

VŠB TECHNICKÁ
UNIVERZITA
OSTRAVA

VSB TECHNICAL
UNIVERSITY
OF OSTRAVA



www.vsb.cz

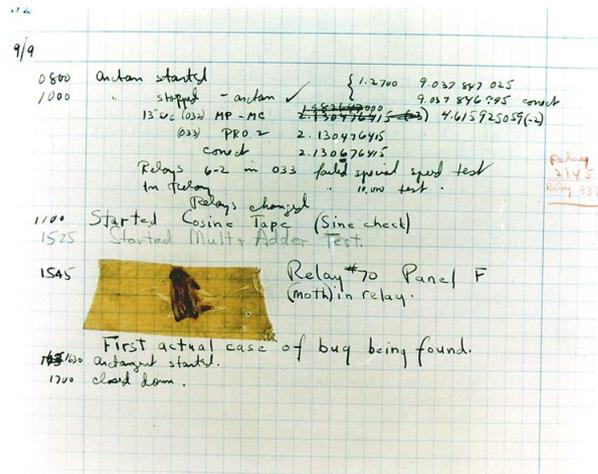
Testování a Softwarová Kvalita

David Ježek

VŠB – Technická Universita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Historie

- Můra zachycená v pomocném relé počítače Harvard Mark II na Harvardské univerzitě, 9 září 1945



05/04/22

TSK

3

The First "Computer Bug" Moth found trapped between points at Relay # 70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1945. The operators affixed the moth to the computer log, with the entry: "First actual case of bug being found". They put out the word that they had "debugged" the machine, thus introducing the term "debugging a computer program". In 1988, the log, with the moth still taped by the entry, was in the Naval Surface Warfare Center Computer Museum at Dahlgren, Virginia.

While [Grace Hopper](#) was working on the [Harvard Mark II](#) Computer at [Harvard University](#), her associates discovered a [moth](#) stuck in a [relay](#) and thereby impeding operation, whereupon she remarked that they were "debugging" the system. Though the term [computer bug](#) cannot be definitively attributed to Admiral Hopper, she did bring the term into popularity. The remains of the moth can be found in the group's log book at the [Smithsonian Institution's National Museum of American History](#) in [Washington, D.C.](#).^[1]

Stříbrná kulka

- V 70 letech minulého století se nastala „softwarová krize“
 - Údržba softwaru stála více jako jeho vývoj
- V 80 letech minulého století vzrostl obrat softwarového průmyslu na 300 miliard dolarů
- CASE (Computer Aided Software Engineering) nástroje nejsou všelék (stříbrné kulka) na softwarovou krizi

Kapitola 1 – Principy testování

- 1.1. Co je to Testování softwaru
- 1.2. Terminologie testování
- 1.3. Proč je testování nezbytné
- 1.4. Základní testovací proces
- 1.5. Opakované testování (Re-testing) a regresní testy
- 1.6. Očekávané výsledky
- 1.7. Prioritizace testů

1.1- Co je to Testování softwaru

Co si lidé obvykle myslí:

- Druhořadá práce
- Verifikace běžícího programu
- Pouze manuální testování
- Nudná rutinní úloha

Profesionální přístup:

- Respektovaná disciplína při vývoji softwaru
- Revize, inspekce kódu, statické analýzy, atd.
- Používání testovacích nástrojů, automatizace testů
- Analýza, návrh, programování testovacích skriptů, vyhodnocování výsledků, etc.

- Testování v minulosti bylo označováno za druhořadou práci mezi softwarovými profesionály. Ale poptávka po kvalitních testerech roste a chápání testování jako speciální disciplíny roste také. Nyní testování utváří nezbytnou aktivitu v softwarovém inženýrství a profesionální testeři jsou přinejmenším stejně respektováni jako profesionální vývojáři.
- Testování testování neznamená pouze verifikaci běžícího programu, zahrnuje rovněž testování požadavků, revize dokumentace, inspekce kódu, statické analýzy, atd.
- Testování neznamená pouze manuální testování, zahrnuje také nástroje pro testování požadavků, statickou analýzu, spouštění testů, zátěžové testování, dynamickou analýzu, ladění, správu testů, atd. Efektivní používání těchto nástrojů vyžaduje profesionála s analytickými, programátorskými a jinými dovednostmi.
- Testování neznamená nudné rutinní úlohy, ale náročnou tvůrčí práci, která zahrnuje analýzu požadavků, návrh testovacích případů a scénářů, programování testovacích skriptů, zhodnocení výsledků testů, atd.

The key difference to remember is that QA is interested in the process whereas testing and quality control are interested in the product. Having a testing component in your development process demonstrates a higher degree of quality (as in QA).

IEEE 12207 defines QA as “... a process for providing adequate assurance that the software products and processes in the

Co je to Testování software

- A) Testování je demonstrace, že v programu NEJSOU žádné vady?
- B) Testování demonstruje, že program provádí požadovanou funkcionalitu bezchybně?
- C) Testování je proces, který demonstruje, že program dělá to co má dělat?
- D) Testování je proces spouštění programu se záměrem nalézt vady?

Testování vs. Zajištění jakosti (Quality Assurance)

- The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of a component or system and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects. (ISTQB)
- Activities focused on providing confidence that quality requirements will be fulfilled. (ISTQB)

05/04/22

TSK

7

- Důkaz, že v programu NEJSOU žádné vady:
Pokud je našim cílem ukázat, že v programu nejsou žádné vady, budeme podvědomě ovlivnění tímto cílem, což znamená, že budeme mít sklon použít data, u kterých je malá pravděpodobnost, že způsobí selhání. Na druhou stranu pokud našim cílem bude ukázat, že program má vady, budeme volit data, která mají větší pravděpodobnost objevení vady. Tento novější přístup je větším přínosem pro program než ten bývalý.
- Testování ukazuje, že program provádí požadovanou funkcionalitu bezchybně:
- Testování je proces, který demonstruje, že program dělá to co má dělat:
Program který dělá to co má pořád obsahuje vady. Což znamená, že pokud je v programu zřejmá vada program nedělá to co má, ale vady jsou v programu i když program dělá to co má.

Na testování programů je lépe nahlížet jako na destruktivní proces snažící se najít vady (jejichž přítomnost se předpokládá) v programu. Úspěšný testovací případ je ten, který posune vývoj tím způsobem, že způsobí selhání programu. Jistě, nakonec chceme použít testování k zajištění jistého stupně přesvědčení, že program dělá to co dělat má a nedělá to co dělat nemá, ale výše zmíněný přístup je nejlepší k dosažení nejlepšího výsledku v počtu nalezených vad.

Co je to Testování software

- A) Testování je demonstrace, že v programu NEJSOU žádné vady?
- B) Testování demonstruje, že program provádí požadovanou funkcionalitu bezchybně?
- C) Testování je proces, který demonstruje, že program dělá to co má dělat?
- D) Testování je proces spouštění programu se záměrem nalézt vady?

*Testování je proces spouštění programu
se záměrem nalézt vadu.*

Glenford J. Myers

Testování vs. Zajištění jakosti (Quality Assurance)

- The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of a component or system and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects. (ISTQB)
- Activities focused on providing confidence that quality requirements will be fulfilled. (ISTQB)

05/04/22

TSK

8

- Důkaz, že v programu NEJSOU žádné vady:
Pokud je našim cílem ukázat, že v programu nejsou žádné vady, budeme podvědomě ovlivnění tímto cílem, což znamená, že budeme mít sklon použít data, u kterých je malá pravděpodobnost, že způsobí selhání. Na druhou stranu pokud našim cílem bude ukázat, že program má vady, budeme volit data, která mají větší pravděpodobnost objevení vady. Tento novější přístup je větším přínosem pro program než ten bývalý.
- Testování ukazuje, že program provádí požadovanou funkcionalitu bezchybně:
- Testování je proces, který demonstruje, že program dělá to co má dělat:
Program který dělá to co má pořád obsahuje vady. Což znamená, že pokud je v programu zřejmá vada program nedělá to co má, ale vady jsou v programu i když program dělá to co má.

Na testování programů je lépe nahlížet jako na destruktivní proces snažící se najít vady (jejichž přítomnost se předpokládá) v programu. Úspěšný testovací případ je ten, který posune vývoj tím způsobem, že způsobí selhání programu. Jistě, nakonec chceme použít testování k zajištění jistého stupně přesvědčení, že program dělá to co dělat má a nedělá to co dělat nemá, ale výše zmíněný přístup je nejlepší k dosažení nejlepšího výsledku v počtu nalezených vad.

Co lze testovat

- Od testování uživatelem zadaných požadavků na systém až po monitorování systému v provozu.
- Od testování funkcionality systému k ověřování dalších aspektů softwaru:
 - Dokumenty (specifikace)
 - Návrh (model)
 - Kód
 - Kód + platforma
 - Produkce, akceptace
 - Použití, podnikový proces

05/04/22

TSK

9

- From testing user requirements to monitoring the system in operation:

Testing is not done only once (e.g. with the first version of the product), but it is a continuous activity throughout product's entire lifecycle (from user requirements, through system design and implementation, to monitoring the system in operation and its maintenance). Testing is most effective in early phases of the development.

- From testing the functionality to checking all other aspects of software:

Testing is not focusing only to the system functionality but to all other attributes of the software:

- Documents (specifications)
- Design (model)
- Code
- Code+platform
- Production, acceptance
- Usage, business process

- Verification:

Its goal is to answer the question: "Have we done the system correctly?" Verification uses a previous development step (i.e. functional specification prior to coding) as the reference. A piece of code that fulfils its specification is verified.

- Validation:

Its goal is to check whether correct product has been built, i.e. whether it fulfils the customers needs. Thus, any step in the development process can be validated against user requirements.

The goal of testing may be verification or validation.

Skutečnost v softwarovém testování

- Testování může ukázat přítomnost vad, ale nemůže prokázat absenci vad. (Dijkstra) 10
- Všechny vady nemohou být nalezeny.
- Testování nevytváří kvalitní software ani neodstraňuje vady.
- Vývoj bez vad znamená – mimo jiné – testovat od samého začátku.
- Perfektní vývojový proces je nemožný, kromě teorie.
- Perfektní požadavky: nereálné

- Testing can show the presence of errors but cannot show the absence of errors:

There are still some errors never found in the software.

- All defects can not be found:

Even for simple programs/applications, the number of possible input combination or possible paths through the program is so large that all cannot be checked.

- Testing does not create quality software or remove defects:

It is the responsibility of development.

- Building without faults means – among other – testing very early:

Populární "argument" proti testování je: "Mněly by jsme tvořit bezchybně od samého začátku a ne hledat vady, když už je vše hotovo". Jistě. Ale "bezchybně od samého začátku" znamená mimo jiné, pořádně vše kontrolovat od samého začátku a po celý čas vývojového procesu. Inspekce specifikace požadavků a dokumentů návrhu může do jisté míry nahradit systémové testy a akceptační testy, ale neznamená to „vývoj bez testování“!

- Perfect development process is impossible, except in theory:

In practice, the way from concept to ready product cannot be guaranteed to be error-free (inaccurate requirements specifications, cognitive errors, organizational errors). Therefore the need to test the final product, regardless how perfect development is.

- Perfect requirements: cognitive impossibility:

Validation of requirements – are they what we really want? – is a kind of testing. But it is often impossible to define all requirements correctly in advance. Testing of the first version of a product is often a kind of additional requirements engineering: "is it what is really needed?"

Testovací terminologie

- “Neexistuje” všeobecně akceptovaná množina pojmů.
- ISEB vychází z Britského standardu BS 7925-1 a BS 7925-2
 - http://www.testingstandards.co.uk/bs_7925-1.htm
 - http://www.testingstandards.co.uk/bs_7925-2.htm
- ISO/IEC/IEEE 29119 Software Testing (1-5)
Nahrazuje:
 - IEEE 829 Test Documentation
 - IEEE 1008 Unit Testing
 - BS 7925-1 Vocabulary of Terms in Software Testing
 - BS 7925-2 Software Component Testing Standard
- ISTQB Glossary <https://www.istqb.org/downloads/glossary.html>
- Mnohé standardy v softwarovém testování poskytují pouze částečnou terminologii.

- Not generally accepted set of terms:
Different experts, tools vendors, companies, and countries use different terminologies (sometimes very exotic). These problems arise very obviously, e.g. after merge or acquisition of more companies.
- Definition of: ISEB
(Information Systems Examinations Board, U.K., www.bcs.org/bcs/products/qualifications/ise
b) A subsidiary of the British Computer Society (BCS) that provides industry-recognized professional training and qualifications for IS competency, ability and performance. ISEB works to develop ITIL and ISO 9000-based products and

Proč terminologie?

- Špatná komunikace
- Příklad: komponenta – modul – unit – základní(basic)-návrh – vývojové,... testování
- Neexistuje "dobrá" a "špatná" terminologie, pouze nedefinovaná a definovaná.
- Obtížnost definování procesu.
- Obtížnost popisu stavu testu a testování.

- Poor communication:

If every test manager puts different meaning to each term, he/she spends lot of time on defining what is what.

- Example: component – module – unit – basic – design – developer, ... testing:

Not only names differ but their precise meaning as well, which makes mapping difficult. Still worse if the same word means two completely different things, like "component" (either module, unit or "an independent component for component-based development").

- There is no "good" and "bad" terminology, only undefined and defined:

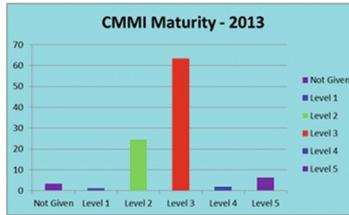
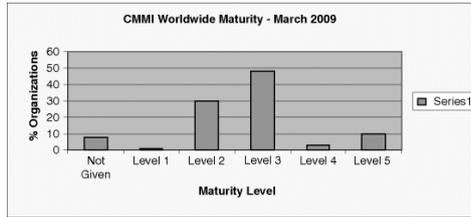
Some people readily argue about the "right" names for things but almost any defined, standardized and generally accepted terminology is almost always better than a

1.3. Proč je testování nezbytné

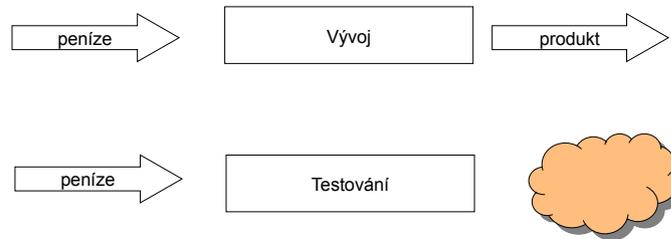
Ztráty softwarového testování

- Důsledkem softwarových vad přichází byznys v USA o ~\$60 miliard.
- 1/3 softwarových selhání se můžeme vyhnout lepším testovacím procesem.
Zdroj: National Institute of Standards and Technology (2002)
- Testovací proces ve většině softwarových společností je na nejnižší úrovni podle modelu CMMI (obvykle 1 nebo 2)
Zdroj: Software Engineering Institute CMU of Pittsburg
- Všechny současné modely vývoje softwaru obsahují testování jako jednu z nezbytných částí.

CMMI Maturity Statistics Worldwide



Testování v obchodních termínech



05/04/22

TSK

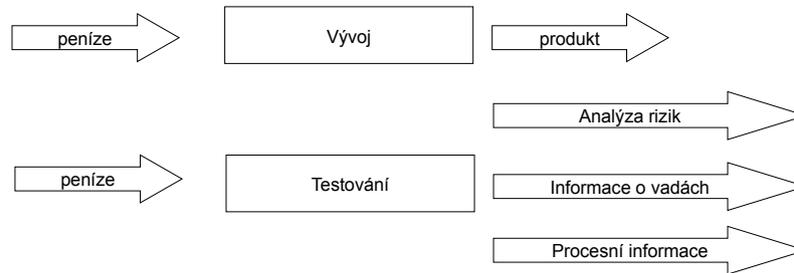
15

- In *Development*, “*money*” (investment) result in *products*, that can be sold any yield revenue.
- In *Testing*, it’s unclear from business perspective how “*money*” (investments) result in anything of value.
- *Testing* produces three kind of outputs:
 - *risk information*: probability that the product will fail in operation – this information is necessary for better delivery decisions
 - *bug information*: input to development to enable them to remove those bugs (and, possibly, to the customer to let them avoid the bugs)
 - *process information*: statistics and other metrics allow to evaluate processes and organization and identify faults in them

Unless there are *customers* for these outputs (managers willing to base their delivery decisions on test results, developers ready to fix defects found in testing, and process owners or projects managers ready to analyze and improve their processes), testing does not produce anything of value.

In other words, high-level testing in low-level environment does not add any immediate value, except as an agent of organizational change.

Testování v obchodních termínech



05/04/22

TSK

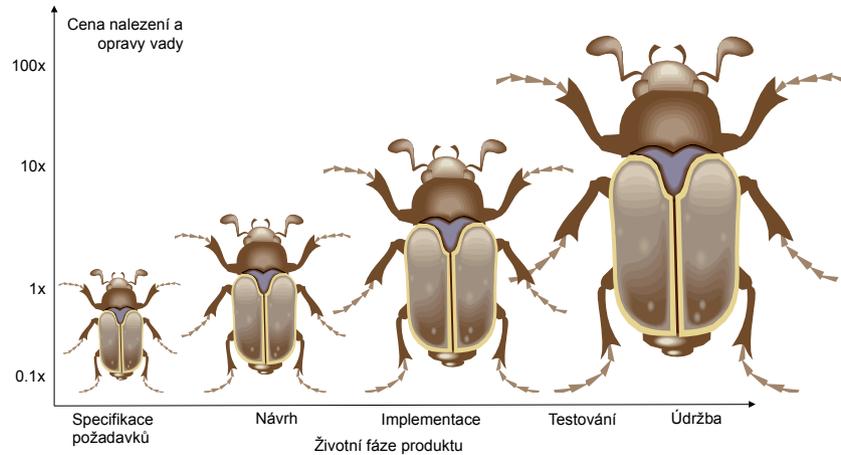
16

- In *Development*, “*money*” (investment) result in *products*, that can be sold any yield revenue.
- In *Testing*, it’s unclear from business perspective how “*money*” (investments) result in anything of value.
- *Testing* produces three kind of outputs:
 - *risk information*: probability that the product will fail in operation – this information is necessary for better delivery decisions
 - *bug information*: input to development to enable them to remove those bugs (and, possibly, to the customer to let them avoid the bugs)
 - *process information*: statistics and other metrics allow to evaluate processes and organization and identify faults in them

Unless there are *customers* for these outputs (managers willing to base their delivery decisions on test results, developers ready to fix defects found in testing, and process owners or projects managers ready to analyze and improve their processes), testing does not produce anything of value.

In other words, high-level testing in low-level environment does not add any immediate value, except as an agent of organizational change.

Testování snižuje cenu



05/04/22

TSK

17

The cost of discovering, localizing, correcting and removing a fault is often hundreds or thousands of times higher in ready product than it is in the early states of development. The cost of re-testing, regression testing, and updating or replacing the faulty software multiply very quickly after release (especially in mass production).

Test is most effective in early phases:

Contrary to traditional approach, test need not wait until development is ready. Test – reviews, inspections and other verification techniques for documentation and models - is actually the most effective in very early stages of any development process.

Testování a Kvalita

- Test měří a podporuje kvalitu.
- Testování je část zajištění jakosti (Quality Assurance)
- Mnoho měřítek jakosti/kvalit:
 - Funkční kvalita (tradiční)
 - Nefunkční kvalita (např. Výkon)
 - Kvalitativní atributy (spravovatelnost, znovupoužitelnost, testovatelnost, ...)
 - Použitelnost pro všechny účastníky (prodejce, maloobchodník, operátor, koncový uživatel, ...)
- Testovací techniky: nástroje pro měření kvality efektivně a výkonně.
- Správa testů: Jak vše zorganizovat.

05/04/22

TSK

18

- Test measures and supports quality:
Test has two goals: to *measure* and *visualize* (the level of quality becomes known) quality and to support achieving quality by identifying product and process faults.
- Testing is a part of Quality Assurance:
The goal is to achieve planned and known quality, not to test. If it could be achieved without testing, test would disappear. The goal for testing is therefore to minimize the volume of testing without compromising quality or to achieve as good (reliable) quality measurement as possible with given resources and time, not to “test as much as possible”.
- Many qualities:
 - *Functional quality*: the system does what the user required.
 - *Non-functional quality*: these aspects (e.g. performance) are growing in importance. Cannot be reliably engineered without extensive testing.
 - *Quality attributes*: there are other attributes (e.g. maintainability, reusability, testability, ...) that must be checked by testing too.
 - *Usability for all stakeholders*: “Usability” is not only important but multidimensional. What is comfortable for the operation may be uncomfortable for the end-user. Therefore growing need for measuring and quality assessment in this area.
- Test techniques: tools and methods to measure quality efficiently and effectively:
Test theory contains knowledge how to test *efficiently* (so that desired levels of quality and test reliability are achieved) and *effectively* (so that it is done as cheaply as possible).
- Test management: how to organize this:
Test management has much in common with general project and process management.

Složitost

- Software – a jeho prostředí – je příliš složitý na provedení úplných testů jeho chování.
- Software je obklopen.
- Software má lidské uživatele.
- Software je částí workflow systému organizace.

- Software is too complex to exhaustively test its behaviour:

Even for relatively simple programs/applications, the number of possible input combinations or possible paths through the program is so large that all cannot be checked. Then testing is necessary as a kind of art of predicting under uncertainty, choosing the few tests we can afford to run that give us best confidence in program's future correct behavior.

- Software environment is too complex to test it exhaustively:

A simple piece of code can be run on different PC-machines, OS (and their versions), with different printers, on different browsers. The number of combination easily becomes huge.

- Software can be embedded:

The testing products means testing SW, HW and “mechanics” around. Again, complexity. Again, methods required to make the best of this mess.

- Software has often human users:

For most applications, the behavior (and needs) of the users cannot be fully predicted by the engineering means only. Testing (acceptance, usability) helps to tackle this aspect.

- Software is part of the organization's workflow:

Engineering considerations are *not* the only important considerations for many software products. Any useful knowledge about product quality is a combination of engineering quality and the product's quality contribution during organizational or marketing usage.

Jak moc testovat?

- Obchodní rozhodnutí založené na akceptovatelné míře rizika
 - Kritérium splnění testu.
 - Kritérium prioritizace testů.
 - Rozhodovací strategie pro dodání
 - Test manažer předkládá míru kvality produktu
- Testování není nikdy připraveno
- Odpovědí je málokdy "víc testů", ale spíše "lepší testování", podívejme se na ukončovací kritéria:
 - Všechny testovací případy byli spuštěny
 - Všechny testovací případy se dokončili
 - Neexistuje nevyřešené (závažné) oznámení o incidentu.
 - Předefinované pokrytí bylo dosaženo.
 - Požadovaná spolehlivost (MTBF) byla dosažena.
 - Odhadnutý počet zbývajících vad je dostatečně nízký.

05/04/22

TSK

20

- This is a risk-based, business decision:
 - Test completion criteria – must be specified in advance in test plan
 - Test prioritization criteria - scales on which to compare test cases' relative importance (severity, urgency, probability, visibility, business criticality, etc.)
 - Decision strategy for the delivery – must be specified in advance (what shall happen if test completion criteria are not fulfilled)
 - Test manager presents products quality - he/she is responsible for the estimation and presentation of product quality but the business decision based on this data is made by responsible manager (project manager, project owner, product owner, etc.).
- Test is never ready:

As exhaustive testing is not possible, we can always test a little more, and there is always some justification for it in (the diminishing) probability that more faults will be found. Unless completion criteria are established and test cases prioritized, the probability of finding more faults cannot be reliably estimated
- The answer is seldom "more testing" but rather "better testing":

Testing must be based on the combination of completion criteria:

 - All test cases executed
 - All test cases passed
 - No unresolved (serious) incident reports
 - Pre-defined coverage achieved
 - Required reliability (MTBF - **Mean time between failures**) achieved
 - Estimated number of remaining faults low enough

Úplné testování

- Úplné testování je nemožné.
- Dokonce i podle teorii, je úplné testování zbytečně nákladné, protože nepřirazuje testům priority.
- Smluvní požadavky na testování.
- Svědomitý postup je důležitý z právního pohledu.

- Exhaustive testing is impossible:

Even for modest-sized applications with few inputs and outputs, the number of test cases quickly becomes huge.

- Contractual requirements on testing:

The contract between the vendor and the customer may contain clauses on the required amount of testing, acceptable reliability levels, or even on specific test techniques or test coverage measures.

- Non-negligent practice important from the legal point of view:

If you ever get sued by your customer, his or her lawyers will sure try the trick of accusing you of negligence because your testing was not “exhaustive”. As defense, the impossibility of exhaustive testing should be

Testování na základě rizik

- Testování hledá vady, které – pokud jsou odstraněny – sníží riziko selhání v provozu systému.
- Testování na základě rizik

- Testing finds faults, which decreases the risk of failure in operation
Testing can be based on any criteria, but the most important is the risk of failure in operation as this is the most obvious indication of quality software.
- Risk-based testing
 - The chosen amount and quality of testing shall be based on how much risk is acceptable
 - Test design (choosing *what* to test) shall be based on the involved risks
 - The order of testing shall be chosen according to the risks
- Error: the "mistake" (human, process or machine) that introduces a fault into software:
 - Human mistake: users forget a need. Requirements engineer misinterprets users' need. Designer makes a logical mistake. Programmer makes a coding mistake.
 - Process mistake: requirements not uniquely identifiable, no routines for coping with changing/new requirements, not enough time to perform design inspections, poor control over programmers' activities, poor motivation, ...
 - Machine mistake: incorrect compiler results, lost files, measurement instruments not precise enough...
- Fault: "bug" or "defect", a faulty piece of code or HW:
Wrong code or missing code, incorrect addressing logic in HW, insufficient bandwidth of a bus or a communication link.
- Failure: when faulty code is executed, it may lead to incorrect results (i.e. to failure):
A faulty piece of code calculates an incorrect result, which is given to the user. A faulty SW or HW "crashes" the system. A faulty system introduces longer delays than allowed during heavy load.
When a failure occurs during tests, the fault may be identified and corrected.
When a failure occurs in operation, it is a (small or large) catastrophe.

Základní pojmy

- **Chyba(Error):** "omyl" (lidská, procesní nebo strojová), která zanáší vady do softwaru.
- **Vada(Fault):** "bug" nebo "nedostatek (defect)", chybná část kódu nebo hardwaru.
- **Selhání(Failure):** pokud se provede vadný kód, může to vést k chybnému výsledku (to znamená k selhání).



05/04/22

TSK

23

- Testing finds faults, which decreases the risk of failure in operation
Testing can be based on any criteria, but the most important is the risk of failure in operation as this is the most obvious indication of quality software.
- Risk-based testing
 - The chosen amount and quality of testing shall be based on how much risk is acceptable
 - Test design (choosing *what* to test) shall be based on the involved risks
 - The order of testing shall be chosen according to the risks
- Error: the "mistake" (human, process or machine) that introduces a fault into software:
 - Human mistake: users forget a need. Requirements engineer misinterprets users' need. Designer makes a logical mistake. Programmer makes a coding mistake.
 - Process mistake: requirements not uniquely identifiable, no routines for coping with changing/new requirements, not enough time to perform design inspections, poor control over programmers' activities, poor motivation, ...
 - Machine mistake: incorrect compiler results, lost files, measurement instruments not precise enough...
- Fault: "bug" or "defect", a faulty piece of code or HW:
Wrong code or missing code, incorrect addressing logic in HW, insufficient bandwidth of a bus or a communication link.
- Failure: when faulty code is executed, it may lead to incorrect results (i.e. to failure):
A faulty piece of code calculates an incorrect result, which is given to the user. A faulty SW or HW "crashes" the system. A faulty system introduces longer delays than allowed during heavy load.
When a failure occurs during tests, the fault may be identified and corrected. When a failure occurs in operation, it is a (small or large) catastrophe.

Cena selhání

- Spolehlivost: je pravděpodobnost že nenastane selhání.
- Slavný "Arianne", American Airlines - rezervační systém a ostatní
- Kvalita života
- Bezpečné-kritické systémy
- Embedded (mobilní) systémy
- Specifikace požadavků na použitelnost v embedded systémech a webových aplikacích.

05/04/22

TSK

24

Famous: "Arianne", American Airlines booking system and more:

The financial cost can be shocking, many billions of dollars. As compared to the estimated cost of additional testing that would probably have discovered the fault (a few hundred thousand dollars).

- Arianne: two faults that did not cause failures in the old rocket model. An overflow exception caused by a different start trajectory and missing exception handling code. Could easily be found if tested.
- American Airlines: new booking system (1988) with complex mathematical algorithms for optimization of the numbers of business class and economy class passengers. It had a fault, which resulted in approximately 4 passengers fewer on every flight. Billions of lost revenue after a few months' operation were the first indication there was a fault at all!

Quality of life:

As anyone using a PC or a buggy VCR software realizes, failures need not be catastrophes to sharply reduce the joy of living.

Safety-critical systems:

More and more safety-critical systems contain software – the necessity of high safety and reliability grows. The cost of failure is injury or human life (railway, aircraft, medical systems). For many safety-critical systems the important attribute is usability (low usability can cause "operator mistake" or "human factor" in an accident, coming usually from confusing or unusable information, especially in stress situations).

Embedded systems

Embedded systems (whether safety-critical or not), require high-quality software, because of the difficulty (or impossibility) of updates. Remember the cost of software errors in some mobile phones.

Usability requirements for embedded systems and Web applications:

Embedded systems and Web applications are mass consumer products, where customers require easy usage. Failure to provide it results in lost revenues or market shares, which is a novel experience for software industry, used more to putting requirements on customers than the other way round!

1.4. – QA Standards

ISO – International Organization for Standardization

- <http://www.iso.ch/>

ISO 9000 family

- ISO 9000:2000 – QMS Fundamentals and Vocabulary
- ISO 9001:2000 – QMS Requirements
- ISO 9004:2000 – QMS Guidelines for performance improvements
- TickIT / ISO 9000-3 & ISO 9001 is a guidance document which explains how ISO 9001 should be interpreted within the software industry

05/04/22

TSK

25

- ISO International Organization for Standardization
 - ISO is the world's leading developer of International Standards.
- ISO standards specify the requirements for state-of-the-art products, services, processes, materials and systems, and for good conformity assessment, managerial and organizational practice.
- ISO standards are designed to be implemented worldwide.
- The ISO 9000 family is primarily concerned with "quality management". This means what the organization does to fulfil:
 - the customer's quality requirements
 - applicable regulatory requirements, while aiming to
 - enhance customer satisfaction
 - achieve continual improvement of its performance in pursuit of these objectives.
- ISO 9000:2000 – QMS Fundamentals and Vocabulary
It describes fundamentals of quality management systems (QMS), which form the subject of the ISO 9000 family, and defines related terms (quality, product, process, process approach, effectiveness, etc.).
- ISO 9001:2000 – QMS Requirements
It specifies requirements for a quality management system where an organization needs to demonstrate its ability to consistently provide product that meets customer and applicable regulatory requirements, and aims to enhance customer satisfaction through the effective application of the system, including processes for continual improvement of the system and the assurance of conformity to customer and applicable regulatory requirements.
- ISO 9004:2000 – QMS Guidelines for performance improvements
This International Standard provides guidelines beyond the requirements given in ISO 9001 in order to consider both the effectiveness and efficiency of a quality management system, and consequently the potential for improvement of the performance of an organization.
- TickIT / ISO 9000-3 & ISO 9001 is a guidance document which explains how ISO 9001 should be interpreted within the software industry
The TickIT program was created by the government of the United Kingdom to provide a method for registering software development systems based on the ISO 9000-3 standard. The scheme was jointly developed by the United Kingdom Department of Trade and Industry (DTI) and the British Computer Society (BSC). The ISO 9000-3 Standard is named "Quality management and quality assurance standards - Part 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software." It was originally written as a "guidance standard." The TickIT program turns it into a compliance standard. It is possible to be certified to ISO 9001 without gaining TickIT, but **ISO 9001 with TickIT is a mark of excellence!**

CMMI

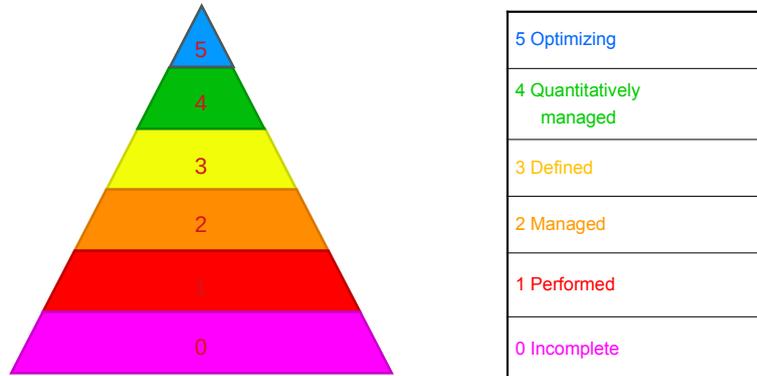
- CMM (Capability Maturity Model) is a reference **model** of mature practices in a specified discipline, used to improve and appraise a group's capability to perform that discipline (e.g. software development practices).
- CMMI is the **integrated process model (CMM)**

CMM + Integration = CMMI

- A CMMI model provides a structured view of process improvement across an organization.
- <http://www.sei.cmu.edu/cmmi>

- CMM (Capability Maturity Model) is a reference **model** of mature practices in a specified discipline
 - Developed by SEI – Software Engineering Institute, established in 1984 as Federally Funded Research and Development Center FFRDC
 - Awarded to Carnegie Mellon University
 - Based on the concepts of Crosby, Deming, Juran, Humphrey, etc.
 - 1986: Start at request of Department of Defense (DoD)
 - 1987: Framework and Maturity Questionnaire
 - 1991: CMM V1.0
 - 1993: CMM V1.1
 - CMMs differ by
 - discipline (e.g. software engineering, system engineering)
 - structure (e.g. staged, continuous)
 - definition of maturity (i.e. process improvement path)
- The CMM builds upon a set of processes and practices that have been developed in collaboration with a broad selection of practitioners.
- CMMI is the **integrated process model**
 - Developed by SEI – Software Engineering Institute at Carnegie Mellon University
 - 2000: CMMI V1.0 – first release together with associated appraisal and training materials
 - 2002: CMMI V1.1 - saw the release of *CMMI*
- A CMMI model provides a structured view of process improvement across an organization:
 - integrate traditionally separate organizations
 - set process improvement goals and priorities
 - provide guidance for quality processes
 - provide a yardstick for appraising current practices

The Capability CMMI Levels



05/04/22

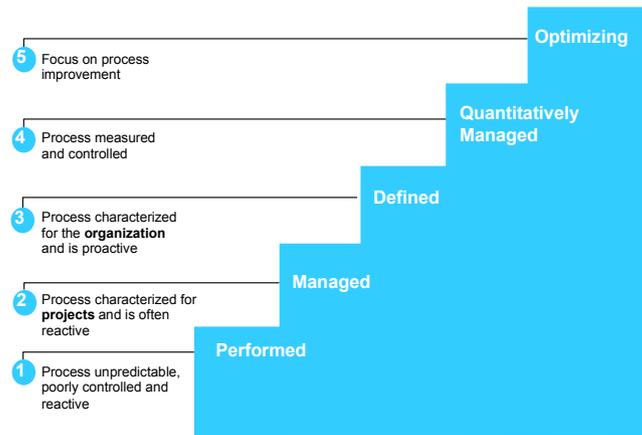
TSK

31

Capability Levels

- A capability level is a well-defined evolutionary plateau describing the organization's capability relative to a particular process area.
- There are six capability levels.
- Each level is a layer in the foundation for continuous process improvement.
- Thus, capability levels are cumulative (i.e., a higher capability level includes the attributes of the lower levels).
- <http://www.tutorialspoint.com/cmmi/cmmi-capability-levels.htm>

The Five CMMI Maturity Levels



05/04/22

TSK

34

Maturity Levels

- A maturity level is a well-defined evolutionary plateau of process improvement.
- There are five maturity levels.
- Each level is a layer in the foundation for continuous process improvement using a proven sequence of improvements, beginning with basic management practices and progressing through a predefined and proven path of successive levels.

