Test 1 teorija

Važnost Large scale Programing

- 1. Softver kao ključni faktor u **finansijama kompanija i nacionalnoj ekonomiji.** Veliki softveri zahtevaju dosta novca jer su projekti dosta skupi kao i **održavanje** velikih sistema može dosta da košta. Primer propalog sistema je ALS - Advanced Logistics System – posle potrošene velike sume novca došlo je do zaključka da ne valja i bilo je jeftinije napustiti ceo projekat i početi ponovo nego popravljati ono što ne valja.

- 2. Socijalna važnost LSP – računari su sve više prisutni u društvu, tako da greške koje mogu da se nađu u softveru mogu dovesti do katastrofalnih posledica.
Npr. 1. Medicinska tragedija: pacijent koji ima rak je dobio smrtonosnu dozu radijacije jer je računar naredio mašini da to da. Slično tome aparati za dijabetičare ukoliko pogrešno izračunaju dozu insulina.
2. **Gubitak misije u svemiru:** misija na Mars je propala zbog **nedostajuće zareze u Fortran petlji**.

Stanja projekta:

- Početna ideja – initial conception
- Analiza zahteva
- Specifikacija
- Visok nivo dizajna – Arhitektura
- identifikacija modula
- Detaljan dizajn i testiranje modula
- Integracija sistema i testiranje
- Testiranje prihvatanja
- Korišćenje i „održavanje“ – isptravljanje grešaka, unapređivanje dodavanje funkcionalnosti/karakteristika- zamena za novim sistemom
- Mogu postojati i druge faze, npr. prototipizacija. Ovo može trajati 25 ili 30 godina, a često je održavanje dominantni trošak!

Problemi sa LSP-om

1. Procena koliko će koštati i dužina velikog projekta je teška, a softverski projekti su sami po sebi teški za upravljanje.

2. Ne programiranje- većina vremena je na analizi i dizajnu, manje na kodiranju pa opet više na testiranju i integraciji.
Progblemi kao što su ustanovljivanje zahteva, namestiti da različiti delovi sistema(napisani od strane različitih ljudi) da rade zajedno i održavanje.

3. Problemi sa održavanjem: obuhvata preko 50% ukupnog truda- Održavanje bolje nazvati evolucijom.
- Cilj samog softverskog inženjerstva je da se razvoj sprovodi tako da održavanje bude relativno jednostavno, što znači da:
    a. Softver mora viti jednostavan za menjanje(modularan).
    b. Dizajn, posebno dizajnerske odluke moraju biti dobro dokumentovane.
    c. Mora biti moguće praćenje zahteva do implementacije.

1. What are the factors that influence software productivity?

4. **Produktivnost**
- Što se tiče profesionalnih programera tiče ona može da varira 10:1.
- Idealno je koristiti male timove iskusnih ljudi.
- Produktivnost se smanju sa veličinom programa.
    effort = constant * size1.5 or
–   productivity = constant * size−1.5

- **produktivanost pod uticajem tehnologije –**
    produktivnost je uglavnom nezavisna od programskog jezika (linije/dan). – stoga naglasak na jezicima višeg nivoa. (npr. Java, Python) koji omogućavaju **brži razvoj i bolju čitljivost**, čak i ako ne povećavaju broj linija koda.)
- produktivnost varira u zavisnosti od
    Ljudi – veština, iskustvo i timska saradnja.
    - **Veličina projekta** – manji projekti mogu biti agilniji, veći zahtevaju bolju organizaciju.
    - **Tehnologija razvoja** – alati, framework-ovi, razvojna okruženja i metodologije (npr. Agile, DevOps).

- **Uticaj prirode projekta na produktivnost –** što je veći broj interakcija između programera to produktivnost opada. Tu imamo i podelu projekta na
a) **Savršeno deljiv projekat** gde se posao može podeliti između više ljudi.
    - **Ukupan napor(effort) je konstantan,** ali vreme izvršenja je obrnuto proporcionalno broju ljudi.(npr ako dvostruko više ljudi radi na projektu vreme se prepolovi)
b)**Potpuno nedeljiv projekat** (kao trudnoća):
    - Ne može se paralelizovati
    - **Vreme je konstantno**, ali **napor raste proporcionalno broju ljudi**, jer dodavanje više ljudi ne ubrzava proces, ali povećava koordinaciju i komplikacije.

