binary. It isn't itself the binary.
    We also might ignore such things as limits on sizes of ints, if we are happy to accept that our proof only applies, if the ints are sufficiently small.)

Categorizing formal methods

- We can categorize formal methods in various ways . . .

Categorizing formal methods

Degree of formality:

- how formal are the specifications and the system description?
- in natural language (informal), or something more mathematical?

Categorizing formal methods

Degree of automation:

- the extremes are fully automatic and fully manual
- most computer-aided methods are somewhere in the middle

Categorizing formal methods

Full or partial verification of properties in the specification

- What is being verified about the system? Just one property? (e.g., that it does not deadlock, say – common for concurrent systems)
- Or many/all properties?
  - (This is usually very expensive, in terms of effort)

Categorizing formal methods

Intended domain of application:

- e.g. hardware vs software;
- reactive vs terminating;
  - reactive systems run a theoretically endless "loop" and aren't intended to terminate – they just keep *reacting* to an environment
  - e.g. operating systems, embedded hardware (modelled with state machines, often)
  - terminating systems terminate, usually with some sort of *result*
- sequential vs concurrent

Categorizing formal methods

pre- vs post-development:

- Is verification done early in development, vs later or afterwards?
- Earlier is obviously better, since things are much more expensive to fix if early, if it turns out our system *doesn't* meet the specs

Categorizing formal methods

- But sometimes the system comes first, then the verification
- Often true for programming languages . . .
  - e.g. Java was released in 1995, and in 1997, a machine-checked proof of "type soundness" of a subset of Java was proved.[1]
  - But: later versions of Java (from 5 onwards) turned out to have *unsound* type systems in various ways. Oops.
  - The interaction of sub-typing and inheritance turned out to make the early OO language Eiffel unsound. Also oops.[2]

[1] Syme. "Proving Java Type Soundness". 1997 [pdf]

[2] William R. Cook. A proposal for making Eiffel type-safe. The Computer Journal, 32(4):305–311, August 1989.

Categorizing formal methods

Are we trying to prove properties of an individual program? Or about *all* programs written in a particular language?

- An example of the first one is proving that a sorting function does what we want it to, or that a compiler implementation obeys some particular formal specification
- An example of the latter is proving results about the *type system* for a language, which lets us show that *all* programs in the language will have some sort of guarantees of good behaviour
  - e.g. Proving that well-typed Java programs cannot be subverted (assuming the JVM and compiler are implemented correctly) – it should be impossible to get a reference which doesn't point to a valid area of memory, for instance.

Aside – type systems

- We often don't think of type systems as being a "formal method", but some type systems are very expressive, and allow us to prove quite strong results about our programs
- We can use them to prove that (for instance) unsanitized user data never gets output to a web page

Type systems

- A type system many of us will have used in high school: consistency of SI units
- We can multiply and divide things which have different units (e.g. distance divided by time, or acceleration multiplied by time) . . .
  . . . but it makes no physical sense to *add* things with different units – we can't add seconds to metres – and the rules for consistency of SI units stop us from doing so, thus avoiding silly mistakes.
- In most programming languages: floating point numbers are used for all physical quantities – nothing to stop you adding a number representing seconds to one representing distance.
- Some languages (e.g. Fortress, F#) have dimensionality and unit checking built into the language –
  useful if coding something with a lot of physical quantities and want checks you haven't performed a physically nonsensical calculation.

Categorizing formal methods

Model-based vs proof-based approaches:

- We will see one example of a *proof* based approach, Hoare logic.
  - Your specification is some formula in some suitable logic
  - In Hoare logic, our specification is what we want the program to *do* – it's expressed as assertions (postconditions which should hold after the program executes, if the preconditions held)
  - You try and *prove* that the system (or some abstraction of it) satisfies the specification.
- Usually requires guidance and expertise from the user

Categorizing formal methods

Model-based approaches:

- Again, our specification is some sort of formula
- This time, our system description is some mathematical structure, a **model**, $\mathcal{M}$
- We check whether the model $\mathcal{M}$ **satisfies** the specification (i.e. has the properties we want)
- In many cases, this can be done automatically.