Maturity Levels Should Not Be Skipped

- Each maturity level provides a necessary foundation for effective implementation of processes at the next level.

Definice testovacího procesu

- Plánování testů
- Specifikace testů
- Provádění testů
- Záznam testů a vyhodnocování
- Ukončovací kritéria

- Test process as part of development or production process

Test is a part of QA, and test process should not be defined separately, but should be seen in the context of overall development process.

- Large companies have own process definitions

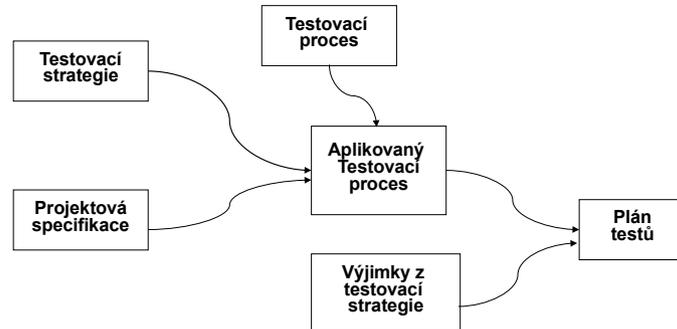
Most development and production companies have own test processes. There can be of course similarities (though used terminologies are often strikingly different), but nevertheless many thousands different test processes exist in industrial reality.

- “COTS” test process

COTS (Commercial Off The Shelf)

It is possible to buy a test processes. The most known vendor today is probably IBM Rational with its RUP – Rational Unified Process (test

Plánování testů



05/04/22

TSK

36

A company's Test Strategy together with its Test Process (defined in the organization) are adopted to the current project based on a Project Specification. This results into an Applied Test Process, i.e. an overall vision “how we will test this time”. This vision is the implemented (described in detail) in a Test Plan. Often, Test Plan is a document written in a natural language.

The process of creating a Test Plan is test planning. The is mostly done very early during project. Later, the processes of test *estimation*, *monitoring* and *control* may lead to changes in the test plan.

Cíle testového plánu

- Vysokoúrovňový testový plán a více detailnější testové plány.
- Spojený s projektovým plánem.
- Dodržuje plán zajištění jakosti (QA plan)
- Správa konfigurací, požadavky, správa incidentů

Even the best test plan will not work unless it is synchronized with other areas, project and technical.

- High-level test plan and more detailed test plans

Depending on project size and complexity, test plan can sensibly be divided into one high-level test plan and some detailed test plans. The division can follow test area or test level, or specific aspects of testing.

- Project plan

Test plan must be inspected for correctness and compliance with overall project plan. Sometimes (in small projects) the test plan is the part of a project plan.

- Follows the QA plan

Hopefully, the function and contents of a test

Specifikace testů

The complete documentation of the test design, test cases and test procedures for a specific test item. (ISTQB)

- Specifikace testů definuje co testovat.
- Specifikace testů je součástí „testware“ (testové případy, testovací skripty, testovací data).
- Základními stavebními kameny testů jsou testovací případy (test cases).
- Specifikace testů – instrukce – skript
- Specifikace testů – požadavky
- Specifikace testů – hodnotící zpráva

05/04/22

TSK

38

- Test specification defines *what* to test

Test specification are repositories of test cases. They should be free from organizational issues, which belong to the test plan(s).

- Test specification is part of testware

Testware – test cases, test scripts, test data, etc. is often under CM control (manages either by test tool or by a separate tool).

- Basic building blocks of test specifications are test cases

Test cases are generated when applying test design techniques. The shell be general and repeatable.

- Test specification – instruction – script

Test cases need not contain all detailed information on *how* to perform them. This information may be put into a separate description, sometimes called *test instructions* (this approach is not practical because of maintenance difficulties).

If test execution is automated, then the instructions for a test tool are called *test script (test program)*. Test script can replace test case (instructions).

- Test specification – requirements

It is desirable that for every test case, there is a link to the requirements behind it and for every requirement, there are links to all test cases that verify it. This is very hard to achieve and maintain without using test tools (test management tools, e.g. Test Manager).

- Test specification – reporting

The test specification must support logging and reporting during and after test execution, mainly through the *identification* of test cases and their steps. This can be easily automated by using test tools (test running tools, e.g. Robot)

Testovací případy (Test case)

- Jedinečné jméno/titulek
- Jedinečné ID
- Popis
- Podmínky/ předpoklady
- Akce (kroky)
- Očekávané výsledky

- Unique name/title

Short test case title enhances readability of the specification and test reports – descriptive unique name of the test case.

- Unique ID

Identification of the test case. All test cases should follow an identical, defined format. This ID must be permanent (adding or removing test cases shall not change ID) – cryptic unique identification of the test case.

- Description

Brief description explaining what functionality the case covers.

- Preconditions / prerequisites

Exact description of required system state prior the execution of the test case.

- Actions (steps)

1.4-Základní testovací proces - pojmy (ISTQB)

- **test case** - A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.
- **test case result** - The final verdict on the execution of a test and its outcomes, such as pass, fail, or error. The result of error is used for situations where it is not clear whether the problem is in the test object.
- **test case specification** - A document specifying a set of test cases (objective, inputs, test actions, expected results, and execution preconditions) for a test item.
- **test specification** - A document that consists of a test design specification, test case specification and/or test procedure specification.
- **test script** - Commonly used to refer to a test procedure specification, especially an automated one.
- **test procedure specification** - A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script.
- **Test Design Specification** - A document specifying the test conditions (coverage items) for a test item, the detailed test approach and identifying the associated high-level test cases.

- Unique name/title

Short test case title enhances readability of the specification and test reports – descriptive unique name of the test case.

- Unique ID

Identification of the test case. All test cases should follow an identical, defined format. This ID must be permanent (adding or removing test cases shall not change ID) – cryptic unique identification of the test case.

- Description

Brief description explaining what functionality the case case covers.

- Preconditions / prerequisites

Exact description of required system state prior the execution of the test case.

- Actions (steps)

Provádění testů

- Manuálně
- Automatizovaně
- Testovací sekvence
- Testovací prostředí
- Testovací data

05/04/22

TSK

41

- Manual

Tester follows the description from the test case and performs step by step all specified actions. Prone to errors, boring (monkey testing) and time-consuming. It is recommended that the author of the test case performs it first.

- Automated

Test tool executes test case according to predefined instructions (test scripts or test program).

The automation scope can include any of / all of the following:

- Preparation (set-up to fulfill preconditions)
- Execution
- Result evaluation (comparing actual and expected results)
- Clean-up (putting system back into some known state)

- Test sequence

Sometimes it is not practical to execute each test case separately but it is better to put test cases into a sequence, e.g.:

- Insert new record
- Search existing record
- Modify existing record
- Delete existing record

- Test environment

There are more environments used for developing, testing and maintaining software applications (DEV – development, IT – functional and performance testing, QA – acceptance testing, PROD – production). Configuration files of test environment as part of testware are under CM control.

- Test data

Test data are various input and output files (for expected and actual results) that must be managed properly as part of testware. If test data are taken from the production, they must be degraded.

Záznam testů a vyhodnocování

- Záznam aktuálních výstupů a porovnání s očekávanými výstupy.
- Off-line vyhodnocování výsledků.
- Logování testů
- Zpráva o testech
- Záznam pokrytí testy
- Zpráva incidentů

- Recording actual outcomes and comparison against expected outcomes

Manual testing: If actual and expected outcomes match, then test case passed. If not, then test case failed, actual outcomes are recorded and incident (defect) is created and assigned to development.

Automated testing: Comparison is done automatically, everything is recorded and even incidents are created.

- Off-line test result evaluation

Sometimes the immediate result (pass/fail) is impossible (too fast execution to allow on-line evaluation by a lower analysis tool or the final result is available only after some other tests have been performed, etc.), so during test execution the results are gathered for the evaluation, which is done later.

- Test log

It is a log of "all" (*relevant* and *important*) what happened during test execution. This activity (log creation) is best to automate, as it is repetitive, boring and requires exactness. It is used for (1) off-line evaluation, (2) failure analysis and debugging and (3) for archiving and future reference.

- Test report

Is a summary of the results of all executed test cases. Must contain as well complete information on configuration and versions of test environment, testware and test object. Some test tools are capable to produce test report.

- Recording test coverage

If test cases are mapped to requirements, test coverage can be easily derived. When executing test cases, the results are projected into requirements with the information how much functionality was successfully tested.

- Incident management

Answer the following questions:

1. Was this really a failure?
2. What presumably caused this failure?
3. How to assign correction responsibility?

Incident must be repeatable – put enough information to the incident report to enable reproducing the incident by the developer who is fixing it.

Ukončování testů

- Kritéria ukončení testů musí být specifikována předem.
- Rozhodovací strategie pro rozhodnutí o vydání/distribuci musí být specifikována předem.
- Test manažer je zodpovědný za odhadnutí a předložení kvality produktu ne za rozhodnutí o vydání/distribuci.
 - Spuštěné TC
 - Úspěšné TC
 - Neúspěšné TC
 - Provedené TC
 - Intenzita selhání
 - Počet hlášených incidentů
 - Odhad kvality produktu
 - Spolehlivost tohoto odhadu
 - Plánovaný odhad kvality produktu.

05/04/22

TSK

43

- Test completion criteria must be specified in advance

In the test plan or similar document.

- Decision strategy for the release/delivery decision must be specified in advance

What shall happen if test completion criteria are not fulfilled, but deadlines are approaching and there is strong pressure to release? The strategy for making this decision should be defined in advance.

- Test manager is responsible for the estimation and presentation of the product quality, not for release/delivery decision

It is the responsibility of test manager to preset to project management accurate and up-to-date data on:

1. Number of percentage of run test cases
2. Number and percentage of passed tests
3. Number and percentage of failed tests
4. Trends in test execution (cumulative number of executed test cases)
5. Trends in failure intensity
6. Similar data on the number of incident reports, their status and trends
7. Estimation of product quality based on the data available
8. Reliability (level of significance) of this estimation
9. Projected estimation of product quality and test reliability for various scenarios

Kritéria ukončení

- Všechny testovací případy byli provedeny.
- Všechny testovací případy dopadly úspěšně.
- Neexistují nevyřešené incidenty.
- Neexistují nevyřešené závažné incidenty.
- Počet nalezených vad.
- Předdefinované pokrytí bylo dosaženo
 - Pokrytí kódu
 - Pokrytí funkcionality
 - Pokrytí požadavků
 - Pokud ne, musí se navrhnout více testovacích případů
- Požadovaná spolehlivost (MTBF) byla dosažena
- Odhadovaný počet vad je dostatečně nízký.

05/04/22

TSK

44

- All test cases executed

It is a sensible criterion, provided good quality, coverage and reliability of those tests (otherwise the less test cases we have, the easier to achieve completion).

- All test cases passed

The previous criterion plus additionally that there must be no failed tests – strong requirement not achievable in practice.

- No unresolved incident reports

It may be the same as the previous one but not necessarily: some incident reports may be postponed, rejected (e.g. caused by faults of test environment or testware, etc.).

- No unresolved serious incident reports

The previous criterion might be too strong – we can divide incident reports according to severity (e.g. 1 and 2 must be resolved).

- Number of faults found

Generally a useless criterion, as it is the estimated number of *remaining* faults that matter. The assumption is that many found faults means few remaining (this can be wrong – many found faults may mean many remaining).

- Pre-defined coverage achieved

Generally better than “all tests... no incidents...” family, because they address the issue of achieved test quality/reliability as well:

- Code coverage: there is a number of different code coverage measures that tell what proportion of tested code have been exercised by executed tests.
- Functional coverage: even very high code coverage does not guarantee that “all” (paths, user scenarios) has been tested. Therefore, it should be complemented by some kind of functional coverage.
- Requirements coverage: all code and all functions may have been tested, but in order to discover *missing* functionality, tests should cover all requirements.

- Required reliability (MTBF) achieved

This can only be calculated if statistical testing is used (MTBF – Mean Time Between Failures).

- Estimated number of remaining faults low enough

Based on the number and frequency of faults discovered so far during testing, an estimation of the number of remaining faults can be made.

Psychologické aspekty testování

- Hlavním cílem je najít vady (NE dokázat správnost)
- Testování je opakující se a detailní
- Testování přináší špatné zprávy
- Testování je “negativní a destruktivní”
- Kariéra testera
- Jak testovat příště?
- Nezávislé testování
- Metriky

05/04/22

TSK

45

- Primary intent to find faults (NOT to prove correctness)
- This comes from the definition of testing.
- Testing is repetitive and detailed
- Many activities which belong to testing are repetitive, requiring attention to detail. If not organized correctly, they may feel boring: not providing enough stimulation and variety.
- Testing brings bad news
- If the goal is to “succeed” and “deliver”, then successful testing brings *bad news* (more work to development, delayed releases). Besides, when testers identify *failures*, developers feel sometimes accused of having made *errors* (which should not be the case, but often is).
- Testing is “negative and destructive”
- You sure recognize that terrible feeling when you have prepared a wonderful dinner, but your spouse (husband, wife) fails to be impressed and notices instead a small stain on the tablecloth? Then you call him/her “negative and destructive”. This is what testers do for a living...
- Testing career
- In some companies, the top level for a tester’s advancement is Test Manager, whereas developers easily advance to Project Managers and into higher managerial levels. In reality, testers make better project managers and high-level managers because they are used to see the whole instead of the details, to adopt the user perspective, to think in terms of risk estimation, to think in terms of quality.
- How to test then?
 - Attitude: *successful* test causes failures (finding faults, not hiding them, shall be rewarded).
 - Attitude and maturity ladder:
 - Testing equals debugging
 - Testing is to prove that it works
 - Testing is to find where it does not work
 - Testing is to reduce the risk that it fails
 - Testing is to measure and estimate the risk that it fails
 - Testing is mental attitude
- Independent testing
- Outsiders make better testers than insiders. However, debugging is better done by people with intimate knowledge of the way system is build.
 - Test cases designed by the developer
 - Test cases designed by someone else (developer)
 - Test cases designed by another department
 - Test cases designed by another organization (outsourced testing)
 - Test cases not designed by a person (generated automatically by tools)
- Independence: not only test design, but planning, support, execution, evaluation
- Metrics
- By gathering and distributing metrics you help developers and management understand what test does.

1.5 – Opakované testování (Re-Testing)

- Znovu spuštění testovacího případu, který naposledy způsobil selhání
- Aktivováno dodáním opravy a změnou stavu incidentu.
- Spuštění nového testovacího případu pokud selhání bylo způsobeno náhodou.
- Testování na podobné nebo související vady.

- Re-running of the test case that caused failure previously

A test case has caused a test object to fail. The fault that (supposedly) caused this failure has been discovered and removed (fixed). The very same test case is executed on the new (corrected) version of the system to ensure that the fault has really been successfully fixed.

- Triggered by delivery and incident report status

Re-testing is normally done after the delivery of a fixed build and after the corresponding incident report has been put into a “fixed” (“corrected”, “re-test”, “put into build”, or similar name) status. Some kind of private re-test before formal release may be used as well.

- Running of a new test case if the fault was previously exposed by chance

When failure occurred by chance without any intentional test case being executed (e.g. by “smoke-test”, “sanity-check”, or “ad-hoc” testing), a new test case should be designed and added. Re-testing means then the execution of this new test case.

- Testing for similar or related faults

During re-testing, even test cases looking for *similar* faults may be executed. For example if a record deletion from a file caused failure, even other record deletion routines may be tested. Re-testing *related* faults is advisable too. For example if a record deletion method has been fixed, then other methods belonging to the same class can be re-tested after correcting the fault. This can be defined as “increased testing”, or new test design caused by faults already found.

Regresní testování

- Regrese z důvodu opravy vady (vedlejší účinky)
- Regrese z důvodu přidání nové funkcionality
- Regrese z důvodu nové platformy
- Regrese z důvodu nové konfigurace nebo po úpravách na žádost zákazníka
- Regrese a plánované dodání oprav

- Regression due to fixing the fault (side effects)

On average, according to empirical data, 10-25% of all fixes actually introduce new faults, sometimes in areas seemingly “far away” (temporally, functionally or structurally) from the original fault. To be able to discover the new faults, test cases seemingly “far away” from the fixed fault must be executed on fixed builds.

- Regression due to added new functionality

Adding new functionality may introduce faults into already existing functionality, or expose faults existing previously, but not found. Therefore, old functionality must be tested again for releases with new functionality.

- Regression due to new platform

A system that executes correctly in one environment may fail in another environment, either due to hidden faults or interface faults. Therefore, regression testing may be required even when not a single software instruction has been changed.

- Regression due to new configuration or after the customization

Sometimes called “configuration testing”. For example, a Java script depends on HW, operating system and browser of the client machine. Including different versions of them, the number of possible combinations is very large, requiring impossibility large amount of regression testing. Special strategies are available to tackle this.

- Regression and delivery planning

To decrease the amount of regression testing, a regression test suite may be run *once* on a release with *many* fault corrections and new functionality added. If an incremental methodology is used (e.g. RUP), then some increments (usually the latest ones) are focusing only on bug fixing which means that only re-testing and regression testing is needed. Regression testing is often used in *maintenance* when emergency fixes and “extra” functionality is introduced.

Regresní schéma

- Snížení frekvence dodávání
- Schéma Round-robin
- Rozšířená selekce pro testovací případy
- Statistický výběr testovacích případů
- Paralelní testování
- “Kouřové testy (Smoke-test)” pro nouzové opravy
- Optimalizace regresní sady:
 - Všeobecná (znalost systému, účel testovacích případů, pokrytí testy)
 - Historická (klesající regrese pro stabilní funkcionality)
 - Závislosti (Související funkcionality)
 - Koordinace mezi úrovněmi testovacích případů (vyhnutí se redundantní regrese na různých úrovních testování)

05/04/22

TSK

48

- Less frequent deliveries
If a regression test takes longer than the time between releases, decreasing the delivery frequency may be an option. If a number of fixes and functionality enhancements are delivered together, less frequent deliveries are possible without increasing the overall development time.
- Round-robin scheme
Example: A regression test suite has 300 test cases. It takes 1 day to execute 100 test cases. Releases come every day. Test cases no 1-100 are executed on release N, 101-200 on N+1, 201-300 on N+2, then again 1-100 on N+1, etc. Even if no release is fully regression tested, a relatively good measure of product quality is achieved.
- Additional selection of test cases
The regression test suite may be pruned to fit the available time. A selection of regression test cases may be used for most releases, while the complete test suite will be executed only before external releases, quality checkpoints, project milestones, etc.
- Statistical selection of test cases
Provided that the data on the probability distribution of user actions is available, test cases can be ordered according to their “importance”, i.e. the relative frequency of the user action that they test. In this way, even if the complete regression test suite is not executed, the reliability level can be estimated for releases.
- Parallel testing
By dividing test execution into a number of parallel tracks, that can execute independently and in parallel, calendar test execution time can be significantly decreased. This applies both to manual and to automatic testing. The cost is that multiple amount of test equipment and of testers are required.
- “Smoke-test” for emergency fixes
Emergency fix – exceptional release that fixes one fault (or low number of faults) or sometimes introduces a new (small in scope) functionality and its delivery is urgently required. As changes in the system are relatively small, complete testing is not needed.
“Smoke-test” or “sanity-check” means execution of a subset of the most important test cases from the regression suite with the goal to check if there is not major problem in the system after the change. Even in the emerging situation, some kind of “smoke-test” must be performed.
- Optimisation of regression suite
 - *General* – basic test techniques can help choose test cases for regression test suites effectively. Required level of test coverage can be used to estimate the needed amount of regression testing. Good system understanding is required to identify and remove repetitive or less important test cases. Redundant test cases can be removed.
 - *History* – regression test cases may become obsolete with time. Stable functionality where faults are no longer discovered during regression testing, need not be tested as extensively as new, unstable functionality, or as a system area with a history of many faults.
 - *Dependencies* – provided a well-designed system with clear-cut dependencies and interfaces, it is possible to minimize the amount of regression for areas that are not related and not connected to the area, where recent changes have occurred.
 - *Test-level co-ordination* – savings in regression test time can often be achieved by coordinating tests run on different levels, to avoid repetition.

Regrese a automatizace

- Sady regresních testů spadají pod management konfigurací (CM)
- Tvorba a sledování incidentů pro testovací případy
- Automatizace se nejvíce vyplatí u regresního testování
- Automatizace testů řízená regresí
- Inkrementální vývoj

- Regression test suites under CM control

All test cases shall be archived and under version control to be able to return back to already not used test cases. Regression test cases are changing from release to release. This applies even more to automated regression testing which increases the amount of testware: test scripts, test programs, test data, test configurations, etc.

- Incident tracking for test cases

Test cases (especially test scripts, test programs, test data) can be faulty or changed for other reasons (e.g. effectiveness). These changes should be controlled and traceable like any software changes. The development and maintenance of testware should be handled like development and maintenance of any other software, i.e. planned, designed, under version management, etc.

- Automation pays best in regression

When test automation is considered, it shall be first of all applied to regression testing. The strategy for regression testing must therefore be known before the automation strategy is developed. Large amount of regression requires automation (the automation is effective starting from number of releases > 3). Performance testing cannot be done without tools (load generation, monitoring, performance measurement, etc.). These tools and test cases may therefore be candidates to be included in regression testing.

- Regression-drive test automation

Introducing test automation into projects must be planned according to the needs of the regression test strategy.

- Incremental development

New development methods ("incremental development", "daily build", "Rapid Application Development", RUP, etc.) become increasingly popular. They are characterized by frequent deliveries, incremental functionality growth, and fast feedback from test to development. Therefore, they require heavy regression testing, which makes both test automation and other techniques for regression optimization especially important.

1.6 – Očekávané výsledky

Proč jsou nezbytné?

- Test = měření kvality = porovnání aktuálního výsledku s očekávaným výsledkem
- A co měření výkonu?
- Výsledky(Results) = Outcomes; výsledek ≠ výstup
- Definice testovacího případu: podmínky – vstupy – očekávané výsledky
- Výsledky jsou částí testware – Management konfigurací (CM)

- Test = measuring quality = comparing actual outcome with expected outcome
Test is verifying whether something is correct or not – means by definition comparing two values: actual and expected. Random testing is (1) normally not really testing at all (2) or testing actual results against our vague and unspecified outcome expectations.
- What about performance measurement?
Performance measurement = benchmarking.
Performance requirements are notoriously vague or absent, but performance testing is thriving.
Explanation? It is then either testing against informal, unspecified “random requirements” or a kind of requirement engineering (trying to find out what the requirements should be) by running ready product.
- Results = outcomes; outcomes ≠ outputs
Application outputs can be test case outcomes, but not all test cases outcomes are outputs – performance levels, state transitions, data modifications are possible test case outcomes which are not application outputs. In order to evaluate them, test environment must provide access to them: through special test outputs, debug tools, etc.
- Test case definition: preconditions – inputs – expected outcomes
When expected test result/outcome is missing, then it is NOT a test case specification at all. Unspecified or insufficiently specified expected outcomes make some failures harder to discover.
- Results are part of testware – CM control
Often, the expected outcome is a data file. Unless it can be incorporated in a test specification, it will require to be under separate CM control. Changing the expected outcome file will have the same effect as directly changing the test specification – a common baseline for them will therefore be required.

Typy výsledků

- Výstup
- Změna stavu
- Změna dat
- Jednoduché a složené výsledky
- “Dlouhotrvající” výsledky
- Kvalitativní atributy (čas, velikost, etc.)
- Netestovatelné?
- Vedlejší účinky

- Outputs

They are most easily observable, therefore often utilized as outcomes/results. Outputs have very many forms: displayed or changed GUI objects, sent messages or signals, printouts, sounds, movements.

- State transitions

Does the system perform correct state transition for a given set of inputs? Outputs following transitions are often used to judge, but the new state is the expected outcome.

- Data changes

Has data changed correctly?

- Simple and compound results

Results may be simple (“Error message appears”) or compound (“new record put into database, index updated, display adjusted, message sent...”).

- “Long-time” results

For example, testing for long-time stability: system still works correctly after a week.

- Quality attributes (time, size, etc.)

Most non-functional requirements are of this kind.

- Non-testable?

- (1) Possibly valid requirements, but formulated in a non-testable way, e.g. “sufficient throughput to handle typical traffic”.
- (2) Valid, measurable requirements, which cannot be measured due to technical constraints.

- Side-effects

Implicitly, every test case has an invisible clause in expected outcome definition “the program does this... and nothing incorrect happens”. “Nothing incorrect” is easily implied, but impossible to verify.

Zdroje výsledků

Nalezení nebo spočítání správných očekávaných výsledků je mnohdy mnohem těžší než by se mohlo zdát. Je to hlavní úloha při přípravě testovacích případů.

- Požadavky
- Oracle
- Specifikace
- Existující systém
- Jiné obdobné systémy
- Standardy
- NE kód

05/04/22

TSK

52

- Requirements
Sufficiently detailed requirement specifications can be used directly as the source of expected test results. Most often however, requirements do not have sufficient quality.
- Oracle
According to BS 7925-1 it is “a mechanism to produce the predicted outcomes to compare with the actual outcomes of the software under test”; often a program, another similar application, etc.
- Specifications
Specifications other than requirement specification (e.g. design specification, use case specification, interface specification, function specification) are generally a good source of expected outcomes – verification means testing whether system works “according to specification”.
- Existing systems
Previous, verified versions of the same system can be used as oracle for getting correct expected results.
- Other similar systems
Any other software – commercial or not – that has already been sufficiently verified and implements part of the functionality of the tested system, often makes a good oracle.
- Standards
Many standards, e.g. in telecommunications, contain detailed specifications that can be used as expected test results. A good example of a test case suite built entirely around standard specification is Sun’s test suite for verification whether a JVM (Java Virtual Machine) conforms to Sun’s Java standard.
- NOT code
(nor the same specification if specification is the test object), Because anything compared to itself (the same source of expected and actual outcomes) will always give “correct” results.

Obtížnost porovnání

- GUI
- Komplexní výsledky
- Absence vedlejších efektů
- Časový aspekt
- Neobvyklé výstupy (multimedia)
- Problém s reálným časem (Real-time) a dlouhým během (Long-time)
- Komplexní výpočty
- Inteligentní a “fuzzy” porovnání

05/04/22

TSK

53

- GUI
Notoriously difficult expected results. Prone to frequent changes, complex, often asynchronous. If treated on pixel level, often useless, require some kind of object approach. Most tools existing today do not cope well with moving or scrolling components.
- Complex outcomes
Actually, GUI outputs are one of them. Comparison may be difficult simple because the results are large and complex.
- Absence of side-effects
For most test cases, there are infinitely many possible outcomes that must not happen. For a test case “press key” with expected outcome “text <<key pressed>> appears” there are innumerable things that are expected NOT to happen: program does not crash, database is not deleted, no – say – blinking green triangle appears in the middle of the screen... etc. Verifying this is impossible, on the other hand some degree of observant caution is necessary.
- Timing aspects
Outcomes that either occur very quickly or last very short time, or are asynchronous, or occur after undefined delay may all be hard to verify correctly.
- Unusual outputs (multimedia)
Video sequences, complex graphics, sounds, smells, etc. are very hard to test.
- Real-time and long-time difficulties
(it is a sub-set of “absence of side effects”)
For real-time, multithread applications there may exist hidden faults that only cause failure when certain rare timing conditions are fulfilled. Such failures are not easily repeatable. During long-time execution a gradual “decay” of software may occur (stability testing aims at those problems). Typical example of such problems are memory-leaks.
- Complex calculations
Their results are hard to verify, may only “look right”. AA booking system fault 1988.
- Intelligent and “fuzzy” comparisons
Whenever correct result is not fully deterministic or analogue rather than discrete, it is difficult to verify.

1.7-Prioritizace testů

Proč prioritizovat testovací případy?

- Určit důležitost a pořadí (v čase)
- “Nikdy není dost času”
- Testování přichází až na konec a je postiženo zpožděním všech ostatních.
- Je těžké udělat dobrou prioritizaci (vícenásobná kritéria s různou vahou).

- Decide importance and order (in time)

To prioritize test cases means to measure their importance on an ordinal scale, then plan test execution accordingly (typically, in descending order of importance, i.e. more important cases before less important).

- “There is never enough time”

Dedicated testers easily become paranoid – they suspect faults everywhere and want to verify every tiny detail. To balance this desire with business reality, we must choose what is most important to test, i.e. prioritize.

- Testing comes last and suffers for all other delays

The day for customer delivery is often holy, but development is nevertheless delayed.

Planned test time is cut as a result, often with

Prioritní kritéria

- Vážnost (selhání)
- Priorita (urgentnost)
- Pravděpodobnost
- Viditelnost
- Priorita požadavků
- Zpětná vazba na/od vývojářů
- Obtížnost (testu)
- Co zákazníci chtějí
- Náchyllost ke změně
- Náchyllost k selhání
- Kritické z hlediska byznysu
- Složitost (testovaného objektu)
- Obtížnost opravy

This is a tentative list of possible prioritization criteria (scales on which to compare test cases' relative importance). This list is not ordered (i.e. it gives no clue to which criteria are more important). The criteria are not independent nor exclusive. For operational usage, they must be defined more in details. Put them into columns and mark each test case with the level of importance:

- H – high
- M – medium
- L – low
- Severity (failure): the consequences of failure (in operation): 1 – fatal, 2 – serious, 3 – disturbing, 4 – tolerable, 5 – minor
- Priority (urgency): how important it is to test this particular function as soon as possible: 1 – immediately, 2 – high priority, 3 – normal queue, 4 – low priority
- Probability: the (estimated) probability of the existence of faults and failure in operation
- Visibility: if a failure occurs, how visible it is? (it relates to “severity”)
- Requirement priorities: if requirements are prioritized, the same order shall apply to test cases
- Feedback to/from development: do the developers need test results to proceed? (similar to “priority”). Do the developers know a specific tricky area or function?
- Difficulty (test): is this test case difficult to do (resource- and time-consuming?)
- What the customer wants: ask the customer what he prefers (it relates to “requirements priorities”)
- Change proneness: does this function change often?
- Error proneness: is it a new, badly designed, or well-knowns “stinker” feature?
- Business criticality: related to “severity” and “what the customer wants”
- Complexity (test object): related to “error proneness”
- Difficult to correct: a fault known to be difficult to correct, may be given lower priority (provided severity is sufficiently low)

Metody prioritizace

- Náhodná (ponechá se pořadí ze specifikace)
- Expertův “sedmý smysl”
- Na základě historických zkušeností s obdobným projektem, produktem nebo zákazníkem.
- Testování podle statistiky používání (Statistical Usage Testing)
- Dostupnost: dodaných kódů, nástrojů, prostředí, doménových expertů...
- Tradiční analýza rizik (SWOT, ...)
- Multidimenzionální analýza rizik (Multi-criteria decision analysis)
 - analytic hierarchy process (AHP)

05/04/22

TSK

56

- Random (the order specs happen)
No method at all, but “the order test specs happen” may actually mirror both the “priority” and “business criticality” as well as “requirements prioritization” – the not so bad.
- Experts’ “gut feeling”
Experts with testing, technical and domain (application) knowledge do the prioritization. Experts are good to have, but their “gut feeling” may often be misleading, unless structured methods (see below) are followed.
- Based on history with similar projects, products or customers
Documented data on previous fault history, priority, severity, etc. is used to prioritize test cases for current project/product according to some chosen criterion (or a chosen

Kapitola 2 – Testování během životního cyklu

- 2.1. – Modely pro testování
- 2.2. – Ekonomika testování
- 2.3. – Plánování testů
- 2.4. – Testování komponent
- 2.5. – Testování integrace komponent
- 2.6. – Systémové testy (funkční)
- 2.7. – Systémové testy (nefunkční)
- 2.8. – Testování integrace systémů
- 2.9. – Akceptační testování
- 2.10. – Obslužné testování (Maintenance testing)

2.1 – Modely pro testování

Verifikace, Validace a Testování

- **Verifikace:** Proces vyhodnocení systému nebo komponenty, který určí zda produkt dané vývojové etapy splňuje podmínky určené na začátku této etapy – vytváříme systém správně
- **Validace:** Rozhodnutí o správnosti produktu softwarového vývoje s ohledem na potřeby a požadavky zákazníka – vytváříme správný systém
- **Testování:** Proces vykonávání/spouštění softwaru pro ověření zda splňuje specifikované požadavky a detekci chyb.

Testing is not only test execution. Static analysis can be performed before the code has been written. Writing and designing test cases is also part of testing. Reviews of requirement specifications and models, and of any other documents, belong to testing as well.

IEEE standards

3.1.36 verification:

- (A) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
- (B) The process of providing objective evidence that the software and its associated products conform to requirements (e.g., for correctness, completeness, consistency, accuracy) for all life cycle activities during each life cycle process (acquisition, supply, development, operation, and maintenance); satisfy standards, practices, and conventions during life cycle processes; and successfully complete each life cycle activity and satisfy all the criteria for initiating succeeding life cycle activities (e.g., building the software correctly).

3.1.35 Validation:

- (A) The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.
- (B) The process of providing evidence that the software and its associated products satisfy system requirements allocated to software at the end of each life cycle activity, solve the right problem (e.g., correctly model physical laws, implement business rules, use the proper system assumptions), and satisfy intended use and user needs.
- NOTE—For (A), see IEEE Std 610.12-1990 [B3].
- NOTE—For subdefinition (A), see IEEE Std 610.12-1990 [B3].

BS 7925-1

- **verification:** The process of evaluating a system or component to determine whether the products of the given development phase satisfy the conditions imposed at the start of that phase. [IEEE]
- **validation:** Determination of the correctness of the products of software development with respect to the user needs and requirements. [IEEE]

ISTQB Glossary

Verification Ref: ISO 9000

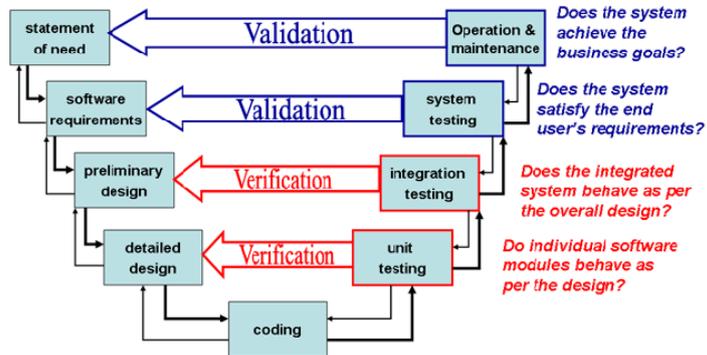
- Confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled.