Chatgpt zaključio

Produktivnost **zavisi od tipa projekta**: količine interakcija, kompleksnosti i deljivosti posla.
Veći broj ljudi ne znači automatski veću brzinu, naročito kod složenih ili nedeljivih projekata.
Generalno projeti nisu lako deljivi.
- Vreme potrošeno u komunikaciji raste sa brojem ljudi.
- Brook.ov Zakon – Dodavanje još ljudi na projekat koji kasni može samo da izazove to da projekat još više kasni.
- Nije praktično povećavati veličinu timova preko 30%.
- Organizacija softverskih projekata je kritična za uspeh.

U suštini produktivnost zavisi od više faktora, teško je predvideti, i teško je kontrolisati.
Trošak komuniciranja je visok pa je bolje razbiti sistem na što veće nezavisnije module kako bi obezbedili uspeh u LSP-u

summary 21
- Postoje više fundamentalnih problema sa LSP-om kao što veličina(kompleksnost), problem raste linearno brže sa veličinom programa, tačnost
- **Produkcija**- pravljenje velikih sistema zahteva dobar menadžment kao i metode za razvoj i tehnike.
- **Promenljivost**- softver je lako menjati, prilagođavati. U praksi interakcije između delova sistema otežava menjanje softvera(mala izmena u jednom delu može izazvati nepredvidive posledice u drugima

2. What is meant by software “decays”?
- Softver postaje sve kompleksniji, struktura mu je lošija i to je čest probleme sa sistemima koji imaju dosta interakcija u smislu promena na jednom mestu dosta utiče na druge delove sistema. Gde bi svaka promena povećavala rizik za greskama i dodatno narušila strukturu, jedan modul poziva drugi i sl.

Nedeljivost- softver je skup ideja za dizajn i razloga za njima, nije samo program kao skupina koda. Vizualizacija pomaže, ali nikad ne može u potpunosti da prikaze razloge zbog kojih su određene odluke donete, nisu vidljivi direktno.
Ovi problemi se nasleđuju u softveru.

Životni ciklus Modela
Softversko inženjerstvo zagovara:

- **disciplinovan pristup** razvoju softvera
- koji obuhvata **više faza aktivnosti**

Model životnog ciklusa (LCM):

- daje **apstraktan opis** tih faza
- pomaže u **razumevanju i diskusiji** o aktivnostima razvoja softvera

Postoji mnogo LCM-ova. U osnovi:

- LCM-ovi koji naglašavaju **tehničke aktivnosti(programiranju testiranju, arhitekturi, waterfall, v model...)**
- LCM-ovi koji naglašavaju **menadžerske odluke i aktivnosti(planiranje, donošenje odluka-Agilni pristup ili SCrum)**

3. Why do we model?
Zašto modelujemo?

- Obezbediti strukturu za rešavanje problema(podeliti problem na delove, odrediti koraci..)
- Eksperimentisati kako bi se istražila višestruka rešenja(lakše je menjati model nego gotov softver)
- Obezbediti apstrakcije za upravljanje složenošću(sakriva nebitne detalje, samo ono što je važno fokus)
- Smanjiti vreme izlaska poslovnih rešenja na tržište(ranije uočavamo probleme, posle manje popravki)
- Smanjiti troškove razvoja(isto rano otkrivene greške u modelu lakse i jeftinije popraviti nego u kodu)
- Upravljati rizikom grešaka

Tehnički LCM

Šta se dešava?
– Konceptualni jaz između ideje (koncepta) i programa je prevelik da bi se premostio u jednom koraku.

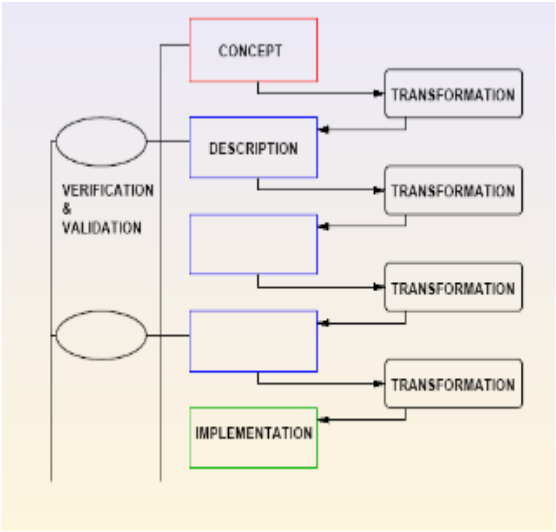
Uopšteno:
– postoji više opisa ili reprezentacija programa između koncepta i implementacije
– rani opisi su apstraktni — sadrže malo detalja
– kasniji opisi su konkretniji — sadrže više detalja

Govori se o **apstrakciji** i **reifikaciji**.