Validation Ref: ISO 9000

- Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled.

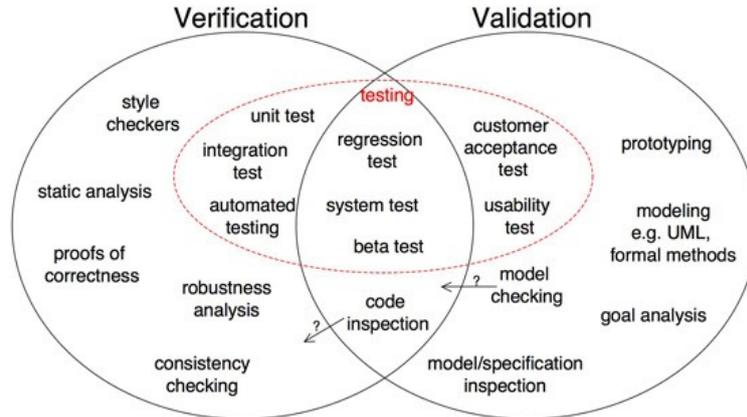
V&V - Where is truth?

Dynamic Testing



http://www.chambers.com.au/glossary/verification_validation.php

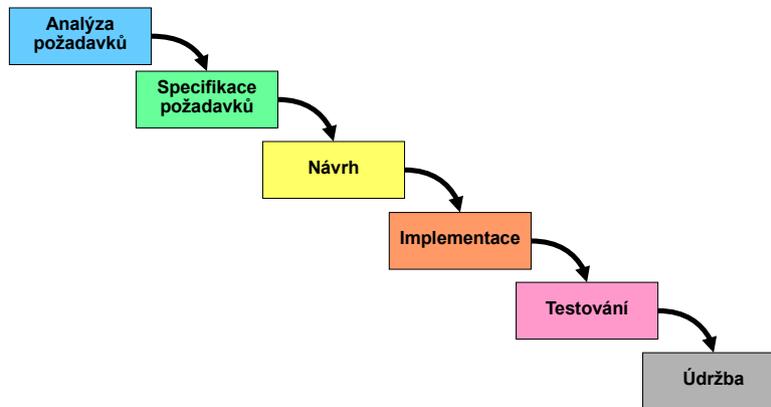
V&V - Where is truth?



<http://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation/>

2.1-Modely pro testování

Vodopádový model



05/04/22

TSK

64

Requirements Analysis

During the requirements analysis phase, basic market research is performed and potential customer requirements are identified, evaluated, and refined. The result of this phase of the process is usually a marketing requirement or product concept specification. Requirements in the concept specification are usually stated in the customer's language.

Requirements Definition

Requirements in the concept specification are reviewed and analysed by software engineers in order to more fully develop and refine the requirements contained in the concept specification. Requirements from the concept specification must be restated in the software developer's language – the software requirements specification.

Design

Once the SRS is developed, software engineers should have a complete description of the requirements the software must implement. This enables software engineers to begin the design phase. It is during this phase that the overall software architecture is defined and the high-level and detailed design work is performed. This work is documented in the software design description.

Coding

The information contained in the SDD should be sufficient to begin to the coding phase. During this phase, the design is transformed or implemented in code. If the SDD is complete, the coding phase proceeds smoothly, since all of the information needed by software engineers is contained in the SDD.

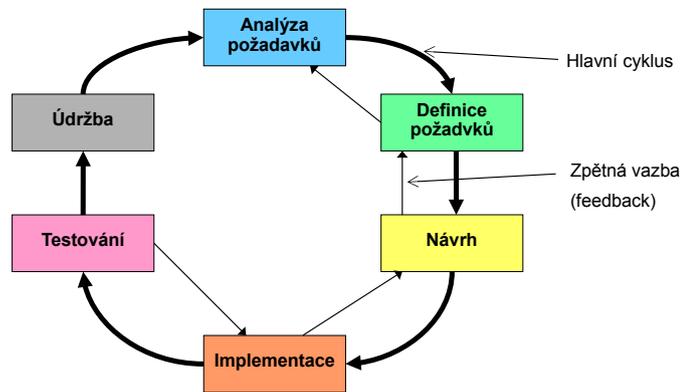
Testing

According to the waterfall model, the testing phase begins when the coding phase is completed. Tests are developed based on information contained in the SRS and the SDD already in the coding phase. These tests determine if the software meets defined requirements. A software validation test plan defines the overall validation testing process. Individual test procedures (test cases, test scripts, test programs) are developed based on a logical breakdown of requirements. The results of the testing activities are usually documented in a software validation test report. Following the successful completion of software validation testing, the product may be shipped to customers.

Maintenance

Once the product is being shipped, the maintenance phase begins. This phase lasts until the support for the product is discontinued. Many of the same activities performed during the development phases are also performed during the maintenance phase.

Vývojový životní cyklus iterativní



05/04/22

TSK

65

Requirements Analysis

During the requirements analysis phase, basic market research is performed and potential customer requirements are identified, evaluated, and refined. The result of this phase of the process is usually a marketing requirement or product concept specification. Requirements in the concept specification are usually stated in the customer's language.

Requirements Definition

Requirements in the concept specification are reviewed and analysed by software engineers in order to more fully develop and refine the requirements contained in the concept specification. Requirements from the concept specification must be restated in the software developer's language – the software requirements specification.

Design

Once the SRS is developed, software engineers should have a complete description of the requirements the software must implement. This enables software engineers to begin the design phase. It is during this phase that the overall software architecture is defined and the high-level and detailed design work is performed. This work is documented in the software design description.

Coding

The information contained in the SDD should be sufficient to begin to the coding phase. During this phase, the design is transformed or implemented in code. If the SDD is complete, the coding phase proceeds smoothly, since all of the information needed by software engineers is contained in the SDD.

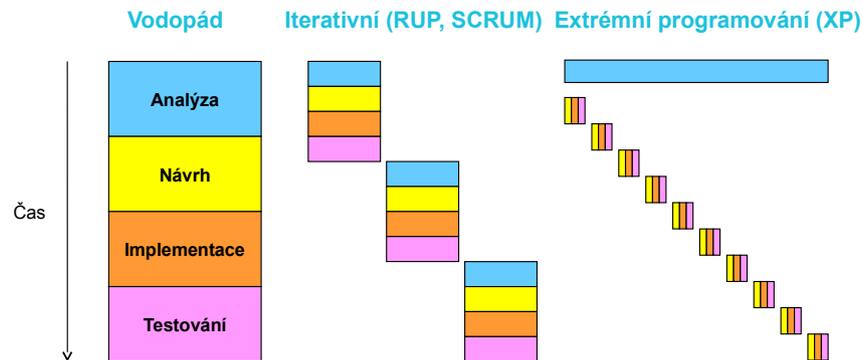
Testing

According to the waterfall model, the testing phase begins when the coding phase is completed. Tests are developed based on information contained in the SRS and the SDD already in the coding phase. These tests determine if the software meets defined requirements. A software validation test plan defines the overall validation testing process. Individual test procedures (test cases, test scripts, test programs) are developed based on a logical breakdown of requirements. The results of the testing activities are usually documented in a software validation test report. Following the successful completion of software validation testing, the product may be shipped to customers.

Maintenance

Once the product is being shipped, the maintenance phase begins. This phase lasts until the support for the product is discontinued. Many of the same activities performed during the development phases are also performed during the maintenance phase.

Zjednodušený náhled na modely



05/04/22

TSK

66

- Waterfall

There is no ideal model. Waterfall model is the right one in ideal world.

Analysis - I understand everything

Design - I design perfect solution with complete and right knowledge of customer and target platform

Coding - Design is coded without bugs

Testing - Well, why the hell test ideal system? Testing can be omitted...

Eureka!!! the system is accepted and it fulfills all stakeholder needs

but ideal does not exist in reality therefore waterfall model is out of touch with reality

- Iterative (RUP, SCRUM)

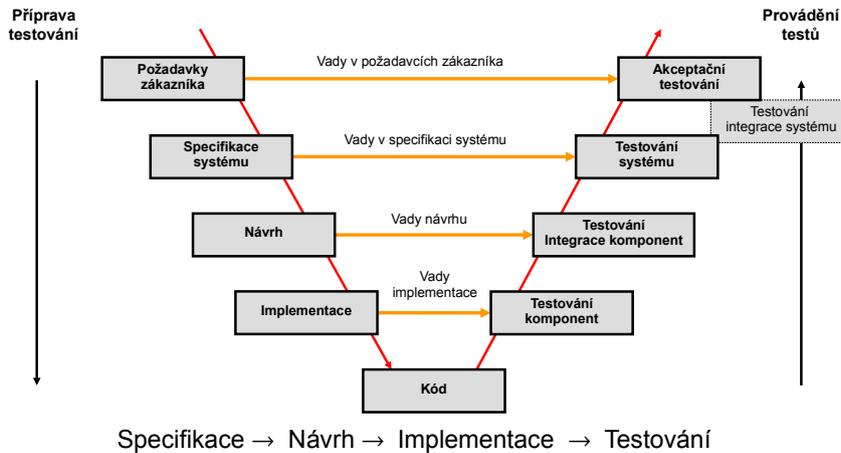
The development is divided into iterations. In the

first iteration we focus on a big picture. The project is split into small pieces (iterations), in which we deliver product to the customer to get customer feedback. Iterations are here to reduce time we are walking the wrong way (one iteration usually takes 2- 3 weeks). The iteration must not be changed during processing, all plans/bugs/etc must be planned for the next iteration. There must be no disturbance from the iteration plan - focus on the target. The iteration should end as planned and evaluated. Unfinished tasks together with bugs found in this iteration must be estimated again and planned for the beginning of the next iteration. Do not save bugs for later, unfixed bug means the work was not done. One or two iterations are planned just to remove bugs (no new functionality is implemented). In SCRUM terminology an iteration is called a Sprint.

- Extreme programming (XP)

It goes about agile software development methodology (rapid development), the set of daily practices that embody and encourage particular XP values: communication (simple design, common metaphors, collaboration of users and programmers, frequent verbal communication and feedback), simplicity (starting with the simplest solution), feedback (from the system by writing unit tests and running periodic integration tests, from the customer by acceptance testing, from the team by quick response to new requirements), courage (design and code for today and not for tomorrow - developers feel comfortable with refactoring their code when necessary) and respect between team members.

V-model: Úrovně testování



05/04/22

TSK

67

For each stage in the model there are deliverables to the next stage, both development and testing. Such a delivery is an example of a baseline.

For example, when the user requirements are ready, they are delivered both to the next development stage and to the corresponding test level, i.e. acceptance testing. The user requirements will be used as input to the system specification (where the system requirements will be the deliverable to the next stage) and the acceptance test design.

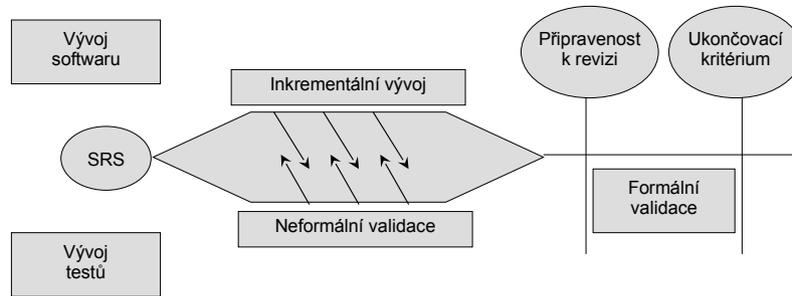
Note that this is a simplified model. In reality, the arrows should point in both directions since each stage naturally will find faults and give feedback to the previous stages.

Úrovně testování

- Testování komponent
- Testování integrace komponent
- Testování systému (funkční a nefunkční)
- Testování integrace systému
- Akceptační testování
- Obslužné testování (Maintenance testing)

- The objectives are different for each test level (see the V-model)
- Test techniques used (black- or white- box)
- Object under test, e.g. component, grouped components, sub-system or complete system
- Responsibility for the test level, e.g. developer, development team, an independent test team or users
- The scope of testing

Paralelní vývojový model



05/04/22

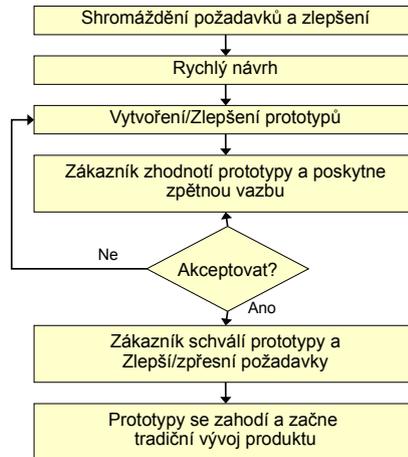
TSK

69

The concurrent development model is well suited for rapid, flexible development. In this model, the SRS is the starting point for development of both software and tests. Developers and test engineers work concurrently to develop and test the software. In the synchronize-and-stabilize model, the project team begins with the product vision, a vague description of what the product should do. An SRS evolves over the course of the project from this product vision.

As bits of the product are developed, they are immediately tested and feedback is provided to developers. In the synchronize-and-stabilize model, this usually occurs about three times during the project (it can occur as

Model rychlého prototypování (The Rapid Prototyping Model)



05/04/22

TSK

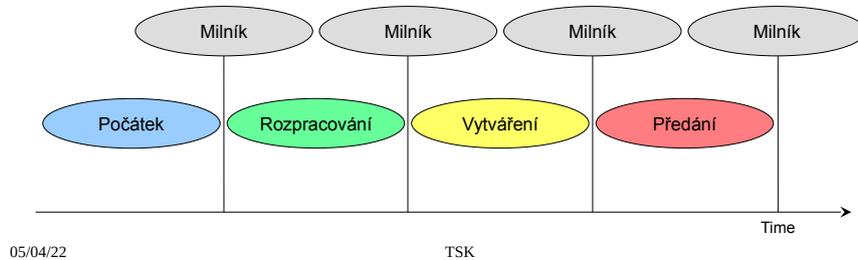
70

By using a prototyping approach, the customer can assess the prototype and provide feedback as to its suitability for a particular application. The prototype can range from a paper schematic all the way to a working system that includes both hardware and software.

The rapid prototyping model begins with a requirements-gathering stage whereby the developers collect and refine product requirements based on whatever information or source are available. Then, a rapid prototype is developed. This prototype is intended to be used for requirements exploration only. It is not intended to be the product. Customers can then evaluate and criticize the prototype, providing the

Rational Unified Process (RUP)

- Iterativní vývoj softwaru
- Management požadavků
- Architektura na bázi komponent
- Vizuální modelování softwaru
- Verifikace kvality softwaru
- Kontrola změn



RUP is an example of object-oriented methodologies that emphasize the incremental, iterative, and concurrent nature of software development.

RUP is a product process developed by Rational Software Corporation that provides project teams with a guide to more effective use of the industry-standard Unified Modeling Language (UML). RUP also provides software-engineering best practices through templates, guidelines, and tools. Most of the tools are, as you might guess, also provided by Rational.

The RUP is based on four consecutive phases. The purpose of the *inception phase* is to establish the business case for the project. This is done by creating several high-level use case diagrams, defining success criteria, risk assessment, resource estimate, and an overall plan showing the four phases and their approximate time frames. Some deliverables the inception phase might include are:

- A vision statement
- An initial set of use cases
- An initial business case
- An initial risk assessment
- An initial project plan
- Prototypes

The purpose of the *elaboration phase* is to analyze the problem domain, establish the overall product architecture, eliminate the highest risks, and refine the project plan. Evolutionary prototypes are developed to mitigate risks and address technical issues and business concerns. Some key deliverables this phase might include are:

- A relatively complete use case model supplemented with text as appropriate
- Architecture description
- Revised risk assessment
- Revised project plan
- Initial development plan
- Initial user manual

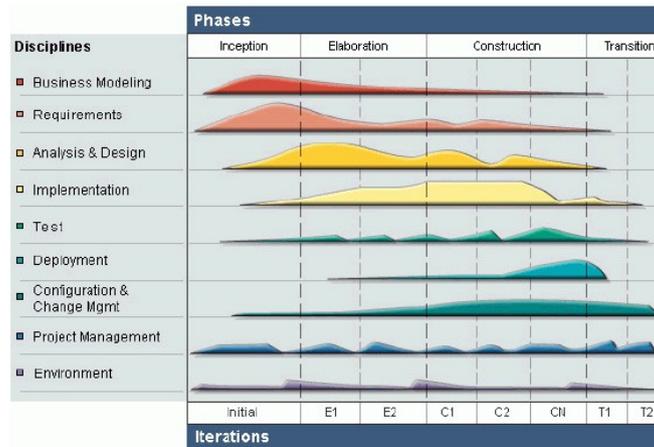
During the *construction phase*, the remaining components are developed, and thoroughly tested. Key deliverables from this phase include:

- Software product operating on target platform
- Revised user manual
- Complete description of current release

The purpose of the *transition phase* is to transition the product from development to the user community. Activities that would typically be performed include:

- Beta testing by users
- Conversion of existing information to new environment
- Training of users
- Product rollout

Rational Unified Process (RUP)



05/04/22

TSK

72

RUP is an example of object-oriented methodologies that emphasize the incremental, iterative, and concurrent nature of software development.

RUP is a product process developed by Rational Software Corporation that provides project teams with a guide to more effective use of the industry-standard Unified Modeling Language (UML). RUP also provides software-engineering best practices through templates, guidelines, and tools. Most of the tools are, as you might guess, also provided by Rational.

The RUP is based on four consecutive phases. The purpose of the *inception phase* is to establish the business case for the project. This is done by creating several high-level use case diagrams, defining success criteria, risk assessment, resource estimate, and an overall plan showing the four phases and their approximate time frames. Some deliverables the inception phase might include are:

- A vision statement
- An initial set of use cases
- An initial business case
- An initial risk assessment
- An initial project plan
- Prototypes

The purpose of the *elaboration phase* is to analyze the problem domain, establish the overall product architecture, eliminate the highest risks, and refine the project plan. Evolutionary prototypes are developed to mitigate risks and address technical issues and business concerns. Some key deliverables this phase might include are:

- A relatively complete use case model supplemented with text as appropriate
- Architecture description
- Revised risk assessment
- Revised project plan
- Initial development plan
- Initial user manual

During the *construction phase*, the remaining components are developed, and thoroughly tested. Key deliverables from this phase include:

- Software product operating on target platform
- Revised user manual
- Complete description of current release

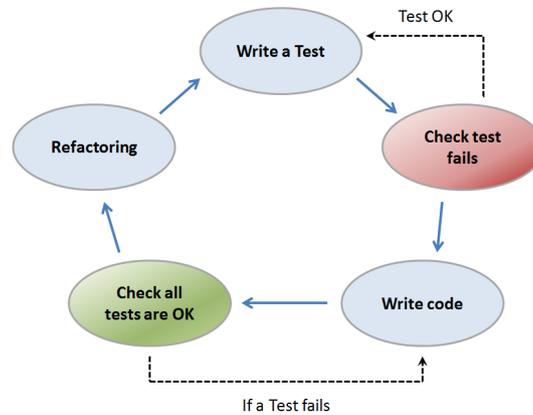
The purpose of the *transition phase* is to transition the product from development to the user community. Activities that would typically be performed include:

- Beta testing by users
- Conversion of existing information to new environment
- Training of users
- Product rollout

Test Driven Development

- TDD adopts a “Test-First” approach in which unit tests are written before code.
- This idea, which dates back to ancient times, was formalized in the mid-1990s by **Kent Beck**, who made it one of the pillars of the Extreme Programming (XP) methodology.
- TDD is a way of managing fear during programming.

Test Driven Development



05/04/22

TSK

74

1. Add a test

In test-driven development, each new feature begins with writing a test. Write a test that defines a function or improvements of a function, which should be very succinct. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.

2. Run all tests and see if the new test fails

This validates that the test harness is working correctly, shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility that the new test is flawed and will always pass. The new test should fail for the expected reason. This step increases the developer's confidence in the new test.

3. Write the code

The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.

At this point, the only purpose of the written code is to pass the test. The programmer must not write code that is beyond the functionality that the test checks.

4. Run tests

If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.

5. Refactor code

The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from recognized design patterns. There are specific and general guidelines for refactoring and

TDD - Clean Tests

- The test code is as important if not more important than the production code!
 - readability
 - simple, clear and as dense a test as possible
 - a unit test should represent only one concept and contain only one assertion

1. Add a test

In test-driven development, each new feature begins with writing a test. Write a test that defines a function or improvements of a function, which should be very succinct. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.

2. Run all tests and see if the new test fails

This validates that the test harness is working correctly, shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility that the new test is flawed and will always pass. The new test should fail for the expected reason. This step increases the developer's confidence in the new test.

3. Write the code

The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.

At this point, the only purpose of the written code is to pass the test. The programmer must not write code that is beyond the functionality that the test checks.

4. Run tests

If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.

5. Refactor code

The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from recognized design patterns. There are specific and general guidelines for refactoring and

TDD – Clean Tests

- 5 other rules that can be easily memorized using the acronym FIRST:
 - **Fast:** a test must be fast to be executed often.
 - **Independent:** tests must not depend on each other.
 - **Repeatable:** a test must be reproducible in any environment.
 - **Self-Validating:** a test must have a binary result (Failure or Success) for a quick and easy conclusion.
 - **Timely:** a test must be written at the appropriate time, i.e. just before the production code it will validate.

1. Add a test

In test-driven development, each new feature begins with writing a test. Write a test that defines a function or improvements of a function, which should be very succinct. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.

2. Run all tests and see if the new test fails

This validates that the test harness is working correctly, shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility that the new test is flawed and will always pass. The new test should fail for the expected reason. This step increases the developer's confidence in the new test.

3. Write the code

The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.

At this point, the only purpose of the written code is to pass the test. The programmer must not write code that is beyond the functionality that the test checks.

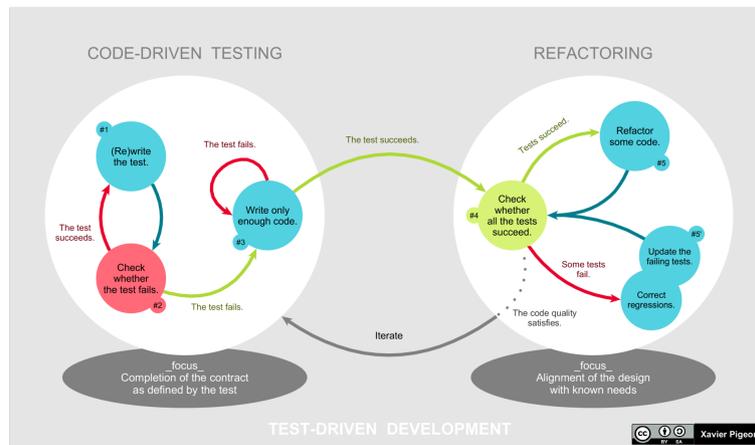
4. Run tests

If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.

5. Refactor code

The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from recognized design patterns. There are specific and general guidelines for refactoring and

Test Driven Development



05/04/22

TSK

77

1. Add a test

In test-driven development, each new feature begins with writing a test. Write a test that defines a function or improvements of a function, which should be very succinct. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.

2. Run all tests and see if the new test fails

This validates that the test harness is working correctly, shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility that the new test is flawed and will always pass. The new test should fail for the expected reason. This step increases the developer's confidence in the new test.

3. Write the code

The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.

At this point, the only purpose of the written code is to pass the test. The programmer must not write code that is beyond the functionality that the test checks.

4. Run tests

If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.

5. Refactor code

The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from recognized design patterns. There are specific and general guidelines for refactoring and

Test Driven Development

- **Test structure** - Effective layout of a test case ensures all required actions are completed, improves the readability of the test case, and smooths the flow of execution. Consistent structure helps in building a self-documenting test case. A commonly applied structure for test cases has (1) setup, (2) execution, (3) validation, and (4) cleanup.
- **Setup:** Put the Unit Under Test (UUT) or the overall test system in the state needed to run the test.
- **Execution:** Trigger/drive the UUT to perform the target behavior and capture all output, such as return values and output parameters. This step is usually very simple.
- **Validation:** Ensure the results of the test are correct. These results may include explicit outputs captured during execution or state changes in the UUT.
- **Cleanup:** Restore the UUT or the overall test system to the pre-test state. This restoration permits another test to execute immediately after this one.[8]

Individual best practices states that one should

- Separate common set-up and teardown logic into test support services utilized by the appropriate test cases.
- Keep each test oracle focused on only the results necessary to validate its test.
- Design time-related tests to allow tolerance for execution in non-real time operating systems. The common practice of allowing a 5-10 percent margin for late execution reduces the potential number of false negatives in test execution.
- Treat your test code with the same respect as your production code. It also must work correctly for both positive and negative cases, last a long time, and be readable and maintainable.
- Get together with your team and review your tests and test practices to share effective techniques and catch bad habits. It may be helpful to review this section during your discussion.

Practices to avoid - "anti-patterns"

- Having test cases depend on system state manipulated from previously executed test cases (i.e., you should always start a unit test from a known and pre-configured state).
- Dependencies between test cases. A test suite where test cases are dependent upon each other is brittle and complex. Execution order should not be presumed. Basic refactoring of the initial test cases or structure of the UUT causes a spiral of increasingly pervasive impacts in associated tests.
- Interdependent tests. Interdependent tests can cause cascading false negatives. A failure in an early test case breaks a later test case even if no actual fault exists in the UUT, increasing defect analysis and debug efforts.

Practices to avoid - "anti-patterns"

- Testing precise execution behavior timing or performance.
- Building "all-knowing oracles". An oracle that inspects more than necessary is more expensive and brittle over time. This very common error is dangerous because it causes a subtle but pervasive time sink across the complex project.
- Testing implementation details.
- Slow running tests.

Test Driven Development

Myth

- You create a 100% regression test suite

Reality

Although this sounds like a good goal, and it is, it unfortunately isn't realistic for several reasons:

- I may have some reusable components/frameworks/... which I've downloaded or purchased which do not come with a test suite, nor perhaps even with source code. Although I can, and often do, create black-box tests which validate the interface of the component these tests won't completely validate the component.
- The user interface is really hard to test. Although user interface testing tools do in fact exist, not everyone owns them and sometimes they are difficult to use. A common strategy is to not automate user interface testing but instead to hope that user testing efforts cover this important aspect of your system. Not an ideal approach, but still a common one.
- Some developers on the team may not have adequate testing skills.
- Database regression testing is a fairly new concept and not yet well supported by tools.
- I may be working on a legacy system and may not yet have gotten around to writing the tests for some of the legacy functionality.

Test Driven Development

Myth

- The unit tests form 100% of your design specification

Reality

- The reality is that the unit test form a fair bit of the design specification, similarly acceptance tests form a fair bit of your requirements specification, but there's more to it than this. (Agile Model - Driven Development -AMDD).
- Because you think about the production code before you write it, you effectively perform detailed design.

Test Driven Development

Myth

- You only need to unit test

Reality

- For all but the simplest systems this is completely false.
- The agile community is very clear about the need for a host of other testing techniques.

Test Driven Development

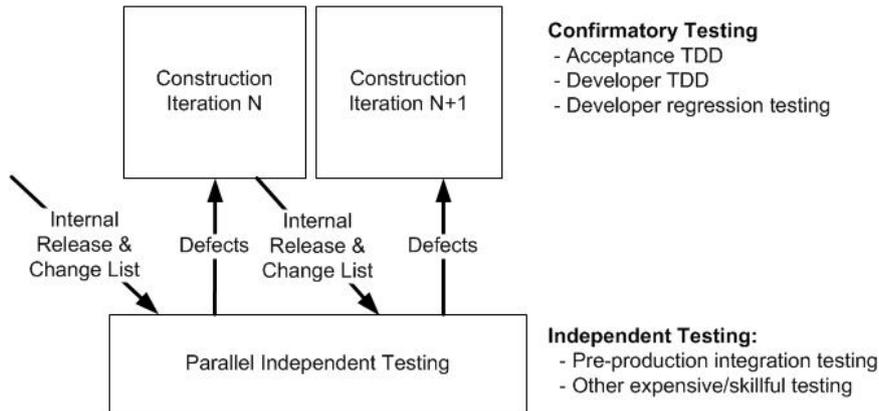
Myth

- TDD is sufficient for testing

Reality

- TDD, at the unit/developer test as well as at the customer test level, is only part of your overall testing efforts.
- At best it comprises your confirmatory testing efforts, but you must also be concerned about independent testing efforts which go beyond this.

Test Driven Development



Copyright 2014 Disciplined Agile Consortium

05/04/22

TSK

86

1. Add a test

In test-driven development, each new feature begins with writing a test. Write a test that defines a function or improvements of a function, which should be very succinct. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.

2. Run all tests and see if the new test fails

This validates that the test harness is working correctly, shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility that the new test is flawed and will always pass. The new test should fail for the expected reason. This step increases the developer's confidence in the new test.

3. Write the code

The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.

At this point, the only purpose of the written code is to pass the test. The programmer must not write code that is beyond the functionality that the test checks.

4. Run tests

If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.

5. Refactor code

The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from recognized design patterns. There are specific and general guidelines for refactoring and

Test Driven Development

Myth

- TDD doesn't scale

Reality

This is partly true, although easy to overcome. TDD scalability issues include:

- 1) Your test suite takes too long to run. This is a common problem:
 - First, separate your test suite into two or more components. One test suite contains the tests for the new functionality that you're currently working on, the other test suite contains all tests. You run the first test suite regularly, migrating older tests for mature portions of your production code to the overall test suite as appropriate. The overall test suite is run in the background, often on a separate machine(s), and/or at night.
 - Several levels of test suite -- development sandbox tests which run in 5 minutes or less, project integration tests which run in a few hours or less, a test suite that runs in many hours or even several days that is run less often.
- 2) Not all developers know how to test.
 - That's often true, so get them some appropriate training and get them pairing with people with unit testing skills.
- 3) Everyone might not be taking a TDD approach.
 - Taking a TDD approach to development is something that everyone on the team needs to agree to do.
 - they either need to start
 - they need to be motivated to leave the team
 - team should give up on TDD.

2.2- Ekonomie testování

Vytváří testování hodnotu nebo ne?

- Redukce obchodních rizik
- Zajistí čas do uvedení na trh (time to market)
- Kontrola nad cenou

- *Reduction of business risk*
 - By assuring that the application functionality meets the business objectives
 - By decreasing of application failure and downtime
 - By assuring the interoperability of the information systems
 - By assuring required performance of the applications
- *Ensuring time to market*
 - By managing the overall software development lifecycle
 - By facilitating the need for diversified competences
 - By managing the complex relationships of the development organization, customer business processes and technology

Návratnost (ROI) testování

- Investice do testování (lidské zdroje, nástroje, testovací prostředí)
- Náklady spojené s vadou
 - Vývoj (ztráta dat, hledání příčiny (RCA), oprava, slinkování(building), distribuce)
 - Byznys(délka výpadku, ztráta zákazníků, ztráta zisk, poškození důvěryhodnosti společnosti)
- Úspora nákladů (investice do testování snižuje náklady spojené s vadami)

Návratnost (ROI)= Úspora nákladů – Investice do testování

05/04/22

TSK

89

ROI = Return On Investments

- Investments to testing

Costs connected with testing activity:

- People (testers, test analysts, test developers, test managers)
- Test tools (test management tools, test-running tools, performance tools, etc.)
- Test environments (exact copy of the production system – can include many expensive servers with operating systems, drivers, applications, databases)

We can save these costs by not doing testing.

- Defect related costs

Fixing found defects brings direct costs in development but bad quality of software brings indirect costs in the business.

Examples of development costs:

- Restoring lost data
- Root Cause Analysis
- Fixing the defects
- Building new release, patch, emergency fix
- Distribution to customers, replacing the system

Examples of business costs:

- System that doesn't work cannot generate profit
- Loss of customers that run over to the competition
- Late delivery => delayed sales => lost profit
- Damage to corporate identity

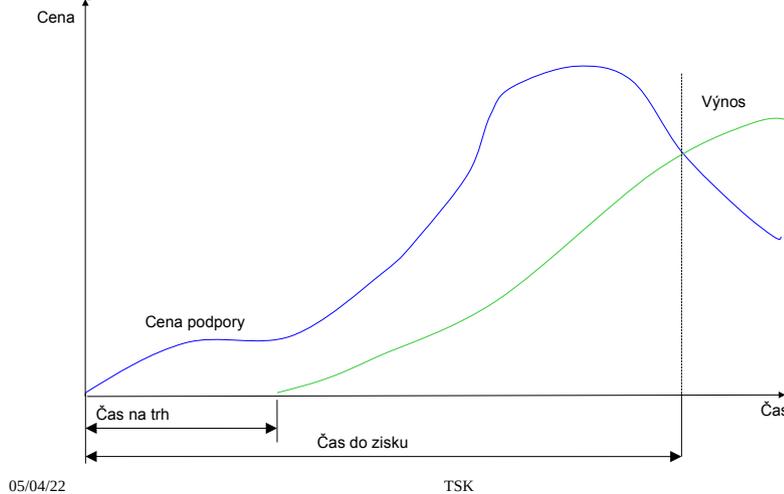
- Cost savings

The overall costs can be controlled by investing to testing.

+ROI - cost savings are greater than investments to testing (when testing process is mature and stable)

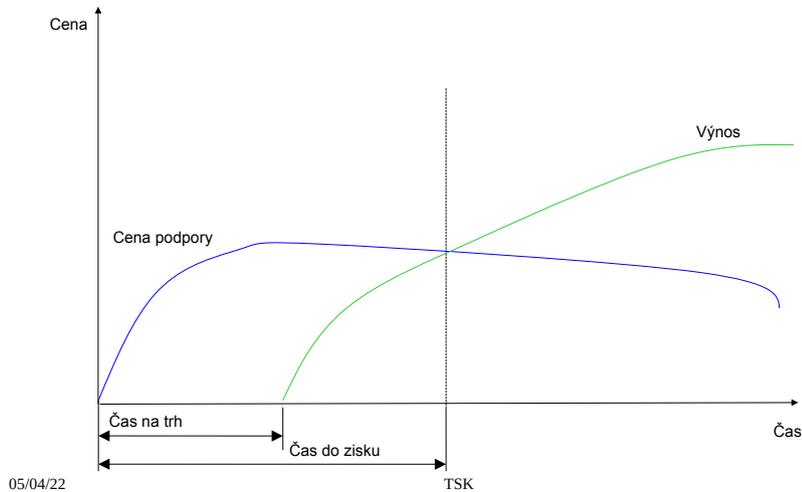
-ROI – cost saving are less than investments to testing (for first projects, first releases, etc.)

Vývoj bez testování



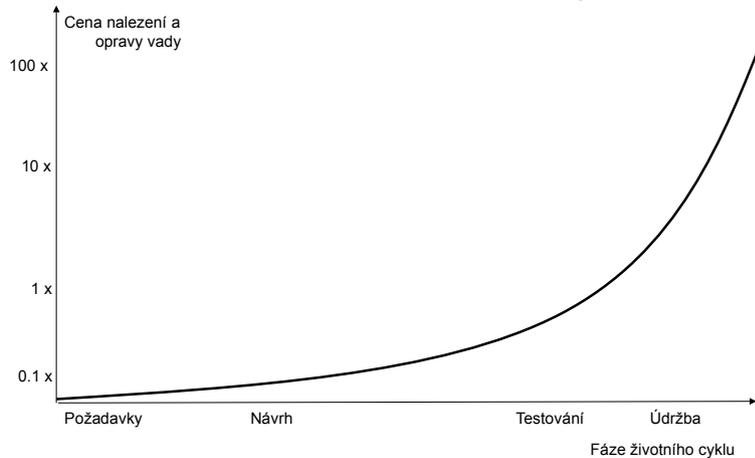
In most organizations, the pressure to get products to market as quickly as possible is intense. Application development time frames that once took years are now compressed to months. For Web applications, the advent of Internet time has led to development cycles measured in days and weeks. As a result, many organizations are looking for ways to streamline the development process in order to meet increasing market pressures and make decisions related to product quality based on some notion that the product is good enough.

Vývoj s testováním



Time-to-profit represents the time from when a product is released to the break-even point – the point at which the revenue stream generated by sales of the product exceeds the cost of maintaining and supporting the product. When the software development organization is focused solely on time-to-market goals, the quality of the product frequently suffers. Releasing a low-quality products usually results in higher maintenance and support costs and unhappy customers. The break-even point occurs much later (if at all). Alternatively, if the software development organization is geared towards achieving time-to-profit goals, the quality of the product is usually much better, thus reducing overall maintenance and

Cena odstranění vady v průběhu vývoje



05/04/22

TSK

92

This picture shows what happens to the costs when faults are found during different development phases. As you can see the costs increase dramatically for faults found during field use.

- *A good way to find faults is to design tests*

Faults in specifications are often found while analyzing specifications during test design. If test design starts early those faults will be found early.

- *Most important faults are found first*

Faults in requirements specifications are the most important ones since they will affect the system design and will be built in to the system during each stage of the development. This type of faults are very expensive to fix if they are found late.

Cena vady

Scénář 1: Vada je nalezena ve specifikaci požadavků ve fázi časné přípravy testů

- Úprava specifikace požadavků a její revize/inspekce
- Ujistění se, že je vše obsaženo v hlavní linii určené pro vývoj

Scénář 2: Zákazník objeví se vážnou vadu v průběhu běžného používání

- Nalezení důvodu selhání
- Analýza problému / vedlejší účinky vady / opravit vadu?
- Oprava vady v kódu a odpovídající specifikaci
- Úprava dokumentace, revize/inspekce
- Testování opravené vady na několika úrovních
- Regresní testování
- Ujistění se, že všichni postižení zákazníci dostanou novou verzi systému/produktu

- Scenario 1: A fault is found in a requirement specification during early test preparation

Analysis of specifications during test preparation often reveals faults. It could be e.g. ambiguous or contradictory requirements, faulty assertions or missing requirements. This kind of faults can be very expensive to fix later, due to fault multiplication in the different development phases. The earlier the faults are found, the cheaper it is to fix them.

- Scenario 2: A serious failure is found by the customer during field use

There are a lot of factors that can make the costs for fixing the fault very high, e.g.:

- Not possible to reproduce the fault in the test environment. Testers/developers must be sent to the customers environment to trace the fault.
- Thousands of instances of the erroneous product/system might exist, in worst case distributed over the whole world, and all of them must be fixed.

Cena chabého testování

- Testování je drahé!
 - V porovnání s čím?
- Nízká kvalita produktu/systému
 - Proč by si to lidé měli koupit?
 - Dlouhodobá reputace, proč by si lidé měli kupovat jakýkoliv produkty/systém od této společnosti?
 - Cenné zdroje musí trávit čas na odstranění vad ve starém produktech/systémech namísto podílení se na tvorbě nového velmi kvalitního produktu/systému
- Mnohem dražší je opravit vadu později

Testing is expensive! This is something that we have to listen to far too often. It is not true that testing is expensive. It depends on what to compare with. If the cost of testing is compared to the cost of development activities within a project, testing may appear as expensive. What will happen if we skip the testing or do poor testing? We will probably end up with a low quality product in field use. There is a close connection between testing and quality. The cost of testing should be compared with the cost of having a low quality product in use instead. The cost for maintaining a low quality product can quite fast become huge.

There are many risks with having a low quality product. One of them is that valuable

2.3 - Plánování testů

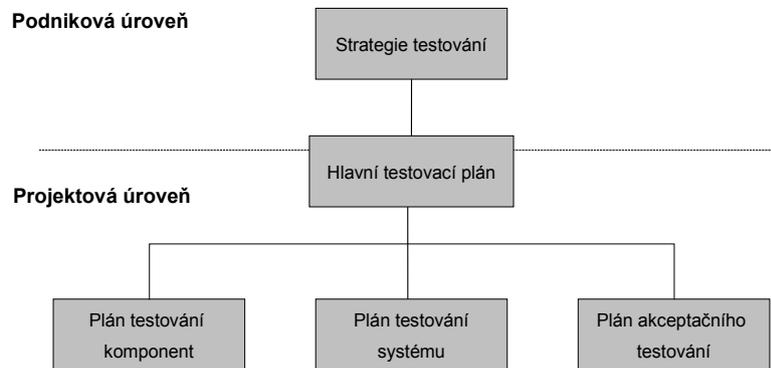
Účel plánování testů

- Plánování, kontrola, koordinace, sledování všech aktivit uvnitř procesu testování
- Určení „směru“ jak testovat – strategie, procesy, metody, nástroje, šablony
- Koordinace a poskytnutí zpětné vazby s okolními procesy
- Definice podmínek pro kontrolované vyřešení neočekávaných událostí

The purpose of the test planning is:

- To plan, control, co-ordinate and follow up all activities within the testing workflow.
- To give directions on how testing should be performed, which means to specify which test strategies, processes, methods, tools and templates to be used.
- To co-ordinate and give continuous feedback to the workflows that provide input for the test planning, and to the workflows that use the output.
- To create conditions for controlled adjustments of unforeseen events.

Úrovně testovacích plánů



05/04/22

TSK

96

Company level

Many companies have a documented test strategy, i.e. a separate test strategy document or a Master Test Plan on company level.

Project level

Depending on project size and complexity, the test planning can be divided into a hierarchy of test plans, e.g. one high level test plan (Master Test Plan) and some more detailed test plans (Component Test Plan, System Test Plan, Acceptance Test Plan). The division can follow test areas, test stages or specific aspects of testing. The picture above is an example of a test plan hierarchy. The Master Test Plan is a comprehensive description of all testing that will be done in

Obsah testovacího plánu (ANSI/IEEE 829-1998)

1. Identifikátor testovacího plánu
2. Úvod
3. Testované položky
4. Testované vlastnosti
5. Netestované vlastnosti
6. Přístup
7. Podmínky úspěšného/neúspěšného ukončení testů položky
8. Podmínky odložení/přerušování a požadavky pro pokračování

1. Test plan identifier

The test plan must be uniquely identified.

2. Introduction

Summarise the software items and features to be tested. References to project authorization, Project Plan, Quality Plan, Configuration Management Plan, relevant policies and standards shall be specified. References to higher level test plans shall also be included, if applicable.

3. Test items

Identify test items including their version/revision. Supply references to available test item documentation, such as requirement specifications, design specifications, users guide, operations guide, installation guide, etc.

4. Features to be tested

Identify all software features and combinations of features to be tested and associated test design specifications.

5. Features not to be tested

Identify features and combinations of features that will not be tested. Specify the reasons why they will not be tested.

6. Approach

Describe the overall approach to testing. Identify groups of features and specify the chosen test approach that will ensure adequate testing for each group. Specify major activities, techniques and tools. Specify the minimum degree of the desired test comprehensiveness and how to determine when the testing effort is sufficient. Identify techniques to be used to trace requirements. Identify significant constraints.

7. Item pass/fail criteria

Specify the criteria to be used to determine whether each test item has passed or failed testing.

8. Suspension criteria and resumption requirements

Specify the criteria used to suspend all testing or parts of the testing activity described in this test plan. Specify testing activities that must be repeated when testing is resumed.

Obsah testovacího plánu (ANSI/IEEE 829-1998)

- 9. Dokumenty (Test deliverables)
- 10. Úkoly testování
- 11. Požadavky na prostředí
- 12. Zodpovědnosti
- 13. Požadavky na počet pracovníků a školení
- 14. Rozvrh/plán
- 15. Rizika a mimořádné události
- 16. Schválení/souhlas

1. Test plan identifier

The test plan must be uniquely identified.

2. Introduction

Summarise the software items and features to be tested. References to project authorization, Project Plan, Quality Plan, Configuration Management Plan, relevant policies and standards shall be specified. References to higher level test plans shall also be included, if applicable.

3. Test items

Identify test items including their version/revision. Supply references to available test item documentation, such as requirement specifications, design specifications, users guide, operations guide, installation guide, etc.

4. Features to be tested

Identify all software features and combinations of features to be tested and associated test design specifications.

5. Features not to be tested

Identify features and combinations of features that will not be tested. Specify the reasons why they will not be tested.

6. Approach

Describe the overall approach to testing. Identify groups of features and specify the chosen test approach that will ensure adequate testing for each group. Specify major activities, techniques and tools. Specify the minimum degree of the desired test comprehensiveness and how to determine when the testing effort is sufficient. Identify techniques to be used to trace requirements. Identify significant constraints.

7. Item pass/fail criteria

Specify the criteria to be used to determine whether each test item has passed or failed testing.

8. Suspension criteria and resumption requirements

Specify the criteria used to suspend all testing or parts of the testing activity described in this test plan. Specify testing activities that must be repeated when testing is resumed.

2.4 - Testování komponent

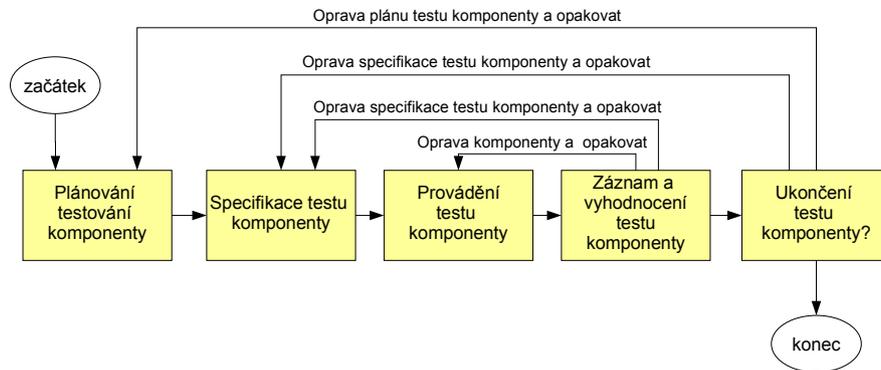
- První možnost vůbec něco spustit
- Rozdílné označení (Unit, Modul, Základní(Basic), ... Testování)
- Obvykle provádějí programátoři
- Mělo by testovat aktivitu v největším detailu

The objective of component (unit, module) testing is to find bugs in individual components (units, modules) by testing them in an isolated environments. Component testing is the first *dynamic* testing activity in the development life cycle. Traditionally (and most practically) component testing have been performed by programmers. One major reason for this is that component testing tends to require knowledge of the code which is why developers are well suited for this. Unfortunately component testing is often viewed more as a debugging activity than as a testing activity.

Mature ladder:

- Developers are checking their own modules with little or no documentation (they are blind

Proces komponentního testování (BS 7925-2)



05/04/22

TSK

100

- Component Test Planning

The component test planning contains two phases. *In the first phase* the overall project test strategy (generic) and the project test plan (project specific) are defined. The project test strategy includes test case selection methods, documentation, entry and exit criteria as well as the component test process itself. The project test plan contains information of the scope of the project, the resources needed and how to apply the strategy in the current project. *The second phase* of the planning deals with components individually. For each component a separate component test plan is produced (to list the specific test case design techniques, the test measurement techniques, the tools including stubs and drivers and the test completion criteria that apply to the specific component).

- Component Test Specification

Component test specification is the activity of applying the test design techniques specified in the component test plan to the information in the design specification, producing a number of test cases. The test cases should be documented in the component test case specifications. Each test case should also have a unique name and contain enough detailed instructions on how to perform the test case and a reference to the requirement that is tested by that test case.

- Component Test Execution

During component test execution, the test cases are executed on the actual module, preferable in the priority order. However, things might happen during the execution of the test cases which may force deviations from the planned order of execution. This is normal and quite all right as long as the deviations are conscious choices.

- Component Test Recording

During the execution of test cases, test results are produced. Basically there are two types of results: *logs* and *pass/fail results*. A log is just a chronological list of events that took place during the execution. The second type of result is the result of comparing the actual and the expected output. After a fault is located it usually pays off to investigate where the fault was first introduced in the design process and it is a good practice to correct the fault in all documents that contain the fault. A component test report is the document which contains a summary of all results of the second type.

- Component Test Completion?

Based on the information in the component test reports, the specified exit criteria in the test strategy and/or the test plans, and the current time budget, the decision whether or not to continue testing, can be performed. Here there are also several options:

- Enough coverage has been obtained and quality of test object is OK => component testing can be ended and the component delivered to the next level of testing (usually to component integration testing).
- All test cases have been executed but enough coverage has not yet been achieved => more test cases have to be designed and executed to increase the coverage.
- Time is out but the quality of the test object is too low => negotiate with project stakeholders to get more time to test and to correct faults (this is however typically NOT the responsibility of the test sub-project).

Testování komponent – co testovat

- Algoritmus a logiku
- Datové struktury (globální a lokální)
- Rozhraní (Interfaces)
- Nezávislé cesty
- Podmínky ohraničení
- Ošetření vad

- Algorithms and logic:
 - - Have algorithms and logic been correctly implemented?
 -
- Data structures (global and local):
 - - Are global data structures used?
 - - If so, what assumptions are made regarding global data?
 - - Are these assumptions valid?
 - - Is local data used?
 - - Is the integrity of local data maintained during all steps of an algorithm's execution?
 -

2.5. – Testování integrace komponent

- Předpoklady
 - Víc než jedna (testovaná a akceptovaná) komponenta/subsystém
- Kroky
 - (Sestavení komponent/subsystémů)
 - Testování sestaveného výsledného subsystému se zaměřením na rozhraní mezi testovanými komponentami/subsystémy.
- Cíl
 - Otestovaný podsystém-systém

The objective of integration testing (integration testing in the small according to BS 7925-1) is to find bugs related to interfaces between modules as they are integrated together. Integration testing can be performed at any level in the development process where two or more parts (SW and/or HW) are to be assembled into something bigger. However, when the complete systems are to be integrated, the usual terminology is “system integration testing”. The starting point of an integration testing activity is two or more parts that have already been tested on their own.

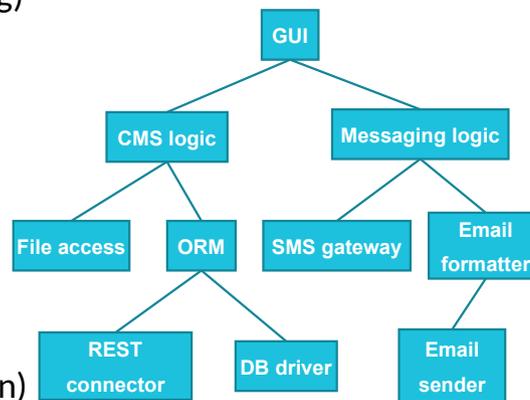
According to some people both the actual assembling of the parts and the following tests are part of the component integration testing. Other people only consider the actual *testing* as the component integration testing.

Even though all parts have already been tested separately we still want to test the assembled part. There are several reasons for this:

- One module can have an adverse effect on another
- Sub-functions, when combined, may not produce the desired major function
- Individually acceptable imprecision in calculations may be magnified to unacceptable levels
- There might be interface problems between two or more of the parts
- Timing problems (in real-time systems) are not detectable by component testing
- Resource contention problems are not detectable by component testing

Strategie

- Velký třesk (Big-bang)
- Odshora dolů (Top-down)
- Odspodu - nahoru (Bottom-up)
- Vlákňová integrace (Thread integration)
- Integrace minimální kapacity (Minimum capability integration)



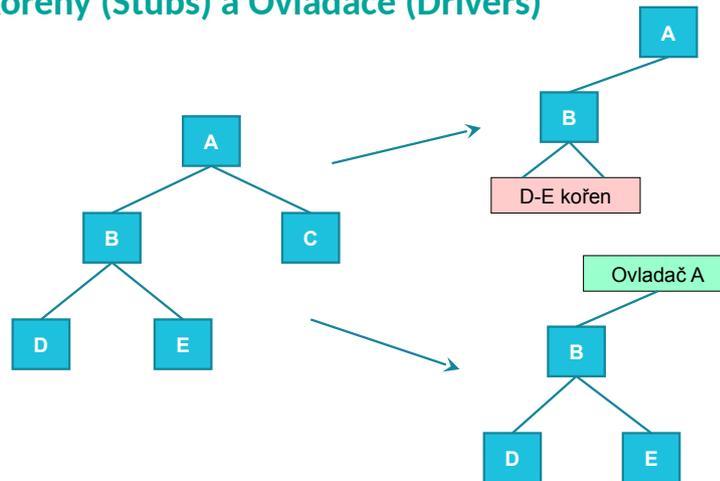
05/04/22

TSK

103

- **Big-Bang**: Assemble all components at once
- **Top-Down**: Start from the top, adding one new component at a time, e.g.:
 - Step 1: A + B
 - Step 2: A + B + C
 - Step 3: A + B + C + D, etc.
- **Bottom-Up**: Start from the bottom, adding one new component at a time, e.g.:
 - Step 1: H + E
 - Step 2: H + E + I
 - Step 3: H + E + I + B, etc.
- **Thread Integration**: Use some order of processing to determine the order of integration, e.g. a specific user transaction, an interrupt, etc.
- **Minimum Capability Integration**: Base the order of integration on which parts of the

Kořeny (Stubs) a Ovladače (Drivers)



05/04/22

TSK

104

- A stub is a small piece of code which is used to simulate a component that is lower in the calling hierarchy. The stub has the same interface as the component it replaces, but the stub has no (or very limited) intelligence. Its only purpose is to make sure the calling component can continue to execute even after the call.
 - A driver is a piece of code which is used to control the execution of one or more components. Usually the driver uses one of the existing interfaces of a module but sometimes, special interfaces are created in a module to allow drivers to be connected.
- Problems: Both stubs and drivers are software and thus need maintenance and configuration management. Neither stubs nor

Strategie

Integrace Velký třesk (Big-Bang)

- + Není potřeba kořenů ani ovladačů
- Obtížné lokalizovat vady, opakované testování po opravě je nákladnější

Integrace Odshora dolů

- + Přidávání k testované základně (jednodušší lokalizace vad), brzké zobrazení "funkčního" GUI
- Je potřeba mít kořeny (stubs), může vypadat více dokončeně než ve skutečnosti je, rozhodující detaily často přicházejí až na konec

Integrace odspodu nahoru

- + Dobré pro integraci hardwaru a sítí, viditelnost detailů
- Není fungující systém dokud není přidána poslední komponenta; je zapotřebí ovladačů (drivers).

05/04/22

TSK

105

- Big-Bang Integration (Everything at once)

It looks as if this is an easy way of integrating but in reality this is not the case. The major reason for this is the difficulty in locating faults. Since nothing is proven to work everything should be suspected every time a failure has been observed. Adding to this is also the "no faults in my code" syndrome. However, when most modules have already been integrated or if we have the lucky situation that after about 50% modules have been integrated without serious problems, then big-banging the rest might be an option to consider.

- Top-Down Integration (Start from top and add one at a time)

A not so obvious advantage with this approach is that after the first few modules have been integrated, the integration can be performed in parallel. The worst disadvantage is that performance critical parts of the system tend to be integrated last since they generally exist closest to the hardware, far away from the user interface. Faults in performance critical parts of the system often cause architectural changes, which take time and thus should be discovered as quickly as possible.

- Bottom-Up Integration (Start from bottom and add one at a time)

With bottom-up integration several teams can start integrating different parts of the system independently of one another. As integration progresses, it will be more and more serialized. The worst problem however with this strategy is the extensive need for stubs and drivers.

Strategie

Vláknová integrace

+ Kritické položky přijdou na řadu první, včasná výstraha před výkonnostními problémy

- Je zapotřebí komplexních kořenů a ovladačů

Integrace minimální kapacity

+ Brzy existuje skutečný fungující (částečně) systém, základní části jsou nejvíce testovány

- Je potřeba kořenů a ovladačů, základní systém může být velký

- Thread Integration (Choose order based on execution)

Thread integration is of much benefit if the amount of stubs and drivers can be kept to a minimum. This is the case only if the software architecture actually reflect the threads or tasks that the system is to perform. The most popular way of partitioning a system into modules is grouping similar functionality into the same module. The effect of such a partitioning from a thread/task perspective is that the same task may require the involvement of many modules, while each one of these modules performs a small part of many tasks. In such a situation thread integration is not recommended.

- Top-Down Integration (Start from top and add one at a time)

A not so obvious advantage with this approach is that after the first few modules have been integrated, the integration can be performed in parallel. The worst disadvantage is that performance critical parts of the system tend to be integrated last since they generally exist closest to the hardware, far away from the user interface. Faults in performance critical parts of the system often cause architectural changes, which take time and thus should be discovered as quickly as possible.

- Minimum Capability Integration (Basic functionality first)

Which are the necessary components for our system to be able to perform the most basic functionality it is designed to do? Suppose we are building a client/server application. The communication between the client and the server could then be considered a basic functionality... To get this to work we need operating systems on both the client and the server, communication software in both ends, some rudimentary application with a user interface on the client and finally some type of processing application, and possibly a database engine on the server. This is really a lot!

Pokyny pro integraci

- Integrační plán určí pořadí sestavování
- Minimalizace podpůrného softwaru
- Integrujte pouze malé množství komponent v jednom kroku
- Integrujte každou komponentu pouze jednou

As have been seen in the previous slides there really is no silver bullet in integration. No strategy will solve all problems. What usually happens is that several strategies are combined into a custom designed strategy that hopefully will work for the particular integration problem it is meant to solve.

A key activity here is integration planning. One part is to decide on the order in which the modules should be integrated and delivered. Another part is to make sure that there is enough back-up resources for unexpected events. This is particularly important during integration since many activities have to be performed serially, without possibilities for concurrent work. Therefore, the effect of a delay in almost any integration activity will

2.6 – Testování systému (funkční)

Funkcionální testování systému

- Testování funkcionálních požadavků
- První příležitost otestovat systém jako celek
- Testování E2E funkcionality
- Dvě rozdílné perspektivy:
 - Testování na základě požadavků
 - Testování na základě podnikových procesů

System testing is often divided into two categories, functional system testing and non-functional system testing. The non-functional tests are as important as the functional tests but they are often not as clearly specified as the functional requirements and may therefore be harder to test. Functional correctness is of course a pre-requisite for non-functional testing. System testing is usually performed by independent testers (departments).

Functional requirement: “Requirement that specifies a function that a system or system component must perform” (ANSI/IEEE Std. 729/1983). Functional system testing is probably the first opportunity to test the system as a whole. Parts of the functionality

Testování systému (funkční)

Testování na základě požadavků

- Testovací případy jsou odvozeny ze specifikace požadavků
- Většina požadavků potřebuje více než jeden testovací případ

Testování na základě podnikových procesů

- Testovací případy jsou tvořeny na základě profilu očekávaných uživatelů
- Byznys scénáře
- Případy užití

An example of requirements-based testing is when test cases are derived from the user requirements specification and the system requirements specification (as used for contracts).

There are usually one or more positive (correctness of the functionality as described in the specifications) and negative (opposite of the requirements – when they render an error message or exception) test cases.

Business process-based testing comes from using the system:

- Test cases are based on expected user profiles

To define the expected user profiles it can be useful to analyse the usage of old similar systems and interview users.

2.7. – Testování systému (systémové/nefunkční)

Co je Nefunkční testování systému?

- Testování nefunkčních požadavků
- Testování charakteristik systému
- Typy testů nefunkčních požadavků:
 - Efektivnosti, Zátěžové, výkonnostní, stresové
 - Bezpečnost, použitelnost
 - Úložiště (Storage), kapacita,
 - Instalovatelnost, platforma, zotavení(recovery), etc.
 - Dokumentace (inspekce a revize)

Testování efektivit

- Generic term „Performance testing“
 - [ISTQB] The process of testing to determine the performance of a software product.
 - [Rex Black] Testing to evaluate the degree to which a system or component accomplishes its designated functions, within given constraints, regarding processing time and throughput rate.

Testování efektivity – Advanced Software Testing Vol.

3

- Load testing
- Stress testing
- Scalability testing
- Resource utilization testing
- Endurance or soak testing
- Spike testing
- Reliability testing
- Background testing
- Tip-over testing

Load testing

- A type of performance testing conducted to evaluate the behavior of a component or system with **increasing load** (e.g., numbers of parallel users and/or numbers of transactions) to determine **what load can be handled** by the component or system.
- Typically, load testing involves various mixes and levels of load, usually focused on anticipated and realistic loads.
- The loads often are designed to look like the transaction requests generated by certain numbers of parallel users. We can then measure response time or throughput. Some people distinguish between multi-user load testing (with realistic numbers of users) and volume load testing (with large numbers of users), but we've not encountered that too often.

Stress testing

- A type of performance testing conducted to evaluate a system or component **at or beyond the limits** of its anticipated or specified workloads or with reduced availability of resources such as access to memory or servers.
- Stress testing takes load testing to the extreme and beyond by reaching and then exceeding maximum capacity and volume.
- The goal here is to ensure that response times, reliability, and functionality degrade slowly and predictably, **culminating in some sort of “go away I’m busy”** message rather than an application or OS crash, lockup, data corruption, or other antisocial failure mode.

Scalability testing

- Takes stress testing even further by **finding the bottlenecks** and then testing the ability of the system to be **enhanced to resolve** the problem.
- In other words, if the plan for handling growth in terms of customers is to add more CPUs to servers, then a scalability test verifies that this will suffice.
- Having identified the bottlenecks, scalability testing can also help establish load monitoring thresholds for production.

Resource utilization testing

- Evaluates the usage of various resources (CPU, memory, disk, etc.) while the system is running at a given load.

Endurance or soak testing

- Running a system at high levels of load for prolonged periods of time. A soak test would normally execute several times more transactions in an entire day (or night) than would be expected in a busy day to identify any performance problems that appear after a large number of transactions have been executed.
- It is possible that a system may stop working after a certain number of transactions have been processed due to memory leaks or other defects. Soak tests provide an opportunity to identify such defects, whereas load tests and stress tests may not find such problems due to their relatively short duration.

Spike testing

- The object of spike testing is to verify a system's stability **during a burst of concurrent user** and/or system activity to varying degrees of load over varying time periods. Here are some examples of business situations that this type of test looks to verify a system against:
 - A fire alarm goes off in a major business center and all employees evacuate. The first alarm drill completes and all employees return to work and log into an IT system within a 20-minute period.
 - A new system is released into production and multiple users access the system within a very small time period.
 - A system or service outage causes all users to lose access to a system. After the outage has been rectified, all users then log back onto the system at the same time.
 - Spike testing should also verify that an **application recovers between periods of spike** activity.

Reliability testing

- Testing the ability of the system to perform required functions under stated conditions for a specified period of time or number of operations.

Background testing

- Executing tests with active background load, often to test functionality or usability under realistic conditions.

Tip-over testing

- Designed to find the point where total saturation or failure occurs. The resource that was exhausted at that point is the weak link. Design changes (ideally) or more hardware (if necessary) can often improve handling and sometimes response time in these extreme conditions.

Efficiency testing and SDLC (1)

- During the development phases, from requirements through implementation, static testing should be done to ensure meaningful requirements and designs from an efficiency viewpoint.
- During unit testing, performance testing of individual units (e.g., functions or classes) should be done. All message and exception handling should be scrutinized; each message type could be a bottleneck. Any synchronization code, use of locks, semaphores, and threading must be tested thoroughly, both statically and dynamically.

Efficiency testing and SDLC (2)

- During integration testing, performance testing of collection of units (builds or backbones) should be performed. Any code that transfers data between modules should be tested. All interfaces should be scrutinized for deadlock problems.
- During system testing, performance testing of the whole system should be done as early as possible. The delivery of functionality into test should be mapped so that those pieces that are delivered can be scheduled for the performance testing that can be done.
- During acceptance testing, the performance of the whole system in production should be demonstrated (after making sure it is going to work with earlier testing, right?).

Zátěžové testování

”Testování prováděné za účelem ověření zda systém nebo komponenta vyhovuje specifikovaným požadavkům na pracovní zátěž” – BS 7925-1

- Může systém:
 - zvládnout očekávanou zátěž?
 - zvládnout očekávanou zátěž po určitou dobu?
 - provádí své úkoly zatím co zvládá očekávanou zátěž?
- Zátěžové testování vyžaduje nástroje
- Zátěžové testování lze mimo jiné použít na nalezení vad alokace paměti (memory leaks) nebo špatné práce s ukazateli (pointer errors)

Can the system handle expected workload over time?

The purpose of these tests is to verify that the system can handle the required load during a long time without problems. The types of faults that can be found during these tests are for instance memory leaks.

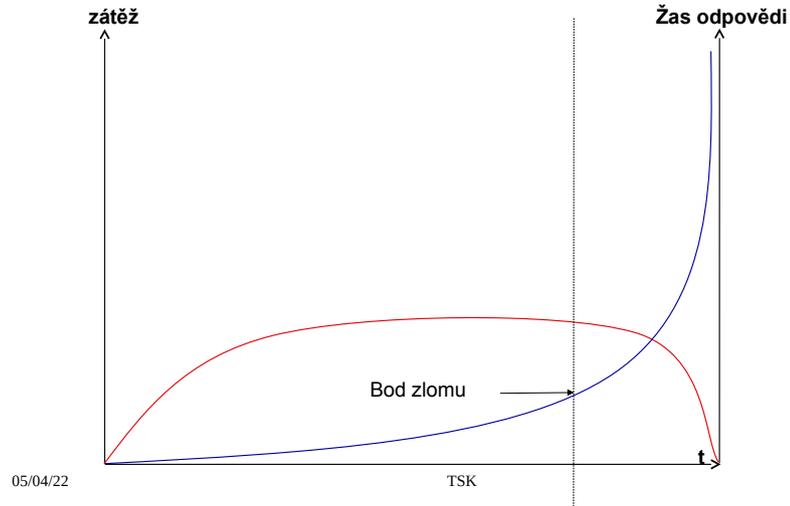
Can the system perform it's tasks while handling expected workload?

To verify this, functional testing is performed while having background load equivalent to the expected workload.

Load testing requires tools

Tools will be needed, e.g. for load generation, test running, test monitoring and analysis.

Zátěžové testování



Load testing could be called also as a stability testing, because this testing is runned under specified work load for hours or even for days, to find out if there are problems e.g. memory usage.

Výkonnostní/Škálovatelné testování

”Testování prováděné za účelem ověření zda systém nebo komponenta vyhovuje specifikovaným výkonnostním požadavkům.” – BS 7925-1

- Může systém:
 - zvládnout požadovaný výkon/propustnost bez dlouhé odezvy?
- Výkonnostní testování vyžaduje nástroje
- Výkonnostní testování je hlavně o kolektivní spolupráci administrátorů týmů databáze, systému, sítí a testů.

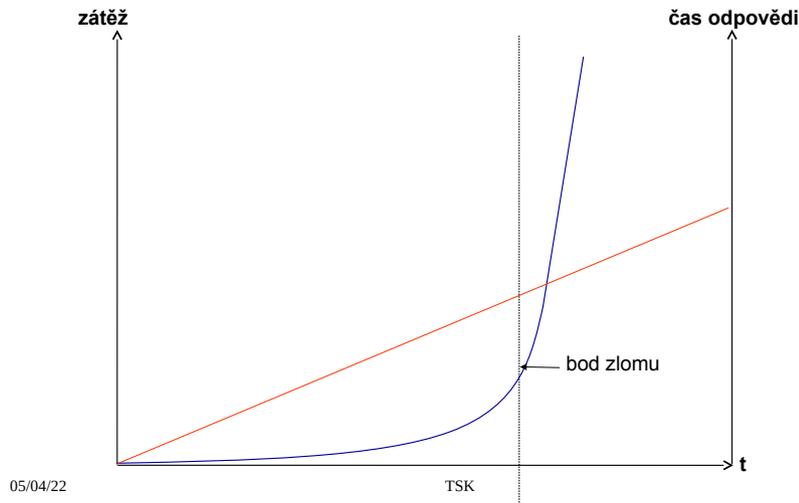
Can system handle required throughput without long delays?

Primary goal of performance testing is to verify that the system can handle the acceptable system performance. The types of faults that can be found during these tests are for instance bottlenecks.

Performance testing requires tools

Tools will be needed, except others, e.g. for generating time/load graphs.

Výkonnostní testování



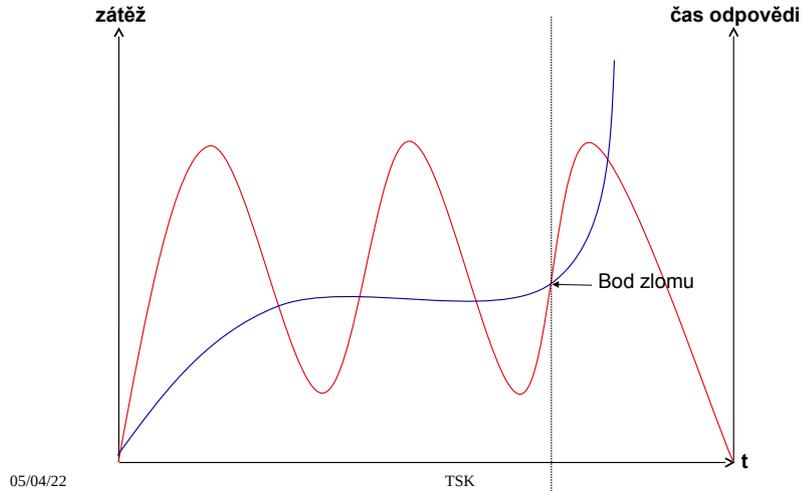
The target of performance testings are to find out the point, when the system starts to suffer, i.e. response time rises and system doesn't respond anymore to user actions.

Stresové/Robustní testování

”Testování prováděné za účelem ověření chování systému nebo komponenty na hranicích specifikované zátěže nebo za nimi.”

- Smysl testu:
 - Může systém zvládnout maximální očekávanou nebo vyšší zátěž?
 - Pokud jsou naše očekávání nesprávná, můžeme si být jisti že systém přežije?
- Mnělo by být testováno pokud možno co nejdřív.
- Stresové testování vyžaduje nástroje.