Tipični opisi:

- **Zahtevi (Requirements)** – ono što korisnik želi i okruženje u kome sistem radi
- **Specifikacija (Specification)** – definicija onoga što treba izgraditi
- **Arhitektura (Architecture)** – opis dizajna na visokom nivou koji prikazuje glavne podsisteme i podelu na module
- **Detaljni dizajn (Detailed Design)** – detaljno funkcionisanje svakog modula
- **Izvorni kod programa (Program Source Code)**
- **Izvršni kod (Executable Code)**

Kako prelazimo sa jednog opisa na sledeći?



Resenje su transformacije

Generalno o transformacijama

- Uzimaju jedno ili više opisa i daju neki drugi rezultat.
- Dodaju informacije, npr. kompajler transformiše **izvorni kod** u **objektni kod** i dodaje informacije o ciljnoj mašini i semantici jezika.
- Nisu nužno automatski izvodive, uključuju:
  – dizajnerske odluke
  – procenu kvaliteta (estetiku)
- Teške transformacije su:
  – koncept → zahtevi
  – zahtevi → arhitektura
- Ako bi sve transformacije bile tačne, razvoj softvera bi bio jednostavan – međutim, greške se često dešavaju, pa je potrebno proveriti da li napredujemo ispravno.

Tipovi iteracije

- **Lokalna:** ponoviti trenutnu transformaciju da se ispravi greška
- **Šira:** ponoviti raniju transformaciju i preraditi je
- **Globalna:** preispitati koncept!

Održavanje – iteracija nakon isporuke

- **Korektivno:** uklanjanje grešaka
- **Adaptivno:** dodavanje nove funkcionalnosti
- **Perfektivno:** poboljšanje načina izvođenja postojećih funkcija
- Veći deo napora ide na **perfektivno održavanje**

Iteracija ne mora biti na potpunim opisima

- Inkrementalni razvoj
- Prototipizacija

Verifikacija vs. validacija i testiranje

- **Verifikacija** potiče od latinskog „verit“ – istina
  Verifikacija znači proveru da li je jedan opis **usklađen sa drugim (ranijim) opisom**.
- **Validacija** potiče od latinskog „valere“ – vrednost
  Validacija znači proveru da li je opis **usklađen sa početnim konceptima**.
  Validacija je potrebna **na svakom koraku** dok se dodaju nove informacije, jer ne možemo biti sigurni da je prvi opis obuhvatio sve zahteve.

Potpunost (Completeness)

- Takođe je potrebno proveriti **potpunost** – tj. da li opis sadrži sve što treba. Ovo je **suptilnije nego što izgleda**.
- Potpunost znači: opisuje sve što treba:
  – **U zahtevima:** kako znati da li je sve obuhvaćeno?
  – **U dizajnu:** ponekad želimo odložiti dizajnerske odluke (npr. apstraktni tipovi podataka) – kako razlikovati da li je informacija **izostavljena slučajno ili namerno**?
  – **Na nekom nivou apstrakcije:** kako znati da li je **odabrani nivo ispravan**?

nalitička potpunost

- Kada su date funkcije/podaci i slično, cilj je da **sve aspekte opišemo**, ali ovo može biti u konfliktu sa odlaganjem dizajnerskih odluka, npr. **ne odlučivati o formatu poruka o greškama dok se ne krene sa implementacijom**.

Potpunost je subjektivna!

- To je **vrednosna prosudba** – može se smatrati **aspektom kvaliteta**.

Šta je Metod?

- **Metod** je ime dato za pristup određenoj fazi u **životnom ciklusu** softvera.
- **Metodologija** znači – **nauka o metodu**:
  – sistematska, teorijska analiza
  – teorijska osnova za razumevanje **koji metod / skup metoda** može biti primenjen u određenom slučaju.
- Pojam metodologija se često **pogrešno koristi** da znači skup metoda koji pokriva ceo (ili većinu) životnog ciklusa.

Metod ima tri komponente:

1. **Notacija** – za pravljenje opisa
2. **Skup smernica** – za sprovođenje transformacija
3. **Skup pravila (i smernica)** – za sprovođenje **verifikacije i validacije**

- Ove komponente ćemo razmotriti kada budemo pričali o **STATEMATE**.