Stresové testování



In stress testing we continually reload and reduce load to find out possible memory leaks or bottlenecks.

Plánování výkonnostního testování

- Získání informací o tom jak bude systém používán
 - Množství uživatelů s rozdílnými profily
 - Podnikové procesy
 - Typické úkoly uživatelů (skripty)
 - Dynamický plán pro virtuální uživatele (scénáře)
- Spuštění výkonnostních testů a monitorování rozdílných částí informačního systému (LAN, servery, klienti, databáze, atd.)
- Poskytnutí zpětné vazby vývojářům s nějakými návrhy
- Vyladování systému dokud není dosaženo požadované výkonnosti

Testování efektivity - příklady

- Úřad zjistil důvod sobotního výpadku elektronického sčítání lidu. „Vyloučili jsme útok na svoje servery zvenčí, prověřil to i NUKIB,“ uvedl předseda ČSÚ Marek Rojíček v sobotu po poledni. Příčinou výpadku online sčítání podle něj je závada modulu pro vyplnění adres, která zapříčinila zahlcení databázových serverů.
-
- Zjednodušeně řečeno, špatně fungující nápověda - našeptávač adres - příliš zahltil servery, a to vedlo k jejich výpadku. Chyba se podle něj neukázala v žádném z testů, které spuštění sčítání předcházely.
-
- Zdroj: https://www.idnes.cz/zpravy/domaci/online-scitani-lidu- nefunguje-sluzba-je-docasne- nedostupna.A210327_074515_domaci_lre

Testování bezpečnosti

„Testování zda systém splňuje specifikované bezpečnostní vlastnosti” - BS 7925-1

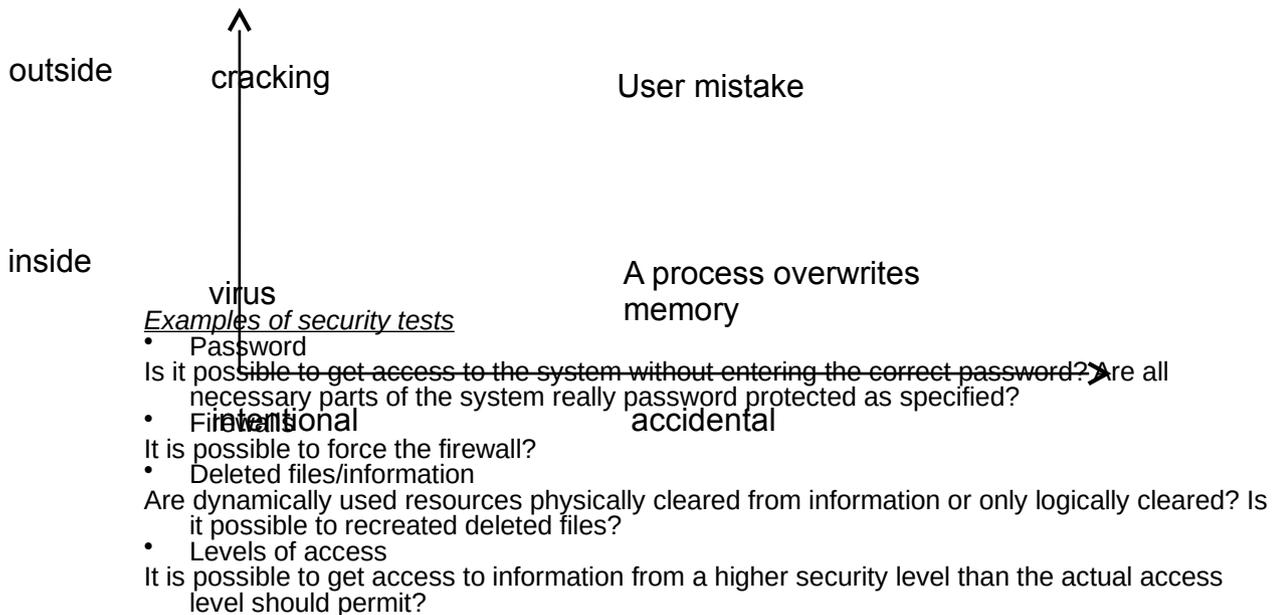
- Účelem bezpečnostního testování je získání jistoty, že systém je dostatečně bezpečný
- Testování bezpečnosti může být velmi nákladné
- Je důležité myslet na bezpečnost v raných fázích projektu
- Hesla, šifrování, bezpečnostní brány (firewall), smazané soubory/informace, úroveň přístupu, atd.

05/04/22

TSK

132

There are different types of security problems:



Testování použitelnosti

“Testování snadnosti s jakou se může uživatel naučit používat produkt.” – BS 7925-1

- Jak specifikovat použitelnost?
- Je vůbec možné testovat použitelnost?
- Přednostně prováděno koncovými uživateli
- Pravděpodobně jejich první kontakt se systémem
- První dojem musí být dobrý!

The real problem of usability testing is the subjectivity of users.

How can we know if the system is usable?

Does the GUI follow available standards?

Is the order for entering data logical?

Are the default values reasonable?

Are messages from the system, e.g. error messages, understandable?

Does the system have a consistent behavior?

Is it easy to convert data from an old system to the new one?

Is the online help logical?

Does it work for both experienced and novice users?

Testování úložiště (storage) a kapacity

Testování úložiště:

“Testování zda systém splňuje specifikované požadavky na úložiště.” – BS 7925-1

Testování kapacity (volume):

“Testování systému na práci s velkým množstvím data” – BS 7925-1

- Oba dva typy testu vyžadují nástroje

Volume testing measures the database behavior under stress.

Both storage and volume testing requires tools since big volumes of data is needed. Dummy data can be used to fill for example a database.

The system can be scaled down to test what happens when the specified limit is reachable.

Example:

The maximum volume of the database that the system can access is 20 Mb. 19.9 Mb is filled with dummy data. What happens when more data is added?

Testování instalovatelnosti

“Testování týkající se instalačních procedur systému” – BS 7925-1

- Je možné nainstalovat systém podle pokynů instalačních procedur?
- Jsou procedury logické, jednoznačné a lze podle nich snadno provádět jednotlivé kroky instalace?
- Je možné systém povýšit (upgrade)?
- Je možné systém odinstalovat?

Installability testing is often as much testing of instructions/documents as testing of the system.

It's connected with the configuration testing:

- Are data (files, database tables) installed and populated correctly?
- Are data consistent after upgrade procedure?
- Are data correctly transformed after applying upgrade procedure?

Is it possible to uninstall the system?

Things to check while testing this is that all files belonging to the uninstalled system are removed and that everything else is restored (e.g. previous version of the system). Another interesting thing to test is to install the system again.

Testování platformy (přenositelnosti)

“Testy navržené za účelem ověření zda software pracuje správně na všech podporovaných platformách/operačních systémech”

- Je webová aplikace ověřena se všemi známými a používanými prohlížeči (Chrome, Firefox, Edge, ...)?
- Je aplikace ověřena na všech podporovaných verzích operačního systému (MS Windows 9x, NT, 2000, XP, Vista, Ubuntu, iOS, Android)?
- Je systém ověřen na všech podporovaných hardwarových platformách?

Platform (portability) testing is based on user specification of hardware and software for which the system was designed and developed. The testing (at least sanity check) must be performed on all environments.

Testování zotavení

“Testování zaměřené na ověření schopnosti systému zotavit se z různě závažných selhání” – BS 7925-1

- Je systém schopen zotavení po softwarovém/hardwarovém selhání nebo chybných dat?
- Jak můžeme do systému vnést realistické selhání?
- Musí být otestována funkčnost zálohování!

The biggest problem with recovery testing is how to generate faults that causes a recovery. A power cut is easy to simulate but it is hard to simulate faulty data without affecting the rest of the system.

If commands exist to initialize recovery/restart, they must be tested. What happens to the ongoing transactions during a recovery? Are the transactions and the information about the transactions handled correctly? Are any transactions lost or destroyed and, in that case, do we know that?

If back-up functionality for the system exists, it must be tested:

- Manual procedures, documentation, where is the back-up data stored?
- How can we be sure that the system works

Testování zotavení - gitlab.com

gitlab.com incident (at 2017/01/31 11pm)

- team-member-1 thinks that perhaps pg_basebackup is refusing to work due to the PostgreSQL data directory being present (despite being empty), decides to remove the directory. After a second or two he notices he ran it on db1.cluster.gitlab.com, instead of db2.cluster.gitlab.com.
- At 2017/01/31 11:27pm UTC, team-member-1 - terminates the removal, but it's too late. Of around 300 GB only about 4.5 GB is left.

The biggest problem with recovery testing is how to generate faults that causes a recovery. A power cut is easy to simulate but it is hard to simulate faulty data without affecting the rest of the system.

If commands exist to initialize recovery/restart, they must be tested. What happens to the ongoing transactions during a recovery? Are the transactions and the information about the transactions handled correctly? Are any transactions lost or destroyed and, in that case, do we know that?

If back-up functionality for the system exists, it must be tested:

- Manual procedures, documentation, where is the back-up data stored?
- How can we be sure that the system works

gitlab.com incident - Problems encountered

- LVM snapshots are by default only taken once every 24 hours. Team-member-1 happened to run one manually about six hours prior to the outage because he was working in load balancing for the database.
- Regular backups seem to also only be taken once per 24 hours, though team-member-1 has not yet been able to figure out where they are stored. According to team-member-2 these don't appear to be working, producing files only a few bytes in size.
- Team-member-3: It looks like `pg_dump` may be failing because PostgreSQL 9.2 binaries are being run instead of 9.6 binaries. This happens because omnibus only uses Pg 9.6 if `data/Pg_VERSION` is set to 9.6, but on workers this file does not exist. As a result it defaults to 9.2, failing silently. No SQL dumps were made as a result. Fog gem may have cleaned out older backups.

The biggest problem with recovery testing is how to generate faults that causes a recovery. A power cut is easy to simulate but it is hard to simulate faulty data without affecting the rest of the system.

If commands exist to initialize recovery/restart, they must be tested. What happens to the ongoing transactions during a recovery? Are the transactions and the information about the transactions handled correctly? Are any transactions lost or destroyed and, in that case, do we know that?

If back-up functionality for the system exists, it must be tested:

- Manual procedures, documentation, where is the back-up data stored?
- How can we be sure that the system works

gitlab.com incident - Problems encountered

- Disk snapshots in Azure are enabled for the NFS server, but not for the DB servers.
- The synchronisation process removes webhooks once it has synchronised data to staging. Unless we can pull these from a regular backup from the past 24 hours they will be lost
- The replication procedure is super fragile, prone to error, relies on a handful of random shell scripts, and is badly documented
- Our backups to S3 apparently don't work either: the bucket is empty
- So in other words, out of five backup/replication techniques deployed none are working reliably or set up in the first place. We ended up restoring a six-hour-old backup.
- `pg_basebackup` will silently wait for a master to initiate the replication progress, according to another production engineer this can take up to 10 minutes. This can lead to one thinking the process is stuck somehow. Running the process using "strace" provided no useful information about what might be going on.

The biggest problem with recovery testing is how to generate faults that causes a recovery. A power cut is easy to simulate but it is hard to simulate faulty data without affecting the rest of the system.

If commands exist to initialize recovery/restart, they must be tested. What happens to the ongoing transactions during a recovery? Are the transactions and the information about the transactions handled correctly? Are any transactions lost or destroyed and, in that case, do we know that?

If back-up functionality for the system exists, it must be tested:

- Manual procedures, documentation, where is the back-up data stored?
- How can we be sure that the system works

2.8 - Testování integrace systémů

Účel

- Integrace a testování kompletního systému v jeho pracovním prostředí
 - Komunikační mezivrstva (middleware)
 - Ostatní interní systémy
 - Externí systémy
 - Intranet, internet
 - Balíky třetích stran
 - Atd.

Many computer systems interact with other computer systems in one way or another. Integrating the system in its working environment is therefore an important activity. Naturally the type of integration depends among other things on what type of system we are building and what type of interaction with other systems we are talking about.

System integration testing is called integration testing in the large (BS 7925-1).

Some examples of surrounding systems that may be candidates for integration with the own system are:

- *Communication middleware* – whenever or system should be connected to standard networks
- *Internal systems* – other systems located

Testování integrace systémů - Strategie

- Po jednom
 - Integrujte vždy jeden další systém v jednom kroku
 - Integrujte jedno rozhraní v jednom kroku
- Pořadí integrace je důležité
 - Nejkritičtější systém první
 - Ale mějte v potaz externí závislosti

Just like in component integration testing, big bang integration seldom pays off. The same applies for system integration testing. Fault localization is harder, and the re-testing after fault fixes are more expensive. The “one thing at a time” strategy can be applied both on system and interface level, but depends as been previously mentioned on what type of system we are working with.

Another similarity with component integration testing (and testing in general) is that we want to integrate the most important systems (execute the most important test cases) first. The two main reasons are:

- If testing has to be aborted prematurely we have performed the most important activities

Testování integrace systémů - Vliv na plánování

- Zdroje
 - Lidské, nástroje, vybavení, čas
- Spolupráce
 - Externí zdroje
 - Prostředky pro komunikaci
- Časový plán projektu
 - Jaký je dopad integračního plánu na celkový plán projektu?

System integration testing often requires knowledge in areas where testers might not be strong. Often needed are for instance communication experts. For the same reason unusual tools and other equipment are needed. Further, system integration testing is often time consuming regardless of integration order. The two main reasons are that the systems that are being integrated are usually quite large and treated as black-boxes making faults very hard to locate and when a fault has been located, turn-around times for error correction are usually long. All these factors suggest that system integration testing is complex and hard to plan and much effort should be spent on planning activities.

Unless all systems to be integrated are

2.9 - Akceptační testování

Co je akceptační testování?

“Formální testování prováděné pro umožnění uživateli, zákazníkovi nebo jiné autorizované osobě rozhodnout zda je systém/komponenta akceptována či nikoliv” – BS 7925-1

- Uživatelské akceptační testování (User Acceptance Testing)
- Akceptační testování kontraktu (Contract Acceptance Testing)
- Alfa a Beta testování

Acceptance testing is similar to system testing, except that customers are present or directly involved. Acceptance testing can be a repeat of (or usually a subset of) the same tests used for system testing or can employ tests developed entirely by customers. In the latter case, it would be prudent to ask your customer for those tests in advance so that you can run as many of them as possible as part of your system testing activity.

Alpha and Beta testing is applied for large-scale products usually used by general public. Prerelease versions are provided to a subset of the future customers.

3 – Techniky statického testování

- 3.1. – Revize a testovací proces
- 3.2. – Typy revizí
- 3.3. – Statická analýza

QA testing techniques can be divided into 2 groups:

- Static Testing Techniques

When no code is executed, e.g.:

- Review (informal, walkthrough, peer review, inspection), usually performed manually on documents
- Static analysis (usually supported by tools for analyzing the code), usually tool supported analysis of code

- Dynamic Testing Techniques

When the code is executed, usually test cases are applied, e.g.:

- Black-box (functional)
- White-box (structural)
- Error-Guessing

3.1 – Revize a testovací proces

Proč dělat revize (reviews)?

- Úspora nákladů
- Srovnání:
- Prevence problémů
- Brzké zapojení členů týmu
- Efektivní učení

Provedení testu		Přísná revize	
Zpráva o incidentu:	0.5 h	Osoby:	4 osoby
Ladění, oprava:	1 h	Revidování:	7 h
Znovusestavení, dodání:	0.5 h	Schůze:	3 h
Opakované testování a regresní testování:	1 h		
Zpráva:	0.5 h	Počet nalezených vad:	11
Celkem:	3.5 h	Celkem:	40h
	x11		

05/04/22

TSK

146

- Cost effective

It is simpler and cheaper to correct faults early because:

- Fewer people involved
- Less travel needed
- Faster test execution
- Less time required for maintenance
- Less debugging needed
- Problem prevention
 - Find the faults before they are implemented
 - Reduce the risk of misunderstanding
 - Will the product become testable?
- Involve members early

Project members will get a common view of the project when involved in reviews

- Effective learning

When reviewing you get a better knowledge

Co revidovat?

- Požadavky zákazníka
- Specifikaci
- Analýzu a návrh
- Kód
- Uživatelskou příručku
- Plán testů
- Specifikaci testů (Testovací případy - Test Cases)

05/04/22

TSK

147

Each development phase is a translation from previous phase and it creates a work product that can be tested to see how successful the translation is. At the early stage of the development lifecycle is the majority of all defects (about 50% of all defects comes only from user requirements) and a review process should prevent this defect migration. The cost of defect migration to the next stage is much higher than finding defect in the phase where it was introduced. At the next stage, it can cost an order of magnitude more and an order of magnitude more again at the stage after that. The cost is maximized if the error is detected after the product is shipped to the customer and minimized if it is detected in the phase where it was introduced.

- User Requirements

The purpose of the requirements phase is to ensure that the users' needs are properly understood before translating them into design. In *User Requirements* the purpose, scope and performance of the required deliverable are defined. Acceptance testing is performed against user requirements.

- System Specification

The functional design is the process of translating user requirements in to the set of external interfaces. In *System Specification* the scope, characteristics and performance criteria of the system in terms of hardware and software, that meet the user requirements, are defined. System Testing is performed against System Specification.

- Design

The internal design is the process of translating the system requirements into a detailed set of data structures, data flows, and algorithms. In *Design* the most appropriate physical solution, positioning against existing architecture and applications to meet the agreed system requirements are specified. Component Testing is performed against the internal design (Architectural Design, Module Design Specifications, Database Schema, etc.).

- Code

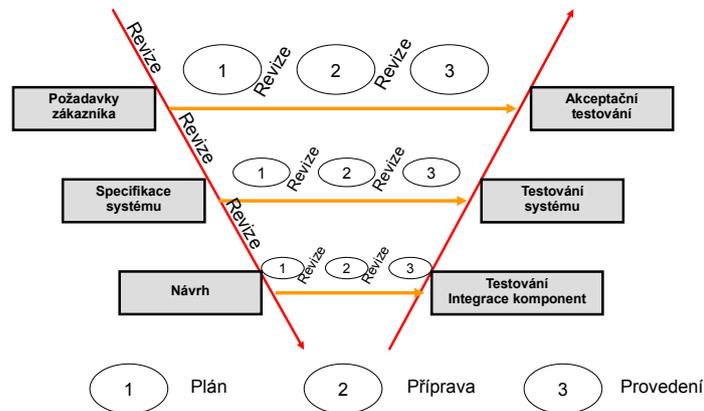
Coding is the process of translating the internal design specification into a specific set of code. We are checking:

- Data reference errors
- Data declaration errors
- Computation errors
- Comparison errors
- Control flow errors
- Interface errors
- Input/Output errors
- Portability

- User's Guide

Every software product that is delivered to a customer consists of both the executable code and the user manuals. A product's documentation is no less important than its code, because its quality is a significant factor in the success or failure of a product. From the point of view of the user, if the manual says do something, and the user follows these instructions and it doesn't work, it doesn't really help that the code is in fact right.

Revize v testovacím procesu



05/04/22

TSK

148

The most important review should be held in the beginning of the project (in the specification and design phases).

Plan, prepare and perform (1-3 steps in the picture above) are the phases of test development and execution.

When to review?

- As early as possible

When there is anything that is meaningful to review.

- Often

Because it is easier to review small documents or parts of documents than to review a big document. Therefore the review should be held often and give fast feedback to the authors.

3.2 – Typy revizí

Techniky revizí

- Neformální revize
- Procházení (Walkthrough)
- Technická (peer) revize
- Inspekce

Cíl a účel

- Verifikace
- Validace
- Shoda
- Zlepšení
- Nalezení vady

05/04/22

TSK

149

Review Techniques

- Informal review

An informal review (buddy checks) is better than none review, provided it is performed by someone else than the author and its objective is to detect defects.

- Walkthrough

A walkthrough is a formal review where the author explains his/her thoughts when going through the document. It could be more or less formal.

- Technical (peer) review

A technical review is a review made by technical experts (colleague, peer). It could be more or less formal.

- Inspection

An inspection is a group review quality improvement process for written materials. It consists of two aspects: product (document itself) improvement and process improvement (of both document production and inspection). The central “event” in this method is an inspection meeting at which defects in the product under review are discovered and discussed.

Goals and Purposes

Depending on the goals and the purposes, different review techniques are most suitable.

- Verification

The goal is to check that the reviewed document is correct. Are we building the system right?

- Validation

The goal is to check that the reviewed document is correct according to customer needs. Are we building the right system?

- Consensus

The purpose is to achieve a common view.

- Improvements

The purpose is to find improvement suggestions. One of the outputs from a review is to find technical improvements, improvements to development – and to the review process.

- Fault finding

The purpose is to find faults. The most important output are the faults found.

Jak revidovat?

- Rozdělit velký dokument na menší části
- Vzorky
- Revizi nechte provádět členy týmu na různých pozicích
- Vyberte vhodný typ revize

- *Split big documents into smaller parts*

It is more efficient to split the document into smaller parts. The reviewer can then manage to review the whole part and the remaining parts of the document can be reviewed by other reviewers.

- *Samples*

This is one way to reduce the effort of reviewing a large document. E.g. you can pick out some of the most important pages or pick out every tenth page of the document and call for a formal review. Every reviewer are reviewing the same pages.

- *Let different types of project members review*

Different people find different things because they have different views.

- *Select a suitable review type*

Neformální revize

- Nedokumentovaná
- Rychlá
- Pár definovaných procedur
- Užitečná ke zjištění zda autor postupuje správným směrem

An informal review (buddy checks) is better than none review, provided it is performed by someone other than the author and its objective is to detect defects. However, simply giving a document to someone else and asking them to look at it closely will turn up defects we might never find on our own.

- Undocumented

Undocumented review process.

- Fast

You cover the whole document. Many pages per hour.

- Few defined procedures

Since the informal review is undocumented, the defined procedures for how the review shall be carried out is usually few (or non at all). The normal situation is that no entry criteria are enforced, no roles are defined, no required skills are required and the agenda is informal.

This sometimes lead to much discussions and meetings that are held for hours resulting in low productivity. This is the main danger with informal reviews.

- Useful to check that the author is on track

It gives feedback to the author about the document and checks that it proceeds in the right way.

It may help the author to gather information, find the requirements and evaluate implementation ideas, when this cannot be achieved by other means, such as the study of source documents, requirements modelling, developers' meetings, etc.

Formální revize

- Plánování
- Dokumentovaná
- Pečlivá
- Zaměřená na určitý účel

Different types of reviews could be more or less formal. Some of the following properties should be included.

- Planning

Time for reviews should be included in the project plan. Each review is rigorously planned.

- Documented

The review process is documented. Here you find specified rules about the review. So the review tasks should be possible to run in the same way from one time to another. The more rules are specified for the review, the more formal it is.

- Thorough

The document which is being reviewed is checked against other documents to verify

Procházení (Walkthrough)

- Autor vysvětluje své myšlenky při postupném procházení dokumentu
- Autor je dobře připraven
- Suché běhy (zkouška nasucho)
- Posluchači (kolegové ze stejné skupiny) jsou nepřipraveni
- Shoda, „prodat to“

A walkthrough is a formal review where the author explains his/her thoughts when going through the document. It could be more or less formal. A company that has regular walkthroughs may have made their own rules for walkthroughs. This has to be seen as a formal walkthrough. The disadvantage of walkthrough is that a review tends to be less objective when the author is the producer.

- *The author well prepared*

The author knows the subject well and is prepared for questions.

- *Dry runs*

Dry run is a manual execution.

Example 1: Pretend to be a computer and manually walk through the execution of a program or a scenario.

Technická (Peer) revize

- Nalezení technických vad / řešení
- Bez účasti managementu
- Nalezené vady a nabídnutá řešení jsou dokumentována
- Techničtí experti

A technical review is a review made by technical experts (colleague, peer). It could be more or less formal. An example of a technical review is the CCB (Configuration Control Board) where problem reports are investigated. One of their purpose are to decide if the problem / fault that is found should be corrected now, later or not at all.

- Find technical faults /solution

The purpose is to find the best technical solution.

- Without management participation

Easier for some people to contribute in the meeting if the management do not participate. Management often does lack the technical depth.

 Found faults and proposed solutions get

Inspekce

- Definuje adaptovatelný formální proces
 - Seznam bodů
 - Pravidla
 - Metriky
 - Vstupní/Výstupní podmínky
- Definuje role
- Definuje entity k vytvoření(Deliverables)
- Dva příklady procesu inspekce nadefinovali: Fagan a Gilb/Graham

05/04/22

TSK

155

An inspection is a group review quality improvement process for written materials. It consists of two aspects: *product* (document itself) *improvement* and *process improvement* (of both document production and inspection). The central “event” in this method is an *inspection meeting* at which defects in the product under review are discovered and discussed.

- Defined adaptable formal process

Defined rules and checklists govern the work in an inspection. Metrics are collected for quality evaluation of both the inspected object and the inspection process itself. If metrics indicate potential improvements in the process, this is possible to an adaptation mechanism built-in in the process. Entry and exit criteria are enforced on the inspected object to allow for good and stable quality in the inspection process.

- Defined Roles

Each participant in the review can have one or more roles in the process (moderator, author, reviewer, manager, review manager).

- Defined Deliverables

The inspected (improved?) object is the most important delivery but there are other types of deliverables as well (discovered faults/problems, updated documents, process improvement proposals, consensus).

- Fagan and Gilb/Graham

There are two well known inspection processes one from Fagan and one from Gilb/Graham, see reference list.

Inspekční proces

- Plánování
- Přehledové (výkop) setkání
- Příprava (individuální)
- Revizní setkání
- Editování
- Pokračování
- Metriky
- Zlepšení procesu

05/04/22

TSK

156

- Planning
The inspection leader often makes the view plan. The review plan includes when, what and how to review, who should perform the review in what roles, etc.
- Overview (kick-off) meeting
The overview meeting is generally held one week before the review meeting to give the participants enough time to conduct the preparation.
Before the overview meeting entry criteria are enforced on the inspected object to see that it meets minimal quality standards required for the inspection to be worthwhile, e.g. that spell checker has been used.
During the overview meeting, the participants are given the inspection object, they are informed of their roles, available time, applicable checklists, etc.
 - Preparation (individual)
Self-study document. How to review, should be explained by the rules of the inspection process.
 - Review meeting
At the start of the review meeting entry criteria for the participants are enforced, i.e. have the participants performed the expected tasks.
Problems found are logged but not examined (discussions should be taken afterwards to reduce time consumption).
The protocol also includes time consumption for meetings as well as individual preparation. This information is later used for evaluation of the results.
 - Editing
When the problems have been solved or explained, the author corrects the document.
 - Follow-up
To check that the corrections are made and the statistics are logged (how many faults found, how much time spent).
This is done by the inspection leader.
 - Metrics
Metrics are recorded and the results are then used for process improvements or delivery decisions. Delivery decisions are taken based on the results of the inspection compared with predefined exit criteria. If a stable inspection process is used, the inspection effectiveness can be calculated, which in turn may provide means for estimating the number of remaining faults in the inspected document. The acceptable number of remaining faults in the document may be part of the exit criteria.
 - Process improvements
 - Feedback to the development process. Education? Lack of decisions?
 - Feedback to the review process. Was the time optimally spent?

Inspekce – Role a zodpovědnosti

- Moderátor
- Autor
- Revizor/ inspektor
- Manažer
- Manažer revizí / vedoucí inspekcí

- Moderator

A moderator trained in the inspection technique conducts the meeting. This role exists only during the meeting.

- Author

The author of the document under review is usually responsible for the investigation of the discovered problems and for carrying out the necessary changes.

-  Reviewer / inspector

The reviewers main responsibility is preparing themselves for the review meeting, i.e. searching for faults and unclear issues while using the allocated time for this task.

-  Manager

The main responsibility of the manager is to provide resources (time as well as people

Inspekce – Nástrahy/rizika

- Nedostatek zkušeností
- Nedostatek dokumentace
- Nedostatek podpory managementu

- Lack of training

People don't have enough knowledge in the review technique. Mentoring and training in review techniques are possible solutions.

- Lack of documentation

- The review process is poorly documented
- The entry criteria are weak
- Source documents are not approved

-  Lack of management support

Too little time is allowed to spend on reviews.

Possible solutions on this problem could be:

-  Ask him/her when he/she wants the problems to be discovered. Explain that the cost increases during the development time.

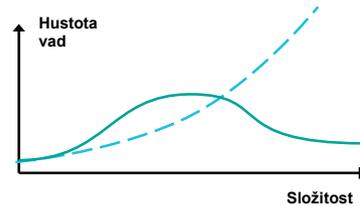
-  Include an estimated time for reviews in the time plan. Check according to earlier

3.3 – Statická analýza

Statické analýzy

“Analýza programu vykonávaná bez spouštění programu.” – BS 7925-1

- Nedostupný kód
- Parametry – záměna typů
- Možné překročení hranic pole (array bound violations)
- Vady nalezené kompilátorem
- Složitost programu



05/04/22

TSK

159

- Unreachable code

Part of a code that you can't reach, e.g. uncalled functions or procedures. Also called *dead code*.

- Parameter type mismatches

E.g. a variable declared with one type is sent to a procedure, but the procedure expects a variable of another type.

- Possible array bound violations

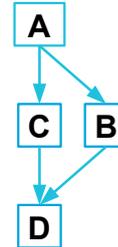
Trying to access an element index outside the boundary value of the array.

- Faults found by compilers

Fault types found by compilers depend first of all on the language – what is legal in it. For example, data type mismatches, missing files, possible division by 0, ranges without stop value, misuse of variables.

Statická analýza

- % změněného zdrojového kódu
- Grafická reprezentace vlastností kódu:
 - Graf toku (Control flow graph - CFG)
 - 1: (A) `int n = read_num();`
 - 2: (A) `if(n % 2 == 0){`
 - 3: (B) `System.out.println(n + " is even.");`
 - 4: (C) `} else {`
 - 5: (C) `System.out.println(n + " is odd.");`
 - 6: (D) `}`
 - Strom volání
 - Sekvenční diagramy
 - Třídní diagramy



05/04/22

TSK

160

- % of the source code changed

Some tools can analyse and tell how many % of the source code have been changed and which parts that have been changed => input to test case generation.

- Graphical representation of code properties

Depends on development tools features.

Analýza datových toků

- Zaměřeno na používání dat (lépe se analyzuje sekvenční kód)
- Příklady:
 - Přístup k proměnné po tom co byla uvolněna (ukazatel)
 - Přístup k proměnné předtím než byla definována
 - Definice bez použití

<pre>if(b > c){ a=3; a=5; System.out.println(a); }</pre>	<pre>a=3; if(a < 3){ b=7; System.out.println(b); }</pre>
---	---

05/04/22

TSK

161

Note: Not to be confused with *data flow testing* which is a dynamic test case selection method.

- Considers the use of data

How are the variables used through the code?

- Definitions with no intervening use

```
IF B > C THEN  A = 3;
```

```
  A = 3; IF A < 3 THEN
```

```
  A = 5;    B = 7;
```

```
  Print A;  Print B;
```

```
END;      END;
```

- Attempted use of a variable after it is killed

For example an attempt to read a variable outside its scope.

- Attempted use of a variable before it is

Statické metriky

- McCabova Cyclomatická míra složitosti (McCabe's Cyclomatic complexity)

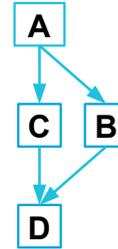
$$M = E - N + 2P$$

E = počet hran

N = počet uzlů

P = počet grafových komponent

- Řádky kódu (Lines of Code - LoC)
- Fan-out a Fan-in
- Úroveň vnoření



05/04/22

TSK

162

- McCabe's Cyclomatic complexity

Is defined as the number of decisions in a program or control flow graph + 1.

$M = E - N + 2P$ where

M = cyclomatic complexity

E = the number of edges of the graph

N = the number of nodes of the graph

P = the number of connected components.

Implications for Software Testing

M is a lower bound for the number of possible paths through the control flow graph.

M is an upper bound for the number of test cases that are necessary to achieve a complete branch coverage.

- Lines of Code (LoC)

Lines of code. It's a common measurement of the size of a program.

- Fan-out and Fan-in

Fan-out is the amount of modules a given module calls. Modules with high Fan-out are often found in the upper part of the call tree.

Fan-in is the amount of modules that call a specific module. Modules with high Fan-in are often found in the lower part of the call tree.

If a module has both high fan-in and fan-out, consider to redesign it.

- Nesting levels

For example many IF-statements nested into each other get a deep nesting level. This means that the code is difficult to understand. It is even worse when the cyclomatic complexity is also high.

One nesting level:

```

IF X > 5 THEN
  PRINT "BIG";
ELSE
  PRINT "SMALL";
ENDIF;
  
```

Who nesting levels:

```

IF X > 5 THEN
  IF X < 10 THEN
    PRINT "BIG UNIT";
  ENDIF;
ELSE
  IF X != 0 THEN
    PRINT "SMALL UNIT";
  ENDIF;
ENDIF;
  
```

4 – Dynamické testovací techniky

- 4.1. – Testování Black a White box
- 4.2. – Techniky Black box testování
- 4.3. – Techniky White box testování
- 4.4. – Testová data
- 4.5. – Tipování vad (Error Guessing)

This part deals with dynamic testing techniques – methods that use executable test cases. These techniques are further divided into two groups (white-box and black-box testing techniques).

4.1 – Testování Black- a White-box

- Strategie
 - Co je účelem testování?
 - Co je cílem testování?
 - Jak dosáhnout cíle?
- Metody výběru testovacích případů (Test Case)
 - Který testovací případ bude proveden?
 - Je to dobrý reprezentant pro všechny možné testovací příklady?
- Kritéria pokrytí
 - Jak mnoho kódu (požadavků, funkcionality) je pokryto?

A good way of dealing with a testing problem is to first clarify the purpose of testing, then to define a goal and finally to develop a strategy for how to reach the goal.

Once the goal has been defined, a test case selection strategy can be constructed. The obvious strategy would be to test everything, but due to infinite possibilities of choosing input this strategy is simply not feasible. Thus we need to carefully select the test cases that are to be executed. These test cases should be good representatives of all the possible test cases. To simplify the selection there exists a large number of test case selection methods, most of them are associated with coverage criteria to determine when to stop testing.

Metody výběru testovacích případů (Test Case)

- White-box / Strukturální / Logic driven
 - Založeno na implementaci (struktura kódu)
- Black-box / Funkcionální / Data driven
 - Na základě požadavků (funkcionální specifikace, rozhraní)



Test cases for dynamic execution are usually divided into two groups depending on the source of information used for creating the test case.

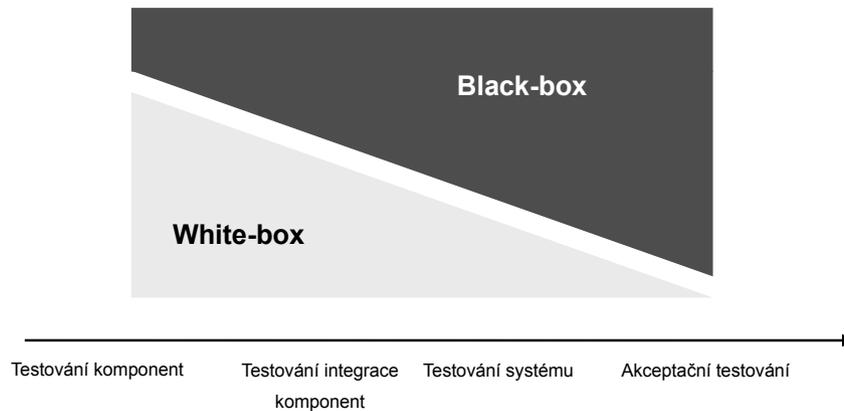
- White-box

Test cases are based on information about the implementation of the test object (structure of the code). The inputs of white-box test cases are generated from the implementation information (from the code). The testing is based on the program logic.

- Black-box

Test cases are aimed at testing the functionality of the test object. The inputs of black-box test cases are taken either from the requirements or from a model created from the requirements. Testing is based on inputs and

Důležitost testovacích metod



05/04/22

TSK

166

The two types of test cases are used a little bit differently in the development lifecycle. White-box test cases are mostly used in the early test phases of the development lifecycle and are of less usage higher up in the testing hierarchy.

There are two reasons for this:

1. The most important is that most white-box methods require extensive knowledge of the code and other parts of the implementation. Later test phases are usually performed by dedicated test specialists with neither deep implementation knowledge nor access to this information.
2. The other reason for not using white-box test case selection methods in later test stages is related to coverage. White-box test cases are usually more fine grained than black-box test cases. Fine grained test case selection methods require a large number of test cases in order to reach high coverage.

Black-box testing techniques are used throughout the development lifecycle. The main advantage with black-box testing techniques is that they only depend on the requirements, which means that test cases can be prepared before the implementation is complete.

Both methods are important. If only white-box testing would be performed, some requirements are not tested (performance requirements). On the other hand if only black-box test cases are used, some parts of the code might remain untested (special features called when a certain value is entered in a certain cell).

Měření pokrytí kódu

- Kolik kódu bylo provedeno?
- Metody pokrytí kódu:
 - Pokrytí segmentu
 - Pokrytí dvojic volání

$$\text{Pokrytí kódu} = \frac{\text{Provedené segmenty kódu/dvojice volání}}{\text{Všechny segmenty kódu/dvojice volání}}$$

- Podpora nástrojů:
 - Často jsou vhodné
 - Pro white-box testování jsou skoro nezbytné

05/04/22

TSK

167

Code coverage metrics respond the question – How much of the code is being executed?

There are usually 2 metrics:

- Segment coverage

A segment is a set of program statements that are executed unconditionally or executed conditionally based on the value of some logical expression. 85% is a practical coverage value.

- Call-pair coverage

A call pair is an interface whereby one module invokes another. Call-pair coverage is especially useful integration testing to ensure that all module interfaces are exercised. 100% is a practical coverage value.

As already has been mentioned, white-box testing techniques use implementation information to derive the input part of the test cases. Most often some aspect of the code, for instance the source code statements, is used for this purpose. Even with quite small programs, the task of keeping track of which statements that have already been tested and which statements that yet remain to be tested is quite difficult. The solution to this problem is to use a tool. There are a large number of commercial code coverage tools available for this. They all work in the same manner: before the source code of the object to be tested is compiled, the code is instrumented by adding extra instructions at strategic places in the original code. This is done by the tool.

The source code with the extra instructions is then compiled as usual and test cases are then executed in the normal way. The added instructions continuously log the progress of the testing and from the results of the logging instructions the tool can calculate which parts of the code that have been executed. Obviously the extra inserted instructions consume execution resources thus distorting performance measurements, so this type of tool is not appropriate during system testing.

Nevertheless, the use of such tools increase both the quality and the productivity of the testing in the earlier test phases.

Testování na základě požadavků

- Jak mnoho vlastností produktu je pokryto testovacími případy (Test Case)?
- **Metrika pokrytí požadavků:**

$$\text{Pokrytí požadavků} = \frac{\text{Testované požadavky}}{\text{Celkový počet požadavků}}$$

- Jaký je průběh testů?
- **Metrika pokrytí testů:**

$$\text{Pokrytí testů} = \frac{\text{Provedené testovací případy}}{\text{Celkový počet testovacích případů}}$$

The basic for all black-box testing is the requirements.

The simplest but still structured way of creating test cases is to write one test case for each requirement. The main drawback with this approach is that most requirements require more than one test case to be tested thoroughly, and different requirements require a different amount of test cases. In this case we can create the coverage matrix that tracks requirements to test cases and vice versa. This feature is usually included in test management tools.

Requirement Coverage responds the question: How much of the product's features is covered by test cases?

Test Coverage responds the question: What's

Vytváření modelů

- Vytváření modelů – obecné vlastnosti
 - Používají se k organizaci informací
 - Často se odhalí vady už při tvorbě modelu
- Testování na základě modelů
 - Testovací případy jsou získány z modelů
 - Příklady
 - Testování syntaxe, Testování přechodu mezi stavy, Testování na základě diagramu případu užití
 - Pokrytí je určeno použitým modelem

A more elaborate way of creating black-box test cases is to transform a set of requirements into a model of the system and derive the test cases from the model instead of directly from the requirements.

In most model-based testing techniques there are well defined coverage criteria which are simple to calculate and interpret.

The main drawbacks with models are limited scope and validation. Often the purpose of the model and the modeling technique used, limits the scope of the model. For instance a syntax graph only captures the syntax of a language. The semantic of that language must be covered somewhere else. The result is that several models need to be developed and used in order to get a reasonable

Metody Black-box testování

- Ekvivalentní rozdělení
- Analýza hraničních hodnot
- Testování přechodu mezi stavy
- Graf příčin a následků (Cause-Effect Graphing)
- Testování syntaxe
- Náhodné testování

- Cause-Effect Graphing

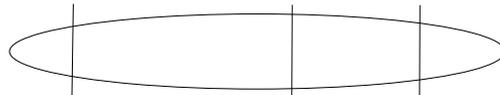
A model based method, which relates effects with causes through Boolean expressions. The main focus is on different combinations of inputs from the equivalence classes. Cause-effect graphing is a way of doing this whilst avoiding the major combinatorial problems that can arise.

- Syntax Testing

A model based method, which focuses on the syntax or rules (how different parts may be assembled) of a language (used during implementation). This method generates valid and invalid input data to a program. It is applicable to programs that have a hidden language that defines the data. Syntax generator is needed.

Ekvivalentní rozdělení

- Identifikují se množiny vstupních hodnot, za předpokladu, že testovaný systém zachází stejně se všemi hodnotami v dané množině
- Vytvoří se jeden testovací případ pro každou identifikovanou množinu (třídu ekvivalence - equivalence class)
- Nejzákladnější testovací technika



05/04/22

TSK

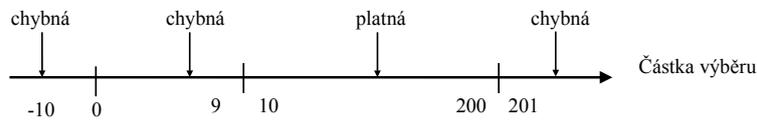
171

Equivalence class partitioning is one of the most basic black-box testing techniques. The underlying idea is that the input domain can be divided into a number of equivalence classes. The characteristic of an equivalence class is the assumption that all values belonging to that class are handled in exactly the same manner by the program.

If this assumption is true, then it would suffice to select one single test case for each equivalence class, since multiple test cases from the same equivalence class would repeat the same test.

Coverage is measured by dividing the number of executed test cases, i.e. the number of tested equivalence classes by the total number of equivalence classes.

Ekvivalentní rozdělení (Příklad)



	Záporná částka	Dělitelný 10 a menší nebo roven 200	Nedělitelný 10 a menší než 200	Více než 200
Dostatek peněz na účtu	1. Výběr zamítnut	2. Výběr <i>povoleno</i>	3. Výběr zamítnut	4. Výběr zamítnut
Nedostatek peněz na účtu	5. Výběr zamítnut	6. Výběr zamítnut	7. Výběr zamítnut	8. Výběr zamítnut

05/04/22

TSK

172

Example: “A withdrawal from an ATM (Automatic Teller Machine) is granted if the account contains at least the desired amount. Furthermore, the amount withdrawn must be an even number of 10 EUR. The largest amount that can be withdrawn is 200 EUR.”

By analyzing the requirements we find several different independent dimensions to this problem:

- Is there enough money in the account?
- Is the desired amount an even 10-number?
- Is the desired amount outside the correct 0-200 range?

One way to organize the information is to make a table as above. Each cell in the table represents an equivalence class, which means that there should be eight test cases to solve this testing problem with equivalence partitioning.

In this example one could argue that negative withdrawal is not technically possible, and even if it was possible, the amount of money in the account would be irrelevant.

This discussion illustrates two difficult questions: how much should we really test? And which tests are most important?

Mostly this boils down to a matter of taste. Our view is that it is better to include too much when designing test cases than to miss vital functionality. Test cases should however always be assigned a priority based on importance to the end user and importance to future testing.

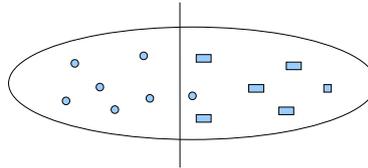
High priority test cases above could be 2, 3, 4 and 6.

Medium priority test cases above could be 7 and 8.

Low priority test cases above could be 1 and 5.

Analýza hraničních hodnot

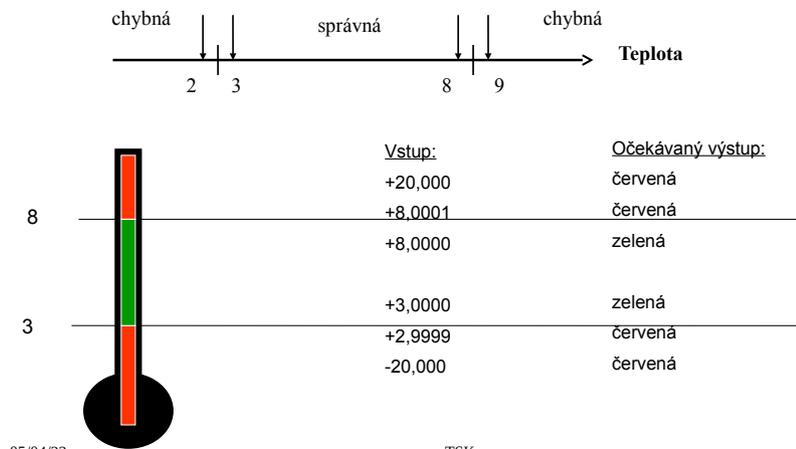
- Pro každou identifikovanou hranici ve vstupech a výstupech se vytvoří dva testovací případy. Jeden na každé straně hranice, tak blízko hranici jak jen to je možné.



Boundary Value Analysis is a refinement of equivalence class partitioning. Instead of choosing any representative from each equivalence class, interest is focused around the boundaries of each class. The idea is to select one test case for each boundary of the equivalence class. The properties of a test case is thus that it belongs to a defined equivalence class and that it tests a value that is preferable on, or at least reasonably close to one of the boundaries of the equivalence class.

The main reason why boundaries are important is that they are generally used by programmers to control the execution of the program, for instance through if- or case-statements. Since the boundaries are being

Analýza hraničních hodnot (Příklad)



05/04/22

TSK

174

Example: "A refrigerator has a red and a green indicator. The optimal temperature in the refrigerator is between +3 an +8 degrees. If the temperature is within this interval, the green indicator is lit, otherwise the red indicator is lit."

The temperature range can be divided into three intervals (equivalence classes).

1. From $-\infty$ (-273?) to but not including +3,0000 resulting in a red light
2. From +3,0000 to +8,0000 resulting in green light
3. From but not including +8,0000 to $+\infty$

When using boundary value analysis, there should be one test case for each boundary in every equivalence class:

Test case 1a:

Negative infinity, even -273 is a little hard to create, and furthermore not very likely to occur. So a good (?) estimation could be -20,000.

Test case 1b:

Here we have the problem of being close enough to the boundary since being on the boundary is outside this interval. Is five valid digits a good estimate?

Test cases 2a and 2b:

Both boundaries are inside the interval so these values are the ones to choose.

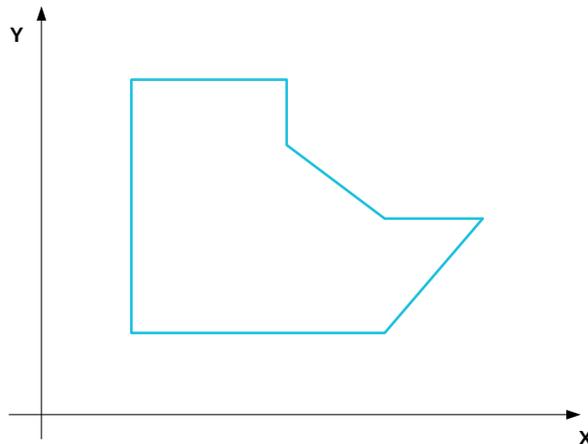
Test case 3a:

Same discussion as in 1b.

Test case 3b:

Same discussion as in 1a.

Analýza hraničních hodnot (Příklad)



05/04/22

TSK

175

Example: “A refrigerator has a red and a green indicator. The optimal temperature in the refrigerator is between +3 and +8 degrees. If the temperature is within this interval, the green indicator is lit, otherwise the red indicator is lit.”

The temperature range can be divided into three intervals (equivalence classes).

1. From $-\infty$ (-273?) to but not including +3,0000 resulting in a red light
2. From +3,0000 to +8,0000 resulting in green light
3. From but not including +8,0000 to $+\infty$

When using boundary value analysis, there should be one test case for each boundary in every equivalence class:

Test case 1a:

Negative infinity, even -273 is a little hard to create, and furthermore not very likely to occur. So a good (?) estimation could be -20,000.

Test case 1b:

Here we have the problem of being close enough to the boundary since being on the boundary is outside this interval. Is five valid digits a good estimate?

Test cases 2a and 2b:

Both boundaries are inside the interval so these values are the ones to choose.

Test case 3a:

Same discussion as in 1b.

Test case 3b:

Same discussion as in 1a.

Analýza hraničních hodnot (Rodné číslo)

Prvních šest čísel rodného čísla je vždy částí data narození dané osoby.

- První dvojčíslí je posledním dvojčíslím roku narození.
- Druhé dvojčíslí je měsícem narození (případně doplněné o úvodní nulu).
- Třetí dvojčíslí je dnem narození (případně doplněné o úvodní nulu).
- Po prvních šesti číslech zpravidla následuje lomítko "/", které se však při zápisu na technických nosičích často vynechává.
- Do 31.12.1953 se za lomítko přidávaly právě tři číslice. Ty vyjadřují pořadové číslo narozené osoby v daný den.
- Od 1. 1. 1954 se za lomítko kromě těchto třech číslic přidává ještě čtvrtá číslice, sloužící jako kontrolní číslice.
 - Kontrolní číslice se doplní tak, aby celé rodné číslo bylo beze zbytku dělitelné jedenáctí. Na posledním místě rodného čísla je tak zbytek po dělení devítimístného rodného čísla číslem jedenáct. Z tohoto pravidla existuje výjimka. Pokud je zbytek po dělení devítimístného čísla roven deseti (a neexistuje žádná kontrolní číslice, která by splňovala předchozí podmínku), jako kontrolní číslice se použije nula (a celé rodné číslo pak dělitelné jedenáctí není). Tato výjimka byla použita zhruba u tisícovky rodných čísel a přidělování takových rodných čísel bylo roku 1985 podle interního předpisu Federálního statistického úřadu č. Vk. 2898/1985 ukončeno.

Example: "A refrigerator has a red and a green indicator. The optimal temperature in the refrigerator is between +3 and +8 degrees. If the temperature is within this interval, the green indicator is lit, otherwise the red indicator is lit."

The temperature range can be divided into three intervals (equivalence classes).

1. From $-\infty$ (-273?) to but not including +3,0000 resulting in a red light
2. From +3,0000 to +8,0000 resulting in green light
3. From but not including +8,0000 to $+\infty$

When using boundary value analysis, there should be one test case for each boundary in every equivalence class:

Test case 1a:

Negative infinity, even -273 is a little hard to create, and furthermore not very likely to occur. So a good (?) estimation could be -20,000.

Test case 1b:

Here we have the problem of being close enough to the boundary since being on the boundary is outside this interval. Is five valid digits a good estimate?

Test cases 2a and 2b:

Both boundaries are inside the interval so these values are the ones to choose.

Test case 3a:

Same discussion as in 1b.

Test case 3b:

Same discussion as in 1a.

Analýza hraničních hodnot (Rodné číslo)

Rozlišení mužů a žen

Obecně platí, že ženám se ke druhému dvojčíslí rodného čísla (tedy k měsíci narození) přičítá číslo 50. Od roku 2004 (zákonem č. 53/2004 Sb.) je zavedena možnost v případě, že jsou v nějaký den vyčerpána všechna platná čtyřčíslí, použít alternativní rodné číslo, u kterého mají muži k číslu měsíce přičteno číslo 20 a ženy 70.

Možné kombinace čísel

- 00-99 pro první dvojčíslí
- 01-12, 21-32, 51-62 a 71-82 pro druhé dvojčíslí
- 01-31 pro třetí dvojčíslí
- 000-9999 pro část za lomítkem

Regulární výraz

`[0-9]{2}[01235678][0-9][0-3][0-9]\v?[0-9]{3,4}`

Example: "A refrigerator has a red and a green indicator. The optimal temperature in the refrigerator is between +3 an +8 degrees. If the temperature is within this interval, the green indicator is lit, otherwise the red indicator is lit."

The temperature range can be divided into three intervals (equivalence classes).

1. From $-\infty$ (-273?) to but not including +3,0000 resulting in a red light
2. From +3,0000 to +8,0000 resulting in green light
3. From but not including +8,0000 to + ∞

When using boundary value analysis, there should be one test case for each boundary in every equivalence class:

Test case 1a:

Negative infinity, even -273 is a little hard to create, and furthermore not very likely to occur. So a good (?) estimation could be -20,000.

Test case 1b:

Here we have the problem of being close enough to the boundary since being on the boundary is outside this interval. Is five valid digits a good estimate?

Test cases 2a and 2b:

Both boundaries are inside the interval so these values are the ones to choose.

Test case 3a:

Same discussion as in 1b.

Test case 3b:

Same discussion as in 1a.

Analýza hraničních hodnot (Řetězce)

- Délka min/max
- Prázdný
- Bílé znaky- mezery, tabulátory, zalomení řádku
- Oddělovače – středník, čárka, dvojtečka, uvozovky, apostrofy
- Speciální znaky
- UTF-8 - čeština, čínština, ...

Example: “A refrigerator has a red and a green indicator. The optimal temperature in the refrigerator is between +3 and +8 degrees. If the temperature is within this interval, the green indicator is lit, otherwise the red indicator is lit.”

The temperature range can be divided into three intervals (equivalence classes).

1. From $-\infty$ (-273?) to but not including +3,0000 resulting in a red light
2. From +3,0000 to +8,0000 resulting in green light
3. From but not including +8,0000 to $+\infty$

When using boundary value analysis, there should be one test case for each boundary in every equivalence class:

Test case 1a:

Negative infinity, even -273 is a little hard to create, and furthermore not very likely to occur. So a good (?) estimation could be -20,000.

Test case 1b:

Here we have the problem of being close enough to the boundary since being on the boundary is outside this interval. Is five valid digits a good estimate?

Test cases 2a and 2b:

Both boundaries are inside the interval so these values are the ones to choose.

Test case 3a:

Same discussion as in 1b.

Test case 3b:

Same discussion as in 1a.

Analýza hraničních hodnot – srovnání

- Detekce vady v případě nejčastějších chyb :

Požadavek	Vada v imp.	ER (EP)	AHH (BVA)
$A < 18$	$A \leq 18$	Ne	Ano
$A < 18$	$A > 18$	Ano	Ano
$A < 18$	$A < 20$	Možná	Ano

- Počet testovacích případů (jedna dimenze) $BVA = 2 * EP$

Which is better, Equivalence Partitioning (EP) or Boundary Value Analysis (BVA)?

The answer depends on what we mean by better. Test cases made by BVA will catch more types of errors, but on the other hand there will be more test cases, which is more time consuming.

If you do boundaries only, you have covered all the partitions as well:

- Technically correct and may be OK if everything works correctly
- If the test fails, is the whole partition wrong, or is a boundary in the wrong place – have to test mid-partition anyway
- Testing only extremes may not give confidence for typical use scenarios (especially for users)

Cíle testování?

Podmínky	Platná část	Tag	Neplatná část	Tag	Platná strana hranice	Tag	Neplatná strana hranice	Tag

- Důkladný přístup: PČ(VP), NČ(IP), PSH(VB), NSH(IB)
- V časové tísni záleží na hlavním účelu testování
 - Minimálně - důvěra v uživatele: Pouze PČ(VP)?
 - Maximální nalezení vady: PSH(VB) nejprve (a dále NSH(IB)?)

Testování přechodu mezi stavy

- Funkční chování je namodelováno pomocí stavového automatu
- Vytvoření testovacích případů pro:
 - A) každý stav automatu.
 - B) použití každého přechodu automatu (0-přechodové pokrytí)
 - C) každou možnou posloupnost přechodů (n-přechodové pokrytí)
- Pokrytí
 - Záleží na vnitřní strategii

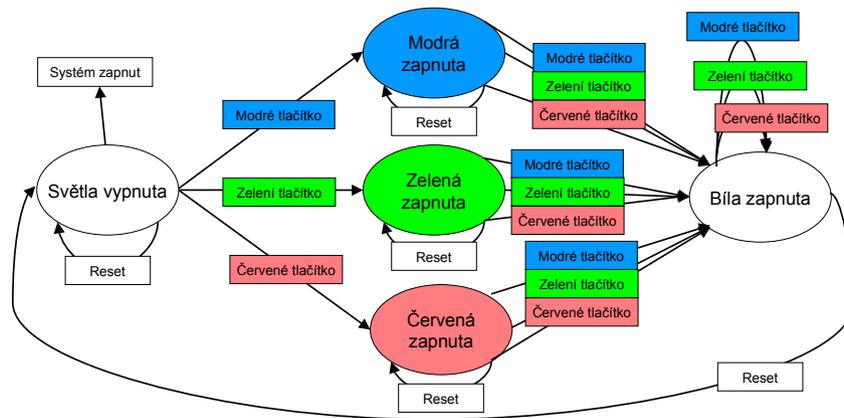
State machine based testing is a quite useful model based black-box testing technique, since any type of functionality that can be represented as a finite state machine can be tested using this technique.

The first step when using state machine testing is to construct the model itself. Sometimes, state machines are used by designers and constructors as implementation tools. In those cases, the state machines can of course be used directly. Otherwise the state machine model has to be constructed based on the requirements by the testers.

Often during construction of the state machine models, faults are found. One of the key properties with a state machine is that all input types can occur regardless of the state of the machine. If a state machine model previously has not been drawn, there are almost always disregarded combinations of state and input, which are very easily discovered when building the model.

When the model is finished, the next step is to construct test cases from it. There are several different strategies. The simplest and least powerful is to cover each state in the model at least once. As soon as there are more than one way of reaching a particular state, state coverage will most likely leave some transitions untested. A more elaborate strategy is therefore to focus on the transitions between the states. 0-switch coverage requires one test case for each possible transition in the model. 1-switch coverage requires a test case for every possible pair of consecutive transitions and finally n-switch coverage requires a test case for every possible n+1 consecutive transitions in the model.

Testování přechodů mezi stavy (příklad)



05/04/22

TSK

182

Example:

- Four keys, four lamps
- After the start, all lamps are off
- A colored key turns "its" lamp on, if all lamps are off
- Next colored key turns the white lamp on and the colored off
- The *Reset* key turns the white lamp off and resets the system

There are 5 states.

To determine how many transitions there are, it is helpful to calculate the number of transitions out from each state (in our case there are 4 transitions):

$(5 \cdot 4 + 1) = 21$ transitions (0-switch)

$(5 \cdot 4 \cdot 4 + 4) = 84$ pairs of transitions (1-switch)

It's easy to understand that 'time-outs', common in real-time applications, will make it even more advanced.

Create test cases:

- A) touching each state
 - 5 test cases – sufficient for such a simple system
- B) using each transition (0-switch coverage)
 - 21 test cases – if the white lamp did not turn on after the green lamp, it is necessary to use "each transition" to catch this fault
- C) using every possible *pair* of transitions (1-switch coverage)
 - 84 test cases – if the *Reset* key does not work after the red lamp and the blue key (but works after all other keys), finding this fault requires trying "all pairs of transitions"

To discover a fault which, for example, causes the system to hang after a thousand loops, still another strategy is required.

The number of tested inputs is another dilemma. Should all possible inputs be tried in each state? The strategy described here do not answer this question.

4.3. - White-box testování

- Vstupy testovacích případů jsou vždy odvozeny od implementace (kódu)
- Nejčastější informace potřebné k implementaci TC:
 - Výrazy
 - Rozhodovací bloky v kódu
 - Použití proměnných
- Očekávané výstupy pro testovací případy jsou získány vždy z požadavků!

When creating white-box test cases the basis is in the implementation. The input part of the test case is derived from the implementation. Commonly used implementation properties include code structure and how variables are used in the code. Less common but nevertheless interesting implementation properties are call-structures and process/object interactions.

Regardless of the white-box test method chosen, expected output is always extracted from the requirements and not from the implementation itself.

White-box Testovací metody

- Testování výrazů/instrukcí (Statement Testing)
- Testování skoků/rozhodování
- Testování toku dat
- Testování podmínek skoků
- Testování kombinace podmínek skoků
- Testování změny podmínky
- Testování LCSAJ

- Statement Testing

The idea with statement coverage is to create enough test cases so that every statement in the source code has been executed at least once

- Branch/Decision Testing

The idea with decision coverage is to execute every single decision in the code at least twice (both possible outcomes of the decision should be executed in order to reach full decision coverage)

- Data Flow Testing

Test cases are designed based on variable usage within the code

- Branch Condition Testing

A test case design technique in which test cases are designed to execute branch condition outcomes

Test cases shall be designed to exercise individual Boolean operand values within

decision conditions.

- Branch Condition Combination Testing

A test case design technique in which test cases are designed to execute combination of branch condition outcomes

Test cases shall be designed to exercise combinations of Boolean operand values

within decision conditions.

- Modified Condition Decision Testing

A test case design technique in which test cases are designed to execute branch condition outcomes that independently affect a decision outcome

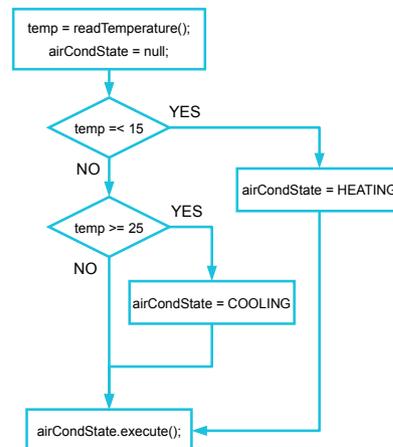
Test cases shall be designed to demonstrate that Boolean operands within a decision condition can independently affect the outcome of the decision.

- LCSAJ Testing

Linear Code Sequence And Jump (LCSAJ) – Select test cases based on jump-free sequence of code. It consists of the following three items: the start of the linear sequence of executable statements, the end of the linear sequence, and the target line to which control flow is transferred at the end of the linear sequence.

Graf kontrolního toku/vývojový diagram

```
public void doAirconditioning() {
    double temp = readTemperature();
    Aircondition airCondState = null;
    if(temp <= 15) {
        airCondState = Aircondition.HEATING;
    }
    else if(temp >= 25) {
        airCondState = Aircondition.COOLING;
    }
    airCondState.execute();
}
```



05/04/22

TSK

185

This is a small piece of code, which implements the temperature regulation. The function “adjust_temperature” is called without arguments. The first thing it does is to read the current temperature, and then depending on the value, either the heater is switched on, the cooler is switched on, or the system is left untouched. The global variable *control* holds the current setting of the heater and cooler.

To the right the is the corresponding control flow graph. To aid the understanding of the control flow graph strategic parts of the code may be inserted in the diamonds and boxes.

McCabe’s cyclomatic complexity measure: No. of diamonds + 1 (2 + 1 = 3) – it says that the more decisions there are in a piece of code,

Testování výrazů/instrukcí (Statement coverage)

- Proveďte se každý výraz/instrukce kódu alespoň jednou během provádění testovacích případů
- Vyžaduje nástroje
 - Instrumentace kódu
 - Měření
- Pokrytí

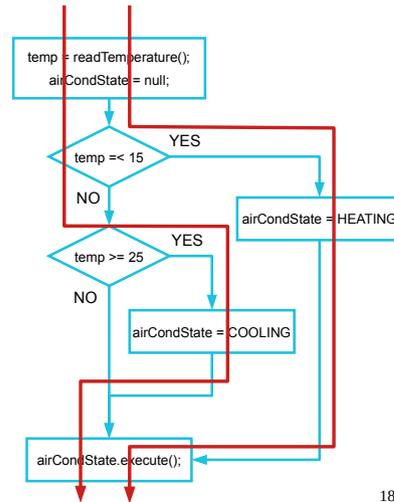
$$\text{Pokrytí výrazů} = \frac{\text{Provedené výrazy}}{\text{Celkový počet výrazů}}$$

Statement coverage is a fundamental white-box testing technique. This idea of statement coverage is to create enough test cases so that every statement in the source code has been executed at least once.

The workflow when using statement coverage is to first execute all existing black-box test cases that has been created while monitoring the execution. This monitoring is in all but the simplest test cases performed with tool support. When all black-box test cases have been executed, the tool can report which parts of the code that remain untested. The idea is now to construct new test cases that will cover as many of the remaining statements as possible. Start with the part of the code that should be reached, walk

Testování výrazů/instrukcí

```
public void doAirconditioning() {
    double temp = readTemperature();
    Aircondition airCondState = null;
    if(temp <= 15) {
        airCondState = Aircondition.HEATING;
    }
    else if(temp >= 25) {
        airCondState = Aircondition.COOLING;
    }
    airCondState.execute();
}
```



05/04/22

TSK

187

When creating test cases for statement coverage we can make use of the control flow graph. We know the statement coverage requires statements in the code to be executed. We also know that the boxes and the diamonds represent all the statements in the code.

By following the two blue arrows through the code we cover all the diamonds and all the boxes are covered and thus we have statement coverage (according to the relation with McCabe measure there should be three or less test cases and in this case two were enough).

By examine the relation we can now also deduce that in the optimal choice of test cases, number of test cases for decision

Testování skoků/rozhodování (Branch/Decision Testing)

- Vytvoří se testovací případy tak aby se každé rozhodnutí provedlo s výsledkem TRUE a FALSE
 - Ekvivalentní k provedení všech skoků
- Vyžaduje nástroje
 - Instrumentace kódu
 - Měření
- Pokrytí

$$\text{Pokrytí rozhodování} = \frac{\text{Provedené výstupy rozhodnutí}}{2 * \text{Celkový počet rozhodování}}$$

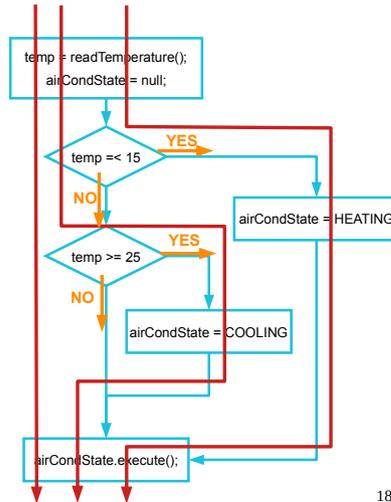
Branch coverage and decision coverage are two names for the same thing.

Decision coverage is a technique similar to statement coverage. The idea with decision coverage is to execute every single decision in the code at least twice. Both possible outcomes of the decision, i.e. true and false, should be executed in order to reach full decision coverage.

By the first glance statement and decision coverage seem to yield exactly the same test cases, since executing every decision with both true and false outcomes will result in all statements being executed, and in order to execute all statements all outcomes of every decision needs to be executed. However this is not entirely true. There is one case in

Testování skoků/rozhodování

```
public void doAirconditioning() {
    double temp = readTemperature();
    Aircondition airCondState = null;
    if(temp <= 15) {
        airCondState = Aircondition.HEATING;
    }
    else if(temp >= 25) {
        airCondState = Aircondition.COOLING;
    }
    airCondState.execute();
}
```



05/04/22

TSK

189

When creating test cases for statement coverage we can make use of the control flow graph. We know the statement coverage requires statements in the code to be executed. We also know that the boxes and the diamonds represent all the statements in the code.

By following the two blue arrows through the code we cover all the diamonds and all the boxes are covered and thus we have statement coverage (according to the relation with McCabe measure there should be three or less test cases and in this case two were enough).

By examine the relation we can now also deduce that in the optimal choice of test cases, number of test cases for decision

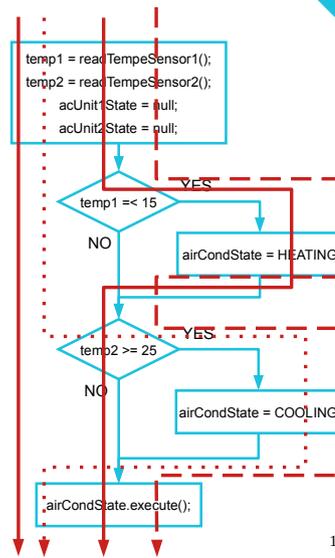
Testování pokrytí cest (Path coverage)

- Pokrytí všech možných průchodů kódem (kombinace jednotlivých rozhodnutí)
- Kód s cykly
 - testy všechny možné počty průchodu → nereálné
 - Doporučené testovat: 0 průchodů, 1 průchod, n průchodů
- Pokrytí

$$\text{Pokrytí cest} = \frac{\text{Počet testovaných cest}}{2^{\text{Počet rozhodnutí}}}$$

Testování pokrytí cest

```
public void adjustTemperature2() {
    double temp1 = readTempeSensor1();
    double temp2 = readTempeSensor2();
    Aircondition acUnit1State = null;
    Aircondition acUnit2State = null;
    if(temp1 <=15) {
        acUnit1State = Aircondition.HEATING;
    }
    if(temp2 >=25) {
        acUnit2State = Aircondition.COOLING;
    }
    acUnit1State.execute();
    acUnit2State.execute();
}
```



05/04/22

TSK

191

When creating test cases for statement coverage we can make use of the control flow graph. We know the statement coverage requires statements in the code to be executed. We also know that the boxes and the diamonds represent all the statements in the code.

By following the two blue arrows through the code we cover all the diamonds and all the boxes are covered and thus we have statement coverage (according to the relation with McCabe measure there should be three or less test cases and in this case two were enough).