Formalni vs. Struktuirani Metodi

- U principu, postoje dve klase metoda:
  – **Formalni Metodi (FM)**
  – **Struktuirani Metodi (SM)**
- Nedavno se pominju i **polu-formalni metodi**, koji integrišu oba pristupa.
- **Formalni metodi** imaju **čvrstu matematičku osnovu**.
- **Struktuirani metodi** su dobro definisani, ali **nemaju čvrstu matematičku osnovu** za potpuno opisivanje funkcionalnosti.
- Tehnički:
  – Formalni metodi omogućavaju **precizno specificiranje funkcionalnosti**
  – Struktuirani metodi omogućavaju **precizno specificiranje strukture sistema**, ali ne i funkcionalnosti.

Zašto koristiti formalne metode?

- Tokom **formalizacije** mogu se otkriti **dvosmislenosti, izostavljanja i kontradikcije** u neformalnom opisu problema.
- **Formalni model** se može dokazati **ispravnim pomoću matematičkih metoda**.
- Formalno specificiran sistem se može analizirati da li poseduje ili ne poseduje željene osobine.
- Formalno verifikovani podsistem se može **pouzdano ugraditi** u veći sistem.

- Formalni model vodi ka **delimično automatizovanim metodama razvoja i alatima**, npr. simulacijama.
- Moguće je **uporediti više dizajna** međusobno.

Opšte klase formalnih metoda (FM)

- **Model-based (zasnovane na modelu):** daju **eksplicitnu** (mada apstraktnu) definiciju **stanja sistema (programa)** i operacija koje transformišu stanje, ali **ne prikazuju eksplicitno konkurentnost** (npr. Z i VDM).
- **Algebraic-based (algebarske):** daju **implicitnu definiciju operacija** tako što povezuju ponašanje različitih operacija bez definisanja stanja, ali opet **ne prikazuju eksplicitno konkurentnost** (npr. OBJ i PLUSS).
- **Process Algebraic (procesno-algebarske):** daju **eksplicitni model konkurentnih procesa** i prikazuju ponašanje putem **ograničenja na dozvoljenu vidljivu komunikaciju između procesa** (npr. CSP i CCS).
- **Logic-based (zasnovane na logici):** opisuju **svojstva sistema**, uključujući **niskonivojsku specifikaciju ponašanja programa** i **specifikaciju vremenskog ponašanja sistema** (npr. Temporalna i Intervalna logika).
- **Net-based (zasnovane na mreži):** daju **implicitni model konkurentnog sistema** u vidu **(kauzalnog) protoka podataka kroz mrežu**, uključujući pravila **pod kojim uslovima podaci mogu teći sa jednog čvora mreže na drugi** (npr. Petrijeve mreže, predicate transition nets).

Kako ih koristimo?!

1. **Specifikacija kao precizan dokument**
    - Formalna specifikacija se koristi kao **precizan zapis šta sistem radi**.
    - Sistem se zatim gradi **bez direktnog oslanjanja na specifikaciju**.
    - Izgradnja se vrši **tradicionalnim tehnikama**.
    - Ako se sistem menja, **specifikacija mora biti promenjena u skladu sa tim**.
2. **Provera usklađenosti sa specifikacijom**
    - Koristi se kao gore, ali **osnovni formalizam se koristi da se pokaže da razvijeni sistem zadovoljava svoju formalnu specifikaciju**.
3. **Formalan razvoj iz specifikacije**
    - Koristi se kao ranije, ali **sistem se formalno izvodi iz svoje specifikacije**, kroz **korake koji čuvaju ispravnost**

Modelovanje zasnovano na stanjima (State-based modelling)

- Modelovanje zasnovano na stanjima se smatra **“sastavljačem” notacija i metoda modelovanja**.
- Svi poznati primeri gore navedenih klasifikacija mogu se **predstaviti/kodirati pomoću konačnih automata (FSM)** – što je **osnovna tema ovog modula**.
- **Teorija automata** je osnova **svih postojećih formalnih metoda i notacija**.
- Teorija automata pruža **sve neophodne mehanizme i tehnike** za **manipulaciju, konstrukciju, analizu i razmišljanje** o modelu sistema.
- **Konačni automati (FSM) su vaš jezik za modelovanje**.