By examine the relation we can now also deduce that in the optimal choice of test cases, number of test cases for decision

Testování toku dat (Data flow coverage)

```

01: public QResult quadratic(double a,
double b, double c) {
02:     double disc = b*b - 4*a*c;
03:     QResult r = new QResult();
04:     if(disc < 0) {
05:         r.isComplex = true;
06:     } else {
07:         r.isComplex = false;
08:     }
09:     if(!r.isComplex) {
10:         r.r1 = (-b + Math.sqrt(disc))/(2*a);
11:         r.r2 = (-b - Math.sqrt(disc))/(2*a);
12:     }
13:     return r;
14: }

```

line	kategorie		
	definice	c-use	p-use
1	a,b,c		
2	disc	a,b,c	
3	r.isComplex, r.r1, r.r2		
4			disc
5	r.isComplex		
6			
7	r.isComplex		
8			
9			r.isComplex
10	r.r1	a,b,disc	
11	r.r2	a,b,disc	
12			
13		r.isComplex, r.r1, r.r2	
14			

05/04/22

T

192

c-use(v): (c for computation) all variables that are used to define other variables in the code corresponding to v

p-use(v; v0): (p for predicate) all variables used in taking the (v; v0) branch out of vertex v.

http://www.inf.ed.ac.uk/teaching/courses/st/2011-12/Resource-folder/07_dataflow1.pdf

Testování toku dat (Data flow coverage)

line	kategorie			Dvojice definice → použití	proměnné	
	definice	c-use	p-use		c-use	p-use
1	a,b,c			Počátek → konec		
2	disc	a,b,c		1→2	a,b,c	
3	r.isComplex, r.r1, r.r2			1→11	a,b,c	
4				1→12	a,b	
5			disc	2→5		disc
6	r.isComplex			2→11	disc	
7				2→12	disc	
8	r.isComplex			3→10		r.isComplex
9				3→13	r.isComplex, r.r1, r.r2	
10			r.isComplex	6→10		r.isComplex
11	r.r1	a,b,disc		6→13	r.isComplex	
12	r.r2	a,b,disc		8→10		r.isComplex
13		r.isComplex, r.r1, r.r2		8→13	r.isComplex	
14				11→13	r.r1	
				12→13	r.r2	

05/04/22

T5K

193

c-use(v): (c for computation) all variables that are used to define other variables in the code corresponding to v

p-use(v; v0): (p for predicate) all variables used in taking the (v; v0) branch out of vertex v.

http://www.inf.ed.ac.uk/teaching/courses/st/2011-12/Resource-folder/07_dataflow1.pdf

Testování podmínek skoků (Branch Condition Testing)

```
if(A || (B && C)) {  
    //do something  
} else {  
    //do something else  
}
```

Případ	A	B	C
1	FALSE	FALSE	FALSE
2	TRUE	TRUE	TRUE

Každý operand podmínky se musí provést pro hodnotu true i false.

Testování kombinace podmínek skoků

```
if(A || (B && C)) {
    //do something
} else {
    //do something else
}
```

- Testuje všechny kombinace booleovských hodnot A, B, C

Případ	A	B	C
1	FALSE	FALSE	FALSE
2	TRUE	FALSE	FALSE
3	FALSE	TRUE	FALSE
4	FALSE	FALSE	TRUE
5	TRUE	TRUE	FALSE
6	FALSE	TRUE	TRUE
7	TRUE	FALSE	TRUE
8	TRUE	TRUE	TRUE

Každý operand podmínky se musí provést pro hodnotu true i false.

Testování změny výsledku podmínky

Případ	A	B	C	Výstup
A1	FALSE	FALSE	TRUE	FALSE
A2	TRUE	FALSE	TRUE	TRUE
Případ	A	B	C	Výstup
B1	FALSE	FALSE	TRUE	FALSE
B2	FALSE	TRUE	TRUE	TRUE
Případ	A	B	C	Výstup
C1	FALSE	TRUE	TRUE	TRUE
C2	FALSE	TRUE	FALSE	FALSE
Případ	A	B	C	Výstup
1 (A1,B1)	FALSE	FALSE	TRUE	FALSE
2 (A2)	TRUE	FALSE	TRUE	TRUE
3 (B2,C1)	FALSE	TRUE	TRUE	TRUE
4 (C2)	FALSE	TRUE	FALSE	FALSE

05/04/22

TSK

196

Modified Condition Decision Testing and Coverage

Modified Condition Decision Coverage (MCDC) is a pragmatic compromise which requires fewer

test cases than Branch Condition Combination Coverage. It is widely used in the development of

avionics software, as required by RTCA/DO-178B.

Modified Condition Decision Coverage requires test cases to show that each Boolean operand (A, B

and C) can independently affect the outcome of the decision. This is less than all the combinations (as

required by Branch Condition Combination Coverage).

For the example decision condition $[A \text{ or } (B \text{ and } C)]$, we first require a pair of test cases where

changing the state of A will change the outcome, but B and C remain constant, i.e. that A can

independently affect the outcome of the condition:

Testování LCSAJ (Linear Code Sequence and Jump)

```
1. int main (void) {
2.   int count = 0, totals[MAXCOLUMNS], val
   = 0;
3.   memset (totals, 0, MAXCOLUMNS *
   sizeof(int));
4.   count = 0;
5.   while ( count < ITERATIONS ) {
6.     val = abs(rand()) % MAXCOLUMNS;
7.     totals[val] += 1;
8.     if ( totals[val] > MAXCOUNT ) {
9.       totals[val] = MAXCOUNT;
10.    }
11.    count++;
12.  }
13.  return (0);
14.}
```

LCSAJ Číslo	Start	Konec	Skok na
1	1	5	13
2	1	8	11
3	1	12	5
4	5	5	13
5	5	8	11
6	5	12	5
7	11	12	5
8	13	13	-1

[http://en.wikipedia.org/wiki/
Linear_code_sequence_and_jump](http://en.wikipedia.org/wiki/Linear_code_sequence_and_jump)

4.4 – Testová data

Příprava testových dat

- Profesionální generátory dat
- Modifikovaná/degradovaná reálná data
- Nástroje pro administraci dat
- Vlastní generátory dat
- Excel, SQL skripty
- Nástroje pro automatizované testování (nástroje pro spouštění testů, nástroje pro přípravu dat, nástroje pro zátěžové testování, generátory zátěže, atd.)

05/04/22

TSK

198

- Professional data generators
 - Data generator controlled by syntax and semantics
 - Stochastic data generator
 - Data generator based on heuristic algorithms
 - Combination of previous methods
- Modified production data

The data must be degraded (omitting sensitive data) before using as test data. The advantage is that we have test data that are close to real production data. The disadvantage is that data must be modified to have all combinations needed for test cases.
- Data administration tools

E.g. File-AID/Data Solution (Compuware), RC Extract (CA), Startool/Comparex (Serena), Relational Tools for Servers (Princeton Softech), detailed knowledge of DB structure and links is needed. Tools are ready for it's sometimes difficult and laborious.
- Own data generators

Development resources are needed, suitable when combining with Excel. Sophisticated data can be generated which are tailored to the needs of test cases.
- Excel

DB tables are stored in Excel, SQL scripts generate DB structures tailored to the needs of test cases. The initial data has to be stored manually – laborious.
- Test automation tools

E.g. WinRunner, QuickTest Pro, LoadRunner, SilkPerformer, etc. Data can be generated during nights, test data can be stored to database, Excel or text files. Tools are often expensive.

Co ovlivňuje přípravu testových dat?

- Složitost a funkcionalita aplikace
- Použitý vývojový nástroj
- Opakování testování
- Množství potřebných dat

- Complexity and functionality of the application

It directly influences the range of testing, mutual linking and the amount of test data (financial systems must be tested in more details than registration systems).

- Used development tool

In case of using test automation tools, the development environment must be compatible with used test tools.

- Repetition of testing

The efficiency of using test automation tools is higher the higher repetition of the same test cases (regression testing) is (valid not only for test data preparation but also for test execution).

- The amount of data needed

Doporučení pro testová data

- Nemazejte stará testová data
- Popisujete testová data
- Kontrolujte testová data
- Očekávané výsledky

- *Don't remove old test data*

Create test data archive for future use.

- *Describe test data*

Create your own information system from metadata describing content, form and effectiveness of test data.

- *Check test data*

Test data must be error free.

- *Expected results*

Don't underestimate time needed for setting expected results for generated test data.

4.5 - Tipování vad (Error guessing)

- Není to strukturovaná/exaktní technika testování
- Myšlenka
 - Rýpejte různě to systému, podle vašeho čichu a předchozích zkušeností a snažte se najít co nejvíce vad
- Tipy a triky
 - Nula a její reprezentace
 - “Bílá místa”
 - Hledání oddělovačů (v textu)
 - Mluvte s lidmi

05/04/22

TSK

201

Error guessing is not a structured testing method since it does not rely on any particular procedure or algorithm. Nevertheless it might be useful as a complement to ordinary methods.

The idea behind error guessing is to exploit the knowledge gathered by the testers from previous testing work. Often experienced testers, support staff and sometimes end-users have a feel for where there might be problems in the system. Error guessing is thus a test activity in which knowledgeable testers are let loose in the system without no other instructions than to find faults. The main advantage with this approach is that faults are often found that have escaped the traditional testing methods, since experience and intuition are major components of error guessing. The two main disadvantages with this approach is that it requires both testing and domain knowledge to work well and although complex faults often are found, this approach does not produce any test coverage results, and thus cannot be used to increase the confidence in the product. Even if this method lacks formal procedures documentation of each test case is still required.

Zero and all representations of zero are good candidates for test cases. Common representations of zero are NULL-pointers and the empty list. In many cases “zero” is a legal input but tends to be forgotten during implementation since algorithms usually are constructed for non-zero values.

Another tip while error guessing is to exchange similar characters for one another. This is most apparent when testing input that can contain space-characters. The notion of “white-space” includes all characters that are represented on the screen by one or more spaces. Example are space, tab, back-tab, and sometimes new-line and cash-return.

For input where matching delimiters, e.g. parentheses are required try unmatched variants, especially at the end of the input.

Often the most efficient way to start an error guessing session is to interview people. Designers can give helpful hints on where they were having problems while designing and implementing. Users can explain how the system is *really* used and support people often have good ideas on where problems were residing in previous releases of the system.

5 – Management testů

- Organizační struktury pro testování
- Konfigurační management
- Test Estimation, Monitorování a kontrola
- Management incidentů
- Standardy pro testování

5.1 – Organizační struktury pro testování

- Vývojáři testují vlastní kód
- Zodpovědnost na vývojovém týmu (kamarádké testování)
- Tester(testeři) ve vývojovém týmu
- Specializovaný tým testerů
- Interní konzultant pro testování, poskytuje rady/doporučení k projektu
- Separátní organizace zaměřená na testování

05/04/22

TSK

203

- Developer's test their own code
 - + The developer knows the code best. The developer can fix found faults.
 - Subjective assessment. If a requirement is misunderstood by the developer, the misunderstanding will remain after test. Reluctance to change from a constructive to a destructive attitude, it is hard to destroy own work.
- Development team responsibility (buddy testing)
 - + At least some independence. Testing is done on friendly terms within the team ("buddy").
 - Focus is on development, not on test. Lack of testing skills. Only a development view of the tested system.
- Tester(s) in the development team
 - + Independent view of the software. Resources dedicated for testing, part of the team, the same goal.
 - Lack of respect, lonely, thankless task. One tester – one view.
- A dedicated team of testers
 - + Dedicated for testing! Specialized in testing, adds objectivity and consistency to testing.
 - Lack of product implementation details. May be confrontational.
- Internal test consultants providing advice to projects
 - + Highly expertised in testing. Can use experiences from earlier projects for better planning and control. Can assure consistency of testing across several projects.
 - No authority, only provide advice. Someone else has to do the testing.
- A separate test organization
 - + Highly expertised in testing. Independent of company internal politics.
 - Lack of company and product knowledge. Expertise gained outside the company.

Organizační struktury pro testování - Nezávislost

- Kdo provádí testování?
 - Testování komponent – vývojáři
 - Testování systému – nezávislý testovací tým
 - Akceptační testování – uživatelé/zákazník
- Nezávislost je mnohem důležitější při návrhu testů než při provádění testů
- Použití strukturovaných testovacích technik zvyšuje nezávislost
- Dobrá testovací strategie využívá kombinaci vývojářů a nezávislých testerů pro testování

A development organization testing its own code is worse than a surgeon operating on himself.

At least the surgeon has vital interest in the result.

The development organization gets paid for delivering a specific product on time. The quality of the product is rarely specified. Testing jeopardizes the possibility to deliver on time.

- *Independence is more important during test design than during test execution*

Tests can be executed by almost anyone as long as they are well specified. It is hard for a developer to be objective while designing tests of software produced by himself/herself. I.e. if the amount of time available for independent testes is small, spend that time on test design, preferably of high level tests (e.g. system test level).

- *The use of structured test techniques increases the independence*

A certain amount of independence can be achieved by using structured testing techniques since it is the technique used, not the tester, that decides what to test.

- *A good test strategy should mix the use of developers and independent testers*

Low level tests (component tests, component integration tests) are preferably done by developers in combination with the use of structured testing techniques. The developers know the code best and with the use of testing techniques they will probably find most of the faults that should be found during component testing and they can provide fast debugging. Higher level tests should be done by independent testers to increase the objectivity and reduce the risk for misunderstood requirements, etc.

Schopnosti specialisty potřebné při testování

- Manažer testů (management testů, zprávy, analýza rizik)
- Analytik testů (návrh testových případů)
- Experti na automatizaci testování (programování testovacích skriptů)
- Expert na výkonostní testování (vytvoření modelu testů)
- Administrátor nebo návrhář databáze (příprava testovacích dat)
- Expert na uživatelské rozhraní (test použitelnosti)
- Manažer testovacího prostředí
- Expert na testovací metodologie
- Expert na testovací nástroje
- Doménový expert

5.2 – Konfigurační management

Co zahrnuje konfigurační management?

- Identifikaci konfigurovaných položek
- Kontrola konfigurací:
 - Hardware
 - Software
 - Dokumenty
 - Metody/Metodiky
 - Nástroje
- Soupis statutu konfigurace (Configuration Status Accounting)
- Konfigurační audit

05/04/22

TSK

206

Configuration management is a discipline applying technical and administrative *direction* and *surveillance* to: *identify* and document the functional and physical characteristics of a configuration item, *control* changes to those characteristics, *record* and *report* change processing and implementation status, and *verify* compliance with specified requirements.

Symptoms of poor configuration management:

- Confusion

Not knowing which the actual/latest version of a configuration item is. Unable to match source and object code. Unable to identify which version of a compiler that generated the object code. Unable to identify the source code changes made in a particular version of the software. Not knowing which version of the system that is delivered to a specific customer.

- Recurrence of bugs

Defects that were fixed suddenly reappear.

- Conflicting changes

Simultaneous changes made to the same source module by multiple developers and some changes lost. Simultaneous changes that is not consistent done to different parts of the system.

- Unauthorized changes

Undocumented features suddenly appear. Tested features suddenly disappear.

Identifikace konfigurací

- Identifikace konfigurovaných položek (CI)
- Značení (Labelling) konfigurovaných položek
 - Značení musí být jedinečné
 - Značení se obvykle skládá ze dvou částí:
 - Jméno, obsahuje titulek a číslo
 - Verze
- Jmenné a verzovací konvence
- Identifikace základních/kontrolních linií(baselines)

Identify all parts that need to be controlled. What is the smallest part to be configuration managed?

The configuration identification shall reflect the product structure.

All configuration items must be labelled. Use a standard for naming and versioning. If your company does not have a standard, define one!

What is a baseline?

- A snapshot of a configuration at a certain point in time.
- A way to measure where in the development cycle a system really is.

Why baselines?

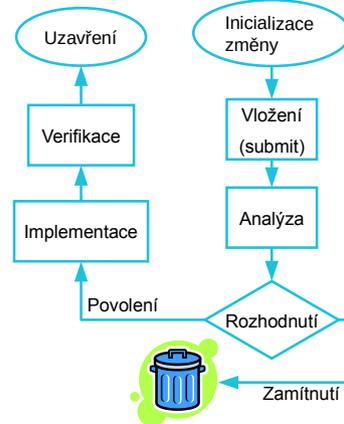
- A stable point from which new projects (or releases) can be developed.
- To roll back to if changes have caused big problems.
- Possibility to recreate the configuration of the system.
- A base for testing.
- A base for supporting.
- A starting point for more formalized control.

Examples of baselines: functional baseline, design baseline, development baseline, product baseline.

Konfigurační management - Kontrola konfigurací

- Management verzí
- Vytváření základních linií (baselines)
- Kontrola změn

Procedura kontrolované změny



05/04/22

TSK

208

Configuration control is formal and structured handling of items, their configuration and changes. According to the foundation syllabus configuration control is the maintenance of the CI's in a library and maintenance of records on how CI's change over time.

- Version management

Managing work in parallel by using file locking (check in/check out), branching and merging in a structured way.

- Establishment of baselines

- Contents
- Quality
- Deviations
- Decision

- Change control

Who is authorized to make decisions regarding changes? Often a Configuration Control Board (CCB) is used for that purpose. A change control procedure must be defined to be able to handle changes in a controlled way. Incident reports can be handled in the same way.

Soupis statutu konfigurace (Configuration Status Accounting)

- Zaznamenání a vytvoření zprávy popisující konfigurované položky a jejich status

Informační strategie

- Co?
 - Jaké změny byly udělány v posledním releasu?
 - Jaké opravy/změny jsou plánovány pro další maintenance release?
 - Jaké incidenty jsou stále otevřené?
 - Jaký je aktuální status konfigurační položky xyz?
 - Jaký je stav incidentu 123 a kdo je zainteresovaný uživatel?
 - atd.
- Pro koho?
 - Projektový manager, design/vývojový/test/release manager, etc.
- Kdy?
 - CCB a projektové schůzky.
 - Milestones, baselines, releases, atd.
- Jak?
 - CM tool, incident tracking tool
 - Logs, configuration records, reports
 - www, bulletin board, release notes, atd.

05/04/22

TSK

209

- What?

Status accounting shall make it possible to answer questions like:

- What changes are made in the latest release?
- Which corrections/changes are planned for the next maintenance release?
- Which incident reports are still open?
- What is the current status of CI xyz?
- What is the status of incident report 123 and who is currently appointed user?
- Etc.
- To whom?
 - Project manager, design, development, test, release manager, etc.
- When?
 - CCB and project meetings.
 - Milestones, baselines, releases, etc.
- How?
 - CM tool, incident tracking tool
 - Logs, configuration records, reports
 - www, bulletin board, release notes, etc.

Konfigurační audit

Auditování konfigurace produktu se většinou provádí před uvolněním produktu.

- Provádí ji skupina reprezentující:
 - Projektový management, konfigurační management, testování, systémové inženýry, návrh, produktový management.
- Může zahrnovat také oblasti:
 - zákazníky, produkci, kvalitu, instalaci/údržbu, bezpečnost, logistika.
- Vyspělost
 - Jsou konfigurační položky dost vyspělé?
 - Jsou otestovány?
 - Jaký byl výsledek testování?
 - Existují neuzavřené incidenty?
- Kompletnost
 - Existují všechny naplánované konfigurační položky?
 - Jsou uloženy na správném místě?
 - Mají správné číslo verze?
- Vyhovění požadavkům
 - Jsou všechny požadavky splněny?
- Integrita
 - Jsou zdroje bezpečné?
 - Je možno znovu sestavit konkrétní verzi produktu a jeho vývojové a testovací prostředí?
- Výstížnost
 - Jsou požadavky validní, tedy jedná se o produkt který zákazník chce nebo potřebuje?

Configuration auditing is a pre-release action, often performed by a group of people representing project management, configuration management, test, system engineering, product management, design. The following areas could also be represented, depending on the situation: customer, production, quality, installation/maintainability, safety, logistics.

- Maturity

Are the configuration items mature enough? Are they tested? What was the result of the tests? Are there any incident reports that are still open?

- Completeness

Does all planned configuration items exist? Are they stored in the right place? Do they have the right version number?

- Compliance with requirements

Are all requirements fulfilled?

- Integrity

Is the source secure? Is it possible to recreate a specific version of the product and its development environment (don't forget the test environment!)?

- Accuracy

Are the requirements valid, i.e. is the product what the customer wanted?

http://www.chambers.com.au/glossary/configuration_audit.php

Konfigurační management a testování

Co má být pod správou konfiguračního managementu?

- Všechny testové dokumenty a testware
- Dokumenty ze kterých testová dokumentace vychází
- Testovací prostředí
- Produkt k testování

Proč?

- Zpětná sledovatelnost (Traceability)

A strong implementation and understanding of CM enables the possibility to establish traceability between:

- Test documentation and product versions
- Executed test cases and version of test environment used
- Incident reports and software/hardware configurations
- Test cases and requirements
- Etc.
- CM can be very complicated in environments where mixed hardware and software platforms are being used, but sophisticated cross-platform CM tools are increasingly available.

5.3 – Odhad testů, monitorování a kontrola

Odhad testů

- Proč se to děje znovu a znovu?
- Jsme opravdu tak neschopní v odhadu testů?
- Co můžeme udělat proto abychom se vyhnuli podobným situacím?



05/04/22

TSK

212

Test estimation is to calculate the effort required to perform the activities specified in the high level test plan in advance. The reason for doing that is the same as for all planning, i.e. to be able to control the progress and to be better prepared to reschedule when something unexpected occurs. The test manager may have to report on deviations from the project/test plans such as running out of time before completion criteria achieved. Test estimation is estimation, done in advance, of the effort required to perform activities specified in the high-level test plan. Rework must also be planned for. Often testing must be interrupted before the planned completion criteria are fulfilled due to lack of time.

Odhad - Pravidlo palce

- Počet řádků zdrojového kódu
 - 1 TC na 30-50 řádků kódu (SW s vysokým rizikem)
 - 1 TC na 100-150 řádků kódu (SW s středním rizikem)
 - 1 TC na 500-1000 řádků kódu (SW s nízkým rizikem)
- Okna (webové stránky) aplikace
 - 3-5 TC na okno (SW s nízkým rizikem)
 - 10-15 TC's na okno (SW s středním rizikem)
 - 20-30 TC's na okno (SW s vysokým rizikem)
- Stupeň modifikace



Stupeň modifikace	Stupeň testování
5%	5%
10%	25%
15%	50%
20%	100%

05/04/22

TSK

213

- Windows Vista přibližně 50 miliónů řádku kódu
- Pečlivě vyvíjený software obsahuje 5 vad na 1000 řádků
- Microsoft má poměr developer : tester = 1:1
- Lines of the source code
 - 1 TC per 30-50 lines of the source code (high-risk SW products)
 - 1 TC per 100-150 lines of the source code (medium-risk SW products)
 - 1 TC per 500-1000 lines of the source code (low-risk SW products)
- Windows (Web-pages) of the application
 - 3-5 TC's per window (high-risk SW products)
 - 10-15 TC's per window (medium-risk SW products)
 - 20-30 TC's per window (low-risk SW products)
- Degree of modifications

Degree of modifications	Degree of Testing
5%	5%
10%	25%
15%	50%
20%	100%

Aktivity odhadu testů

- Identifikovat testovací aktivity
- Odhadnout čas pro každou aktivitu
- Identifikovat potřebné zdroje a schopnosti
- V jakém pořadí by měli být aktivity prováděny?
- Pro každou aktivitu je třeba určit
 - Počáteční a koncové datum
 - Zdroje pro danou aktivitu

This is no different from estimating any job.

If possible, base your estimation on history from earlier projects:

- The expected number of faults to be found.
- Time to write and report incidents.
- Time from incident reporting until the fault is fixed
- Etc.

Estimation is dependent of:

- The project size.
- The number of features/characteristics to be tested, i.e. the scope of testing.
- The complexity of the product/system.
- Quality objectives.
- The expected quality of the system when delivered to test (low quality = more testing).
- The size and complexity of the test environment.
- The usage of tools.

Co dělá odhad složitým?

- Testování není nezávislé
 - Kvalita softwaru dodána k testování?
 - Kvalita požadavků, které se mají otestovat?
- Vady budou nalezeny!
 - Jak mnoho?
 - Závažnost?
 - Čas potřebný na opravu?
- Testovací prostředí
- Jak mnoho iterací?

05/04/22

TSK

215

Testing is not independent

- For example, if the quality of the software delivered to test is poor, this will strongly affect the test plan since more testing will be needed.
- Late deliveries.
- Missing functionality in delivered software.
- Important to be flexible and adjust the testing to the current situation.

Faults will be found!

- Unplanned deliveries due to the need to fix found defects.
- Re-testing and regression testing is needed.

Test environment

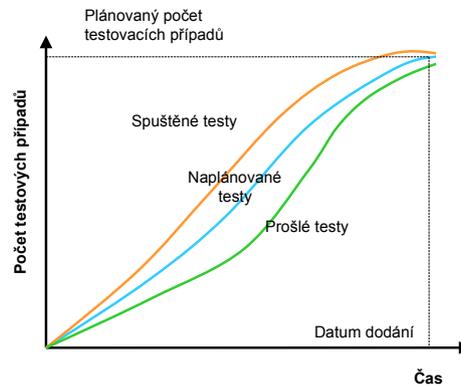
- Limited access to the test environment.
- Test environment unstable.
- Parts of the test environment that are out of our control could be delivered late, malfunctioning, etc.

How many iterations?

- Iterations are based on the test plan which can change during software development
- Only 1-2 iterations can be planned in details
- Plan iterations only for fixing found defects in the later stage

Monitorování průběhu provádění testů Bez odchylek od plánu

- Vysoká kvalita?
 - Nalezeno pár vad
- Jsou testy dostatečné?



05/04/22

TSK

216

Is the software quality necessarily high because we do not find many faults?

Scenario 1, low quality of tests:

The quality of the tests can be poor due to lack of testing skills or because no testing techniques are used.

Actions to prevent this scenario from happening:

- Use structured testing techniques.
- Use a test strategy based on risks.
- Use a test coverage criteria.
- Review/Inspect test specifications before writing test instructions or test scripts to make sure that all necessary tests are included and that the right things are tested.

Possible actions if it happens:

- Stop test execution until the quality of the tests has been improved by adding tests using testing techniques and experienced testers.

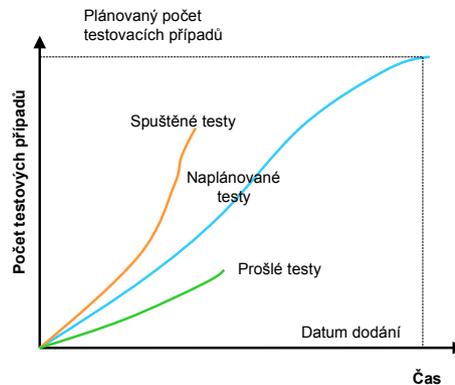
Scenario 2, high quality of tests:

A well defined strategy for using testing techniques in the different test phases has been used. Inspections showed that the quality of the tests were good. Congratulations to a successful software development project!

Monitorování průběhu provádění testů

Možné problémy ke sledování

- Nízká kvalita
 - Softwaru
 - Testů
- Kritéria pro vstup do testování
- Nejprve prováděny jednoduché testy



05/04/22

TSK

217

Why is it hard to get the test cases passed? The big difference between the number of test cases run and the number of test cases passed indicates low quality of either software or tests (or both), probably in combination with a poor test entry criteria.

Low quality of software or tests

Stop testing until the faults are fixed/the quality of tests are improved. Reallocate resources for debugging/fault fixing and/or improvements of tests.

Test entry criteria

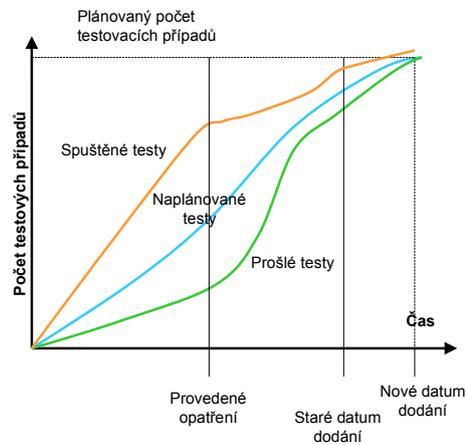
If the test entry criteria is poor, change the criteria. Don't accept software that does not fulfill the criteria.

Easy tests run first

The difference between the number of planned test cases and the number of run test cases may be caused by executing the easy (the less time consuming) tests first. All the complicated test cases (the ones that have the highest probability of revealing serious problems) are still waiting to be executed.

Monitorování průběhu provádění testů

Změny provedené pro zlepšení průběhu



05/04/22

TSK

218

Actions are taken to improve the test progress. Despite that it is obvious that the testing will not be ready before the delivery date. This will affect the time plan for the whole project. Probably the delivery date will have to be changed.

What if all tests are not executed before the delivery date?

- What are the risks of not running the tests?
- What are the severity of outstanding faults?
- Is it possible to continue testing after release?

Řízení testů

Co dělat v případě že nastanou události, které ovlivní plán testů:

- Přeplánování zdrojů
- Změny v plánu spouštění testů
- Změny v testovacím prostředí
- Změny ve vstupních/výstupních kritériích
- Změna počtu testovacích iterací
- Změna v sadě testovacích případů (test suit)
- Změna data dodání/uvedení na trh

05/04/22

TSK

219

Things will happen that affects the test plan! We all know that testing is not independent, it is closely connected to development and in addition, it is the activity that comes last in the software development process (at least the test execution). The better prepared we are the easier it is to control the test activities and any unforeseen events.

Re-allocation of resources

Re-resources can be moved from one activity to another, more resources may be needed, etc.

Changes to the test schedule

For example, if a test object is delivered to test later than planned, that might affect the order in which the tests can be executed, and therefore the schedule must be changed. The estimated time for each test activity can also be changed if the estimation does not agree with the actual time needed.

Changes to the entry/exit criteria

If the entry criteria to test/re-test is too weak, i.e. the quality of the software delivered to test is poor, it can be changed. The exit criteria from test can also be changed if it shows not to be appropriate.

This will affect the test schedule.

Changes to the number of test iterations

If more defects are found than expected, this will enforce more re-testing and regression testing and also the number of test iterations planned. This number is also changed when a functionality planned is moved to next iterations.

Changes to the test suite

Test suite (set of test cases) is prepared during implementation phase. When the software is delivered to test, it can be found out that more test cases have to be developed.

Changes of the release date

Even if we take actions to eliminate the difference between the test plan and the reality, we cannot fully avoid changes of the release date. This must be communicated to customers by professionals with some explanations.

5.4 – Management incidentů (Incident Management)

Co to je incident?

- Význačná, neplánovaná událost, která se stane v průběhu testování, nebo jiná událost, která vyžaduje další průzkum nebo nápravu.
 - Rozdíl mezi skutečným a očekávaným výsledkem testu
 - Možné důvody:
 - Selhání softwaru
 - Očekávaný výsledek je nesprávný
 - Test nebyl proveden správně
 - Může vzniknout ve vztahu ke kódu a/nebo k dokumentaci

Other common names for incident reports are trouble reports, error reports, problem reports, etc.

Incident shall be logged when someone other than the author of the product under test performs the testing. That is not always true, sometimes also the author logs incidents. It depends on the test process, the development process, the size of the project, etc.

It is easy to miss or forget important information if incidents are not reported immediately when they occur. Valuable time for debugging/fault fixing will also be lost if the incidents are reported later, therefore incidents shall be reported as soon as possible.

Detailed description of the incident is important for developers to reproduce the fault and to speed up the fixing of the fault

Informace obsažené v incidentu

- Jednoznačné ID zprávy o incidentu
- ID testovacího případu a číslo revize/verze
- ID testovaného softwaru a číslo revize/verze
- ID testovacího prostředí a číslo verze/revize
- Jméno testera
- Očekávaný a skutečný výsledek
- Závažnost(Severity), rozsah(scope) a prioritá
- Jakékoliv další relevantní potřebné k reprodukci nebo opravě potenciální vady

05/04/22

TSK

221

Describe the incident and the environment as carefully as possible. Don't forget to include the version of test case, software under test and the test environment (e.g. for Web applications specify Web browser and its version).

To make sure that the provided information is sufficient, ask someone else to check the incident report before you submit it!

Severity – indicates the importance to users.

- High – if the system crashes or is unusable. It has significant impact on the functionality. Prevents usage of the application.
- Medium – if a workaround is available. It does not have a great effect on the functionality but is quite obvious as well as disturbing to the user. Usage of the application is limited.
- Low – cosmetic problems (≠ usability!). Presents a small problem, but does not affect the functionality. In many instances, this represents a change to the user interface or in documentation .

Priority – indicates the urgency to fix the fault.

- High – stops further testing or testing is heavily hindered. Requires immediate actions.
- Medium – stops some tests, other tests can proceed. Solution is implemented in the next release.
- Low – possible to proceed. Solution is planned for next or any other release or not implemented at all.

Note that a fault that stops further testing and therefore has high priority can have low severity.

Sledování incidentů

- Kdykoliv musí být možné zjistit současný status nahlášeného incidentu např.:
 - Nový (New)
 - Otevřený (Open), Znovu otevřený (Re-open)
 - Čekající (Wait), Zamítnutý (Reject), Opravený (Fixed)
 - Zahrnut v sestavení (Included to build)
 - Ověřený (Verified)
 - Uzavřený (Closed)

Incidents should be tracked from discovery through the various stages to their final resolution.

They have different states, e.g.:

New - the defect has been found and reported to some defect management tool by a tester.

Open – the defect manager assigns the defect to a developer for analysing the cause of the defect.

Wait – the defect cannot be fixed, some information is missing

Reject – the defect was not accepted, no corrections are needed (misunderstanding, defect in test case or test data, problem was not reproduced, defect caused by hardware, back end or network problems, etc.).

Fixed – the defect was fixed, it is ready for testing.

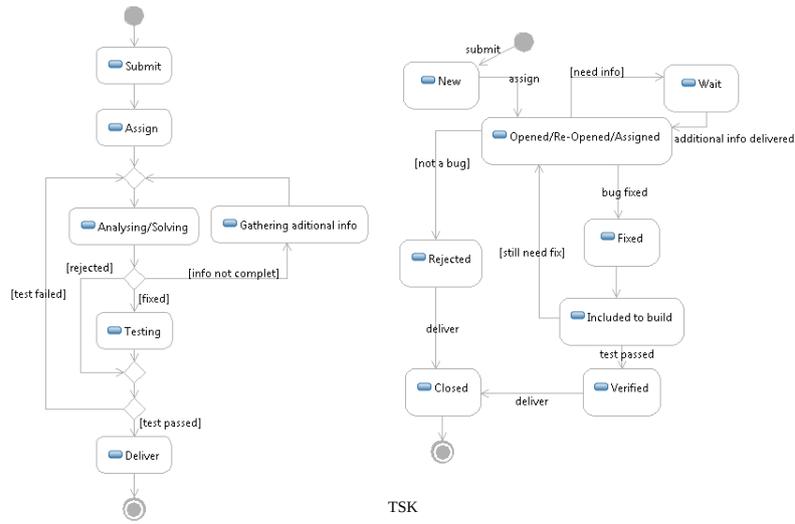
Included to build – the build/configuration manager includes the files with the corrections (the fix) to the next internal build.

Verified – the fix was successfully tested (accepted by the tester).

Re-open – the fix was not done properly or produced side effects (rejected by the tester).

Closed – the fix was delivered to an official release.

Management incidentů



05/04.

TSK

223

Incidents should be tracked from discovery through the various stages to their final resolution. They have different states, e.g.:

New - the defect has been found and reported to some defect management tool by a tester.

Open - the defect manager assigns the defect to a developer for analysing the cause of the defect.

Wait - the defect cannot be fixed, some information is missing

Reject - the defect was not accepted, no corrections are needed (misunderstanding, defect in test case or test data, problem was not reproduced, defect caused by hardware, back end or network problems, etc.).

Fixed - the defect was fixed, it is ready for testing.

Included to build - the build/configuration manager includes the files with the corrections (the fix) to the next internal build.

Verified - the fix was successfully tested (accepted by the tester).

Re-open - the fix was not done properly or produced side effects (rejected by the tester).

Closed - the fix was delivered to an official release.

5.5 – Standardy pro testování

Typy standardů pro testování

- Standard Zajištění kvality (QA)
 - Určují, že testování by mělo být provedeno
 - ISO 9000
- Standardy pro jednotlivá průmyslová odvětví
 - Specifikují úroveň testování
 - Railway signalling standard
 - DO 178b, Software Considerations in Airborne Systems and Equipment Certification

QA standards

- ISO 9000

Industry-specific standards

- Railway signalling standard (many different standards exist, often national)
- DO 178b, Software Considerations in Airborne Systems and Equipment Certification

Testing standards

- BS 7925-1, Vocabulary of Terms in Software Testing
- BS 7925-2, Software Component Testing. Give guidelines for the most common test documents and test activities. Includes six black box and seven white box test methods.
- IEEE Std. 829/1998, Standard for Software Test Documentation
- IEEE Std. 1008/1987, Standard for Software Unit Testing
- IEEE Std. 1012/1998, Standard for Software Verification and Validation

Ideally testing standards should be referenced from the other two.

Typy standardů pro testování

- Standardy pro testování
 - Specifikují jak provádět testování
 - BS 7925-1, Slovníček pojmů v softwarovém testování
 - BS 7925-2, Testování softwarových komponent. Určuje směrnice pro nejčastější dokumenty v testování a testovací aktivity. Obsahuje 6 black-box a 7 white-box testovacích metod.
 - IEEE Std. 829/1998, Standard pro dokumentaci k testování
 - IEEE Std. 1008/1987, Standard pro testování komponent (Unit testing)
 - IEEE Std. 1012/2016, IEEE Standard for System, Software, and Hardware Verification and Validation
- ISO/IEC/IEEE 29119 Software Testing (1-5) nahrazuje:
 - IEEE 829 Test Documentation
 - IEEE 1008 Unit Testing
 - BS 7925-1 Vocabulary of Terms in Software Testing
 - BS 7925-2 Software Component Testing Standard

http://www.testingstandards.co.uk/bs_7925-1.htm

http://www.testingstandards.co.uk/bs_7925-2.htm

6 – Testovací nástroje

- Typy testovacích nástrojů (CAST tool – Computer Aided Software Testing)
- Výběr nástrojů a implementace/nasazení

Typy testovacích nástrojů

- Nástroje na správu požadavků (Requirements)
- Nástroje na statickou analýzu (Static Analysis)
- Nástroje pro návrh testů (Test Design)
- Nástroje pro přípravu testových dat (Test Data Preparation)
- Nástroje pro spouštění testů (Test-running)
- Nástroje pro Harness testování (Test Harness & Drivers)
- Nástroje pro výkonnostní testování (Performance Test)
- Nástroje pro dynamickou analýzu (Dynamic Analysis)
- Ladící nástroje (Debugging)
- Porovnávací nástroje (Comparison)
- Nástroje pro správu testů (Test Management)
- Nástroje pro zjišťování pokrytí (Coverage)

CAST = Computer Aided Software Testing

Various types of test tool functionality are available (see the list above) depending on what functionality we need. In industrial reality, there is seldom one-to-one mapping between a “tool” and a “functionality”. For example, the comparison functionality may sometimes be implemented in a separate “comparison tool”, but more often this comparison will be a part of a test-running tool.

Large, complex systems often have more input/output interfaces and attributes to measure, than is supported by any single tool. A common situation is that a number of different test-running tools are used. Another tool, a test harness controls them, or one of the tools is given the role of test harness and controls the other tools.

Nástroje na správu požadavků (Requirements)

- Nástroje pro testování požadavků a specifikace požadavků:
 - Kompletnost
 - Konzistence pojmů
 - Model požadavků: graf, animace
- Sledovatelnost (Traceability):
 - Od požadavků k testovacím případům
 - Od testovacích případů k požadavkům
 - Z produktu k požadavkům a testovacím případům
- Propojení nástroje pro požadavky s nástrojem pro management testů



Requirements engineering is not part of testing, so the tools used for gathering, analysis, and structuring of requirements do not belong to this chapter. However, these tools often provide automated support for traceability, which is important for testing. Examples of requirement management tools: Analyst Pro/ Goda Software, Doors/Telelogic, Caliber/Borland, Gatherspace/Gatherspace.com, RequisitePro/IBM Rational, SpeedEV RM/SpeedEV, SteelTrace/SteelTrace and TeamTrace/WA Systems.

- Tools for testing requirements and requirements specifications

Requirement specifications themselves have to be tested for completeness, consistency, etc. This is most often done using reviews and inspections. For formal modelling methods, automatic verification is available. Automatic validation of the requirements is harder.

- Traceability

Mapping between requirements and test cases is needed to verify that all requirements are correctly implemented, to identify all affected test cases when a requirement is changed, to know what requirements are affected in case that a test case fails. The set of all requirements can be divided into subsets for product versions, customer deliveries or increments. For a given product version or increment, applicable sets of test cases can be then easily generated in accordance with the requirements. The requirement management tools support this too.

- Connecting requirements tools to test management tools

For test management purposes, the connection between test management tools and requirement management tools should be automated. Test reports shall seamlessly translate into lists of verified/not verified requirements, and lists of requirements shall seamlessly generate suits of test cases. A manual intervention makes this process less reliable. Test management tools nowadays include requirement management functionality (e.g. Quality Center/Mercury).

Nástroje pro statickou analýzu

- Testování bez spuštění kódu
- Kompilátory, pokročilá analýza syntaxe
- Chyby v toku dat
- Měření složitosti a další atributy
- Grafická reprezentace řídicího toku
- Jak používat statickou analýzu?

- <https://sonar.cs.vsb.cz>
- Sonar Lint for Eclipse
- <https://www.sonarqube.org>
- https://en.wikipedia.org/wiki/MISRA_C



05/04/22

TSK

229

Static analysis is testing without *running*, but by *examining* the code. Typically, it is performed on the source code. It can help to find faults earlier (that dynamic analysis can), or more easily (a fault in the source code may be easier to localize using static analysis than it can be found from its consequences, i.e. after dynamic testing), or warn for possible faults and bad design (data analysis). Examples of static analysis tools: Cantata/IPL, CMT++/Testwell, LDRA Testbed/LDRA, Logiscope/Telelogic and Prevent/Coverity.

- Compilers, more advanced syntax analysis

Compilers perform a lot of static analysis, especially when warnings are not disabled. Further, for many high-level languages, more advanced static analysis can be performed. These tools discover data flow faults, identify unreachable code and parameter type mismatches, warn for possible array bound violations, ranges without stop value, etc.

- Data flow faults

A number of data flow faults exist. Some of them (e.g. undefined variable) may prevent the creation of executable code, some (e.g. not using a declared variable at all) may be harmless, some (e.g. using a variable before it has been assigned a value) may be disastrous.

- Measuring complexity

McCabe's cyclomatic complexity index, Lines Of Code, etc. are metrics that can be calculated using static analysis tools. Some tools can calculate other metrics, like the amount of changed code, nested levels, call trees, the number of times a particular routine is called by other routines, etc. These measures do not directly discover faults, but may point at badly designed areas and help when prioritizing test cases.

- Graphic representation of control flow

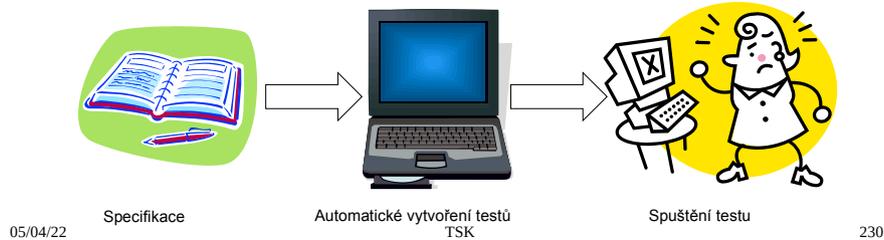
Static analysis tools (even some compilers) are able to produce a graphic representation of control flow.

- How to use static analysis

The results of static analysis are less reliable for object oriented system, distributed systems and real-time systems. In such systems a lot of information is not available until the program is executed, which makes static analysis less efficient. Static analysis may discover faults earlier and more cheaply than dynamic testing, but it does discover all faults that dynamic testing does.

Nástroje pro návrh testů

- Nejvíce „obtížná“ úroveň automatizace v testování
- Typy automatického generování testů:
 - Testovací skripty z formální specifikace testů
 - Testovací případy z požadavků nebo návrhu specifikace
 - Testovací případy z analýzy zdrojového kódu



Automated test case design and automated test case generation are potentially very powerful techniques of quality assurance. They are however seldom practiced, as using automatic test case design requires formal, high-integrity requirements and design specifications. On the other hand, this subject is extensively covered by academic research. Examples of automated test case design tools: Automated Test Designer/AtYourSideConsulting, GUITAR/University of Maryland, TAU TTCN Suite/Telelogic.

- Test scripts from formal test specifications

Automatic test script generation. Design and description of test cases must have been done manually, then tool support is used for the generation of actual test script code. “Capture” activity of capture-playback tools may be treated like a special case of automatic test script generation.

- Test cases from requirements or from design specification

There are formal languages for the description of requirements and system design, from which test cases can be generated automatically. The specification is either held in a CASE tool repository or in the tool itself.

- Test cases from source code analysis

This method is commonly used for the generation of stubs (replacement code for missing subroutines) or drivers (code that invokes tested routines). Note that expected outcomes must not be generated from code, otherwise a “self-fulfilling prophecy” will be created.

Nástroje pro přípravu testových dat

- Používají se jako „pre-execution“ nástroje
- Příprava vstupních dat a/nebo očekávaných výsledků
- Nástroje na produkci dat k naplnění databází, registrů, atd.
- Nástroje, které natáhnou data z existujících testových databází
- Speciální případy:
 - “Nahrávací” část C/R-nástrojů
 - Špatná architektura testware: “zadržovaná” testová data
 - Data-driven automatizované provádění testů



05/04/22

TSK

231

Tools for test data preparation extract or create data for use in tests. The test data is generated mainly from requirement and design specifications.

- Used as pre-execution tools

The test data is prepared before the test execution starts.

- Preparing input data and/or expected outcomes

The test data that is prepared before the test execution are the inputs and the expected outcomes: the actual outcomes are produced by the tested system after the input has been fed to the application.

- Tools that produce data to populate databases, registers, etc.

They can deal with a wide range of different file and database formats, that's why conversions are often needed.

- Tools that fetch data from existing test database

In many situations, test data (both input data and expected outcomes) is available, but its format must be changed. The translation from one to another data format may be done by conversion tools, often written specifically for this purpose.

- “Capture” part of C/R-tools (“Capture/Playback” tools)

Input data is captured (mouse movements and clicks, keyboard activity, etc.), and expected outcome data is logged and archived (typically, hard-coded into the test script). Expected outcomes are typically GUI-outputs, like GUI-objects and their states, text, graphical objects and their attributes.

- Bad testware architecture: “embedded” test data

A common error is placing test data (prepared manually or generated automatically) “hard coded” in test scripts and programs, which causes poor maintainability. This is one of the common errors when using capture-replay tools.

- Data-driven automatic test execution

Test data, if not hard coded in test program code, can be fetched by the test execution program from a file and applied.

Nástroje pro spouštění testů

- Nástroje, které naplní vstupy, zaznamenají výstupy a porovnají výsledky
- Příklad nástrojů pro spouštění testů:
 - “Přehrávací” část C/R-nástrojů
 - Pro konzolové/textové aplikace
 - Pro GUI (GUI objekty, Bitmapy)
- Externí nebo vestavěné porovnání (vyhodnocení testů)
- Testová data zadržována ve skriptech nebo uložena v souborech
- Obvykle používány k automatizované regresi
- Testové procedury v programovatelném skriptovacím jazyce



05/04/22

TSK

232

Test-running tools (also called *test executors*, “Capture/Playback”: C/R-tools), are the tools which apply the inputs, capture real outcomes and compare them with expected outcomes. An on-line input generator is thus typical part of a test-running tool, sometimes a compare tool is invoked, and possibly, the test running-tool has some test harness functionality. Examples of test-running tools: AutoTester One/AutoTester, CAPBAK/Software Research, QACenter/Compuware, QuickTest/Mercury, Robot/IBM Rational, SilkTest/Seque, and WinRunner/Mercury.

The procedures are typically written in a programmable script language (like C, VisualBasic, Perl), whereas test cases, input data and expected results may be held in separate test repositories.

- “Playback” part of C/R-tools

It runs previously generated test scripts, applying previously captured inputs and comparing test outcomes (typically, GUI outputs) with expected outcomes.

- For character-based applications

“Capture-Playback” facilities for dumb-terminal applications. The tools simulate terminal keystrokes and compares screen responses with those previously captured. Old-fashioned, simple but powerful. Receives actual outcomes by sequentially recording output stream from the tested application.

- For GUI (GUI objects, Bitmaps)

Used for WIMP (Windows Icon Mouse Pointer) interfaces. Applies inputs as mouse movements and keystrokes, and compares the GUI objects (windows, fields, buttons and other controls), their state and timing with expected outputs. Even bitmaps can be captured and compared to actual bitmaps.

- External or built-in comparison (test result evaluation) facility

The running of tests requires the ability to generate inputs, to capture outcomes and to compare actual with expected outcomes. These abilities are common functions of one and the same test tool, but other solutions are also possible.

- Test data hard-coded or in files

What inputs to send and what expected outcomes to compare actual ones with, is defined in data either stored in script files (hard-coded) or in separate data files.

- Commonly used for automated regression

For regression testing, test-running tools are the best candidates as regression test cases are repeated very often. The ROI (return-on-investment) is the most effective.

- Test procedures in programmable script languages

Such procedures or scripts are like other software programs: they can be edited and changed, debugged, executed, archived and re-used. Some tools use graphical languages for the test script programming.

Test Harnesses & Drivers

- Testové ovladače: nástroje pro spouštění testů bez generátoru vstupů
- Simulátory a emulátory jsou často používány jako ovladače
- Harness: běh mnoha skriptů/nástrojů bez zásahu
 - Vyvolání skriptů a naplnění testových dat
 - Akce v případě selhání
 - Management testů



05/04/22

TSK

233

- Test driver: a test-running tool without input generator

Typically test-running tools stimulate test object through an input interface. The test driver invokes routines and simulates them by internal parameter passing interfaces. Often, drivers are custom-written tools.

- Simulators and emulators are often used as drivers

When the target execution platform is not available, testing is often performed in a simulated environment. Simulators and emulators may be able to control execution directly by loading and execution directly by loading and executing the program, or may act as input generators. Emulators simulate HW.

- Harness: unattended running of many scripts (tools)

An exact borderline between test harness and test-running tool “from below”, and between test harness and test management tool “from above”, is not defined. Typically, a test harness is like a “super test-running tool”, controlling perhaps a few test-running tools. Test harness can be used to run groups of existing automated test scripts. Its typical activities are:

- Fetching of scripts and test data: matching right versions
- Actions for failures: reset, continue, invoke another script if previous script failed?
- Test management: reporting, incident reports

Nástroje pro výkonnostní testování

- Nástroje pro generování zátěže a pro sledování a měření
- Generování zátěže:
 - Generátor vstupů nebo stimulování ovladačem testovaného objektu
 - Úroveň stimulace
 - Zátěž prostředí
- Logování pro ladění a vyhodnocení výsledků
- On-line a off-line analýza výkonu
- Grafy a vztah zátěž-čas odezvy
- Používají se především pro testování nefunkčních požadavků... ale také pro dlouhodobé tetování (“background testing”)



05/04/22

TSK

234

Load generation tools are used for the execution of load, performance and/or stress/robustness testing. However, they need not necessary be dedicated “load tools”. A tool that acts as an input generator for functional testing (generating inputs for functional test cases – one at a time), may often be used as a load generator as well (generating a lot of inputs in order to verify performance attributes of the system). Producing load is only half of the story. System responses must be captured, measured, analyzed and compared to the expected outputs. The number of executed transactions are logged. Many performance testing tools provide graphs of load against response times. The most important result from the performance testing is finding bottlenecks in the system. Examples of performance test tools: LoadRunner/Mercury, Performance Center/Mercury, QACenter Performance Edition/Compuware, SilkPerformer/Segue and Vantage/Compuware.

- Input generation or driver stimulation of test object

Systems can be loaded either through their ordinary input channels (like sending many http-requests to a Web server), or through the “backdoor” (using ways not normally available to users in operational conditions). Beside inputs, load can be CPU-cycles, memory usage, data volume in a database, additional processes, etc. Load can apply to any resources present in the system.

- Level of stimulation

For example, the performance testing of a client/server system and a Web-application have much in common. The load for testing may be created all the way from the client side (by running an input generator on one or multiple copies of the client application), or between client and server (e.g. generator of http-requests for the Web-server or SQL requests for a database server application), or “inside” the system (e.g. between Web-server and database application).

- Environment load

The test environment used for performance testing must have the same parameters as a real production system. The load generators are often separated from this environment so as the results are not degraded by the generators themselves.

- Logging for debugging and result evaluation

Like for tracing, data is recorded in a buffer. The difference is in the goal rather than in the technique: the data is gathered specifically for test outcome evaluation, not “in case a failure occurs”, as is the case with tracing. Hardware tools are commonly used for tracing and logging, as they do not influence the behavior of the test object and are often faster.

- On-line and off-line performance analysis

Performance analysis may be made on-line in a monitoring-like way, or done off-line by the analysis of test logs. This depends on the amount of data logged, which often prevents to do the analysis on-line. If the goal is monitoring the production system, we prefer on-line measuring tools.

- Graph and load/response time relation

The greater load (throughput per unit of time), the longer response delay. The actual test is whether or not the maximum required throughput can be achieved before the maximum allowed delay is passed.

- Used mainly for non-functional testing... but even for “background testing”

Load generators and measurement tools are naturally used mostly for non-functional testing of performance attributes, but may also be used for the test of error handling (*stress testing* – what happens if the load is too much?) or for *background (long duration) test* – repeating some functional tests with a “typical” load in the background for many days.

Nástroje pro dynamickou analýzu

- Informace o stavu softwaru v průběhu jeho spouštění
- Hledání vedlejších efektů (side-effects)
- Využití paměti (zápis do “špatné/neplatné” paměti)
- Memory leaks (alokace a uvolňování paměti)
- Nepřiřazené ukazatele, aritmetika ukazatelů
- <https://visualvm.github.io/>



05/04/22

TSK

235

Examples of dynamic analysis tools: BoundsChecker/Compuware, Cantata/IPL, GlowCode/Electric Software, WinTask/UniBlue.

- Hunting side-effects

The expected outcome of a test case is one or just a few, but there can be an infinite variety of “unexpected” (i.e. wrong) outcomes. The tested application – besides doing what is expected of it – must not do anything else.

For example, a routine must not change any other memory contents than its own variables.

However, consequences of such a fault may be far-away (in time) and not at all immediately visible. Dynamic analysis is a set of testing techniques that specialize in looking for faults not immediately visible from the analysis of the expected outcomes.

- Memory usage

If a routine by mistake writes its data over another routine’s memory area, the result is a “dynamic fault”, i.e. the fault which is not permanently present in the executable code. Such faults may be captured using the operating system’s own memory control mechanisms. Sometimes finding them requires the usage of special tools.

- Memory leaks

Faults in the handling of dynamic memory are common, but their immediate effects are hardly visible. Only after prolonged execution do they cause a failure, when no more memory is available. There are tools able to identify and localize such “memory leaks”, i.e. situations when not all allocated memory is returned when de-allocated.

- Unassigned pointers, pointer arithmetic

The pointer is a variable that contains the address of a memory area (often dynamic memory). If a memory address is not assigned to the pointer (or the memory address is wrong, e.g. the address of dynamic memory which have already been de-allocated), writing to such address is the disaster.

Ladící nástroje

- “Nejvíce využívané programátory k reprodukci selhání a zjištění stavu programu”
- Používané ke kontrole nad během programu
- Především pro ladění, nikoliv testování
- Ladící skript příkazů může být využito při automatizovaném testování
- Ladící nástroj lze použít jako simulátor provádění
- Rozdílné typy ladících nástrojů



05/04/22

TSK

236

- Used to control program execution

Executing programs line-by-line, halting execution, setting and examining program variables, etc.

- Debugger command scripts can support automated testing

Most debug tools have the ability to execute commands read from files. Such command scripts can be useful as low-level test programs (especially for white-box testing).

- Debugger as execution simulator

Debuggers are sometimes available on a simulator platform on a host machine. This facility can be used for input generation and as test-running tool.

- Different types of debugging tools

- HW-tools (ICE, JTAG, logic analyzer)
- Assembly language level or source language level: for white-box testing, it is important that debugger understands the source code
- Tools understanding OS, processes, etc.

Porovnávací nástroje

- Slouží k detekování rozdílu mezi současným a očekávaným výsledkem
- Očekávané výsledky zadržované nebo v souboru
- Očekávané výsledky mohou být výstupy (GUI, konzolová aplikace nebo jakékoliv jiné rozhraní)
- Výsledky mohou být komplexní/složité:
 - On-line porovnávání
 - Off-line porovnávání



Comparison tools compare *actual* with *expected* outcomes. “Comparison tools” are seldom found as stand-alone applications, instead they are typically built into available test-running tools.

- Expected results hard-coded or from file

Comparing files or database contents require being able to deal with a range of file and database formats.

- Expected results can be outputs (GUI, character-based or any other interface)

Outcomes can be outputs, e.g. character screens, GUI objects, bitmap images, and many, many other. Comparison tools for special interfaces, for the analysis of proprietary protocols and for the evaluation of test outcomes that require domain knowledge, are often developed locally and tailored to their applications.

- Results can be complex

For the analysis of complex data, comparison tools often have filtering and masking capabilities. For textual output, chosen lines or columns of text can be ignored. For GUI output, some objects or areas of the screen can be ignored.

Nástroje pro správu testů

Velmi široká kategorie nástrojů a funkcí/aktivit:

- Správa Testware
- Monitorování, plánování a kontrola stavu testů
- Management a plánování testového projektu
- Management incidentů
- Analýza výsledků a vytváření zpráv



05/04/22

TSK

238

Examples of test management tools: DigitalTester/Digital Tester, QADirector/Compuware, SilkPlan Pro/Segue, TestDirector, QACenter/Mercury and TestManager/IBM Rational.

- Testware management

Tools are concerned with the creation, management and control of test documentation (test plans, specifications and results).

- Monitoring, scheduling and test status control

Test management tools shall facilitate these activities by providing readable statistics (if possible presented graphically) on the number of test cases run and passed, trends, risk levels, etc.

- Test project management and planning

For general management (time, resources, activities, risk management, etc.), ordinary project management tools are often applicable.

- Incident management

Some commercial test management tools have built-in functionality for incident management. Normally, separate incident management tools are used.

- Result analysis and reporting

Test management tools must either contain or have access to complete test execution record information (often stored in a test results database).

Nástroje pro zjišťování pokrytí

- Měření struktury pokrytí testy
- Provedení instrumentace před spuštěním
- Dynamika: měření v průběhu provádění
- Různé míry pokrytí
- Jazykově závislé
- Obtížné použití pro embedded systémy
- Použití: měří kvalitu testů



Coverage tools measure the percentage of the source code covered, i.e. executed at least once, during the test execution. Examples of coverage tools: Bullseye Coverage/Bullseye Testing Technology, Cantata/IPL, Code Coverage/DMS, CTC++/Testwell, LDRA Testbed/LDRA, Panorama/ISA Inc and TCAT C/C++/Software Research.

- Pre-execution - instrumentation

To measure the coverage, the source code must first be instrumented. Additional instructions are inserted into it.

- Dynamic – measurement during execution

During execution, each time such an extra instruction is executed, the corresponding counter is incremented. After the execution, by examining the values of the counters it is possible to determine the coverage and other statistics.

- Various coverage measures

There are many different measures of structural coverage.

- Language-dependent

The coverage measurement requires that the tool understands the language in which the tested application is written.

- Difficulties for embedded systems

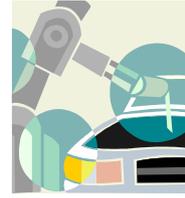
Embedded systems make the measurement of coverage difficult. There may not be enough space in memory for the instructions added during instrumentation and no source code information.

- Usage: measuring the quality of tests

The goal of measuring coverage is to measure the quality of tests. During test suit design, tests shall be added to improve the coverage until the required level is achieved. Note that structural coverage should be interpreted more cautiously for real-time and for object-oriented systems than for sequential, one-process systems.

6.2 - Výběr nástrojů a implementace

Které aktivity automatizovat?



- Identifikujte výhody a určete priority
- Automatizujte administrativní aktivity
- Automatizujte neustále se opakující aktivity
- Automatizujte to co je snadné zautomatizovat (alespoň na začátku)
- Automatizujte tam kde je to nezbytné (př. výkonnostní testy)
- NENÍ nezbytné automatizovat provádění testů!
- Patrně NE návrh testů; do not “capture”

05/04/22

TSK

240

- Automate administrative activities

The more a given test-related activity is “clerical” and the less it is “intellectual”, the easier and more rewarding it is to automate.

- Automate repetitive activities

Automation pays best for repetitive activities. They can be identified as those considered as “boring” and done very unwillingly.

- Automate what is easy to automate (at least at the beginning)

For the successful introduction of automation, it is best to begin with a small-scale pilot project, achieve early success and promote it.

- Automate where necessary (e.g. performance)

Most activities can be automated, but some have to be automated, e.g. performance testing or coverage analysis. These are good entry points for the usage of test tools and test automation.

- NOT necessarily test execution!

Test tools are not only test-running tools. Automated test execution may be harmful unless more basic parts of test and surrounding processes are automated.

- Probably NOT test design: do not “capture”

Unless your product and your processes are very mature, and formal design methods in place, do not automate test design. Automated test case generation using “capture” may look deceptively easy, but can be very harmful and create a non-maintainable testware architecture.

Automatizovaný chaos = Rychlejší chaos

- Automatizace vyžaduje vyspělý proces testování
- Úspěšná automatizace vyžaduje:
 - Psanou specifikaci testů
 - Znalost očekávaných výsledků
 - Management konfigurací pro testware
 - Management incidentů
 - Relativně stabilní požadavky
 - Stabilní organizace a zodpovědnost za testování



05/04/22

TSK

241

- Automation requires mature test process

Test process must be mature, systematic and disciplined, otherwise it is not only not going to work, but make matters even worse. If this condition is not fulfilled, process improvement must be done before or at least in parallel with test tool introduction. How given tools fit your test process is at least as important as their functionality when choosing tools.

- Written test specifications

Automation without written specification what to automate may be fun, but seldom successful.

- Known expected outputs

Without them, manual testing is better. The automation of test execution without the automation of result evaluation will produce mountains of test results to evaluate and a false sense of security.

- Incident Management

As mentioned before, successful automation of test execution requires that more basic activities are already automated.

- Relatively stable requirements

Unless the requirements are relatively stable, testware maintenance will soon become a major issue. The choosing of test tools and building of test environment may not be possible if too little is known about the test object.

- Stable organization and test responsibility

Test automation is time-consuming and requires careful planning and dedicated resources. Automation is seldom successful if it uses project resources only. Dedicated line resources are needed to keep and transfer automation and tool competence between projects.

Výběr nástroje: kritéria

- Co nyní potřebujete
 - Detailní seznam technických požadavků
 - Detailní seznam netechnických požadavků
- Dlouhodobá podniková strategie v automatizaci testování
- Integrace s vaším testovacím procesem
- Integrace s ostatními (testovacími)nástroji
- Čas, rozpočet, podpora, školení



05/04/22

TSK

242

- Detailed list of technical requirements

All technical details on the planned scope and goal of the automation project will be needed for the decision.

- Detailed list of non-technical requirements

The introduction of test tools is not only a technical decision, but business and organizational decision as well.

- Long-term automation strategy of your organization

Your test tool introduction plans should be synchronized with your company's overall test strategy and QA system. If there are non, automation may succeed but anyway become "shelfware" because of the lack of long-time commitment.

- Integration with your test process

If you do not have one, get one first – then start talking about automation again.

- Integration with your other (test) tools

Using test tools is supposed to save the overall amount of work, not create additional work because of non-compatible tools. Take into account tool's integration with your development and execution platforms. Consider integrated tools offered by some vendors.

- Time, budget, support, training

A complete project plan must be prepared and used for the introduction of test tools and test automation.

Výběr nástrojů: 4 kroky

- 4 etapy výběrového procesu podle ISEB:
 - Vytvoření krátkého seznamu kandidátů
 - Sjednání předvedení
 - Ohodnocení vybraných nástrojů
 - Přezkoumání a výběr nástroje



Implementace/nasazení nástroje

- Je to vývoj – použijte standardní vývojový proces!
- Plánování zdrojů, podpora managementu
- Podpora a mentoring (razení od odborníků)
- Školení
- Zkušební projekt
- Včasné zhodnocení
- Prezentace prvních úspěchů
- Úprava testovacího procesu



05/04/22

TSK

244

- *This is development – use your standard development process!*

A good and defined development process applies to testware as much as it applies to the product. Unreliable testware will not provide reliable test results.

- *Plan resources, management support*

Test automation can save execution time (and resources in the later stage), but requires resources (for script development) and full management support. Shall not be attempted as an amateur activity.

- *Support and mentoring*

Access to support (tool experts and test automation experts) and mentoring for the new test automation team, is vital for success).

- *Training*

Most often forgotten or underestimated in test automation projects. Plenty of training is needed, both for testers and for developers.

- *Pilot project*

To gather experience, to allow early evaluation, to provide training and to achieve early success – all that is easier to achieve if automation is started on a narrow front only.

- *Early evaluation*

After some time into the test automation implementation, evaluate costs and benefits, identify changes in the test process, perform risk analysis again. This may help put efforts back on track, and cancel what holds a very poor promise of success.

- *Publicity of early success*

It is an important step in assuring future resources, diminishing defensive attitudes and going over from project to line activity.

Zkouška a zavedení

- Zkušební cíle
 - Zkušenosti s nástrojem
 - Identifikace požadavků na změnu testovacího procesu
 - Zhodnocení skutečné ceny a výhod
- Podmínky zavedení (Roll-out pre-conditions)
 - Na základě pozitivního ohodnocení zkušebního projektu
 - Dohoda se zákazníkem
 - Dohoda s managementem a propojení s projektem



Výhody automatizace

- Rychlejší a levnější
- Lepší přesnost
- Stabilní kvalita testování
- Automatické logování a vytváření zpráv
- Kompletnější regrese
- Uvolnění lidských zdrojů
- Lepší pokrytí
- Více negativních testů
- “Nemožné” testy
- Spolehlivější a kvalitnější odhady

- Faster and cheaper

Tools generally work faster, without breaks and 24x7. If the test automation is applied to the right project and at the right time, less resources are needed – so it is cheaper.

- Better accuracy

Tools can measure what we cannot even see and they make no errors.

- Stable test quality

Tools work in exactly the same way each time, do not get tired, have no personal problems and, above all, do not get bored.

- Automatic logging and reporting

This is the most “clerical” of all test activities, therefore most repetitive and boring. Can be (partially) automated without automating test execution.

- More complete regression

Faster execution allows for more extensive regression, which in turn supports more flexible policy for adding functionality, faster fault-fixing, etc.

- Liberation of human resources

Testers are relieved of boring work and free to do what people are best at: intellectual work.

- Better coverage

Testing faster means testing more.

- More negative tests

More testing of “crazy” boundary conditions.

- “Impossible” tests become possible

I.e. tests that require speed or volume not possible for human testers (e.g. load tests).

- Reliability and quality estimates

By performing tests each time in the same way, and performing more regression testing, data for process analysis and improvement becomes available.

Nástrahy automatizace

- Více času, peněz, zdrojů!
- Neexistující testovací proces
- Chybějící požadavky
- Chybějící specifikace testů
- Chybějící výsledky
- Špatný management konfigurací (CM)
- Špatný management incidentů
- Obranný postoj
- Příliš málo regrese
- Exotické cílové systémy
- Nekompatibilita s ostatními nástroji
- Potřeba vlastního nástroje
- Mnoho školení
- Mobilita testovací laboratoře
- Záchrana zpožděného projektu

05/04/22

TSK

247

- More time, money, resources!
Automated testing is more expensive than manual. It requires expensive tools, test specialists (developers) and time to develop test scripts in the development phase. On the other hand the automation is effective if some conditions are fulfilled.
- No test process
Test process needs to be specified especially when test tools is used.
- Missing requirements
Without having clear requirements we cannot do good automation.
- Missing test spec
Automated testing requires the same test specification (designed test cases) as manual testing.
- Missing outcomes
Expected outcomes are the most important inputs to test case design and test scripts development.
- Poor CM
Testware is like any other software and must be under CM (the version of test scripts must be in compliance with the version of the system under test).
- Poor incident management
Defects found by test scripts must be reported to development and actions have to be taken. Reporting of incidents can be automated.
- Defensive attitudes
Sometimes testers or managers feel threatened by the prospect of test automation. Even if you do not agree, they must not be dismissed. Usually, there is some rationale behind this attitude.
- Too little regression
Unless the amount of regression testing is large enough and testing is repeating ≥ 3 times, the automation of test execution may not be profitable, even if it is technically tempting. However, this need not apply to other forms of automation.
- Exotic target system
Means that no commercial tools are available. Building your own tools is a major undertaking and must be planned carefully.
- Incompatible with other tools
You may already have tools in place that would not work with your test-running tool. A case for re-consideration.
- Own tools needed
In some cases development of your own tools is inevitable or more effective.
- Much training
If your test team is generally inexperienced, and no experts on test automation are available, trainings and consultations are needed (both internal, external).
- Mobile test lab
If the mobility of the test lab is very important, adding more instruments may not be feasible.
- To save late projects
Test automation requires more resources and time up-front, and brings about savings in test execution time later on. Therefore, an attempt to save projects already late by bringing test automation in, is counterproductive.

Automatizace v různých úrovních testování

- **Komponentní:** kořeny a ovladače, analýza pokrytí
- **Integrační:** rozhraní a protokoly
- **Systemové:** automatizované spouštění testů a výkonnostní testování
- **Akceptační:** požadavky, instalace, použitelnost

The strength of the case for test automation is not related to the level of testing. However, different automation goals and techniques may dominate on different levels, as the scope and goals for test on different levels are different.

The amount of white-box testing is greatest in component testing. The amount of black-box testing is greater in system than in component testing, but the difference is not as distinct as for white-box testing.

TSK - cvičení