Kako se koriste formalne metode?

1. **Napraviti (formalnu) specifikaciju i:**
    - Ostaviti je takvu kakva jeste.
2. **Napraviti (formalnu) specifikaciju i:**
    - Izgraditi softverski sistem koristeći **klasične metode** (struktuirane, objektno-orijentisane itd.)
    - **Formalno dokazati** da implementacija odgovara svojoj specifikaciji.
3. **Napraviti (formalnu) specifikaciju i:**
    - **Postepeno je razvijati u sistem** – potrebni su dodatni formalizmi (**transformacioni sistem**).

Šta je sistem?

Sistem je **sastavljen i relativno nezavistan celina** koja može **razmenjivati (materiju, energiju, informacije) sa svojim okruženjem**.

Dva ključna elementa u ovoj definiciji su:

1. **Skup komponenti** koje čine **povezanu celinu**
2. **Razmena sa okruženjem**

Okruženje je važno i može **uticati na (dizajn) sistema** – par **pretpostavka / obaveza** (npr. automati za prodaju).

Definicija
Sistem Sastavljena i relativno nezavisna celina
Okruženje Sve što je izvan sistema, sa čime sistem razmenjuje informacije, energiju ili materiju

Automat za prodaju (sistem) i prostorija u kojoj stoji (okruženje)

4. What are the main differences between environment and system? Use examples to illustrate your answer.

Sistem vs. Okruženje – primer automata za kafu

1. Sistem (automat za kafu)

- Sve što je **unutra u automatu** i što direktno učestvuje u pravljenju kafe:
    - Mehanizam za kovanice (prima novac)
    - Dugmad za izbor vrste kafe
    - Mehanizam za mlevenje i kuvanje kafe
    - Rezervoari za vodu i kafu
    - Isporučni mehanizam za šolju

Šta sistem radi:

- Prima input (novac i izbor kafe)
- Izvršava interni proces: proverava uplatu, bira količinu i vrstu kafe, kuvanje i točenje kafe
- Daje izlaz: isporuka kafe

---

2. Okruženje (sve što nije automat, a utiče na njega)

- Korisnik – ubacuje novčiće, bira vrstu kafe
- Struja – omogućava automat da radi
- Ambijent – temperatura i vlaga mogu uticati na rad mehanizma
- Šolja i pribor – fizički elementi koji omogućavaju isporuku kafe

Šta okruženje radi:

- Šalje input sistemu (novac, izbor)
- Prima output od sistema (šolja kafe)
- Može indirektno uticati na sistem (npr. hladna soba usporava grejanje vode

Kako karakterisati sisteme?

Sistemi se često dele u dve kategorije da bi se napravila **razlika između jednostavnijih i složenijih sistema**.

Primeri ove podele su:

- **Sekvencijalni vs. paralelni**
- **Centralizovani vs. distribuirani**
- **Deterministički vs. nedeterministički**
    - Jedan očekivani rezultat vs. više mogućih rezultata
- **Završni (terminirajući) vs. nezavršni (neterminirajući)**
    - Npr. kompajleri (završavaju) vs. nuklearni reaktori, berza (neprestano rade)

Sistemi se mogu deliti i na:

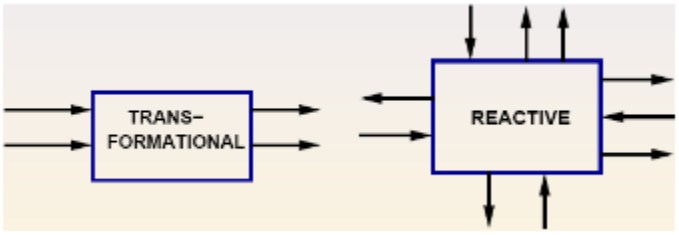
1. **Transformacijski (transformational) sistemi**
    - Pretvaraju **ulaz u izlaz** kroz neku definisanu funkciju ili proces.
    - Fokus je na **proračunu ili obradi podataka**, a ne na kontinuiranom praćenju okruženja.
    - Primer: **kompajleri, programi za proračune, konvertori fajlova**
        - Ulaz: izvorni kod → Izlaz: objektni kod
2. **Reaktivni (reactive) sistemi**
    - **Neprestano reaguju na događaje iz okruženja**.
    - Fokus je na **interakciji sa okruženjem u realnom vremenu**, često tokom dužeg vremenskog perioda.
    - Primer: **automat za kafu, sistemi za kontrolu saobraćaja, robotski sistemi**
        - Ulaz: korisničke akcije, senzorski podaci → Izlaz: reakcija sistema (kafa, svetlosni signali, pokreti robota)

Transformacijski sistemi

- Opisuju se preko **odnosa između početnog i odgovarajućeg konačnog stanja**; imaju **linearni tok**, jer su važna samo **početna i odgovarajuća konačna stanja**.
- Primeri uključuju: **algoritme sortiranja, kompajlere i druge algoritme koji računaju funkciju**.

Reaktivni sistemi

- Ne računaju funkciju, već su u **kontinuiranoj interakciji sa svojim okruženjem**.
- Primeri: **TV prijemnik, digitalni satovi, čipovi, interaktivni softverski sistemi, računarske igre** (npr. TrackMan, Monkey Island, Tomb Raider), ali i **monitor srca na intenzivnoj nezi**.



5. In what way do transformational and reactive systems differ? Use examples to illustrate your answer
6. Determine which of the following are reactive and which are transformational: – Traffic light controller – Vending machine – system for solving system of linear equations – Billing system – Library loan system – Operating system – Compilers – Police interrogation system – Parking Lot

Transformacijski / reaktivni sistemi

- **Transformacijski sistemi** su dobro proučeni; za njihovo programiranje i analizu postoji **mnogo dobrih jezika i metoda**.
- Objašnjavamo **zašto je jezik Statecharts dobar kandidat za specificiranje i programiranje reaktivnih sistema**.

Koristiti transformacijske tehnike?

Ako su **transformacijski sistemi** tako dobro proučeni, zašto **reaktivni sistem** ne smatramo transformacijskim?

- Mogli bismo jednostavno reći da **reaktivni sistem transformiše niz ulaza u niz izlaza**.
- To **ne funkcioniše zbog „povratne sprege“ (feedback)**, kao što ilustruje **Brock-Ackermann paradoks**.
- Na primer, razmislimo o **dva sistema: bafer sa jednim mestom i bafer sa dva mesta**
    - Ako ih posmatramo transformacijski, oba pokazuju **isti odnos početnog i konačnog stanja**, iako se njihovo ponašanje u interakciji sa okruženjem razlikuje.

7. In what way does a 3-place buffer differ from a 2-place buffer in the Brock-Ackermann Paradox?
Zaključak

- Ako se **izlaz ovih sistema vraća kao povratna sprega i kombinuje sa ulazom**, sistemi se ponašaju **drugačije**.
- Da bi se **karakterisalo ponašanje sistema koji komunicira sa okruženjem putem povratne sprege**, potrebno je **specifikovati relativni redosled izlaznih događaja u odnosu na ulazne događaje** (tj. **potrebno je znati kada se izlaz proizvodi**).

Ponašanje početnog i konačnog stanja

- Transformacijski sistemi imaju **linearnu strukturu**, i isto važi za konvencionalne jezike koji se koriste za njihovo specificiranje i programiranje.
- Ono što se opisuje je **kako se konačno stanje dobija iz početnog stanja**.
- **Relativno vreme** kada se međustanja izračunavaju nije važno, niti je važan njihov identitet, **sve dok je poznato odgovarajuće konačno stanje**.

Za **reaktivne sisteme** situacija je potpuno drugačija:

- **Trenutak dolaska novog ulaza** je relevantan za ponašanje sistema.
- **Unutrašnje stanje sistema** u trenutku ulaza je važno za reakciju sistema.
- Reaktivni sistemi možda **čak nemaju konačno stanje**!

Dakle, u reaktivnim sistemima **ne postoji glavni sekvencijalni tok kontrole** (kao kod transformacijskih sistema), i **izrazi/komande mogu imati više ulaznih i izlaznih tačaka**.

- Formalne metode poput **Z, CSP, Petri-nets** nisu pogodne za modelovanje reaktivnih sistema.
- Potrebne su **nove metode koje mogu da se bave vremenom i njegovim uticajem na relativni redosled događaja**.
- **Okruženje je još važnije kod reaktivnih sistema**, jer predstavlja **ključni deo interakcije** (za njega se može reći da je „prvi građanin“ sistema).

8. In developing a software system, we normally start with ‘concept’ and end by the ‘implementation’. How is the gap between the two phases is bridged?
9. In what way can formal methods be used? 7
10. What distinguishes formal methods from structured techniques?