11. DIZAJN ARHITEKTURE

Veliki se sustavi obično rastavljaju na podsustave koji pružaju neki povezani set usluga. Početni proces dizajna koji identificira ove podsustave i uspostavlja osnovu za kontrolu i komunikaciju tih podsustava naziva se dizajn arhitekture. Rezultat ovog procesa dizajna je opis arhitekture softvera.

Dizajn arhitekture je rana faza procesa dizajna sustava. Predstavlja kritičnu vezu između procesa specifikacije i dizajna. Često se izvodi paralelno sa nekim specifikacijskim aktivnostima. Uključuje identifikaciju velikih komponenti od kojih se sustav sastoji te komunikaciju među njima.

Prednosti eksplicitnog dizajniranja i dokumentiranja arhitekture softvera:

1. Komunikacija između stakeholdera – Arhitektura softvera je u biti high-level prezentacija sustava, o kojoj onda mogu raspravljati različite grupe stakeholdera.
2. Analiza sustava – Odluke koje se donesu kod dizajna arhtekture imaju velik utjecaj na to hoće li sustav biti u mogućnosti ostvariti ne-funkcionalne zahtjeve kao što su održljivost, pouzdanost i performanse.
3. Large-scale reuse – Arhitektura sustava je često jednaka za sustave koji imaju slične zahtjeve, pa je možemo često ponovo upotrebljavati.

Arhitektura sustava utječe na performanse, robusnost, distribuiranost i održavanje sustava.

Koji ćemo stil arhitekture upotrijebiti za sustav koji razvijamo ovisi o ne-funkcionalnim zahtjevima. Karakteristike arhitekture i sustava:

1. Performanse – Ako su performanse sustava kritičan zahtjev, onda bi arhitektura sustava trebala biti dizajnirana na način da lokalizira kritične operacije unutar malog broja podsustava među kojima postoji što manje komunikacije. Dakle, trebali bi se upotrebaljavati veliki, a ne mali podsustavi, a sve s ciljem da se smanji komunikacija među njima.
2. Osiguranje (security) – Ako je osiguranje kritičan zahtjev, trebala bi se upotrebljavati slojevita arhitektura kod koje bi kritični odsječci bili zaštićeni u unutarnjim slojevima strukture.
3. Sigurnost (safety) – Ako je sigurnost kritičan zahtjev, arhitektura bi trebala biti dizajnirana na način da se sve operacije povezane sa sigurnošću nalaze ili unutar jednog podsustava ili unutar malog broja podsustava.
4. Dostupnost – Ako je dostupnost kritičan zahtjev, arhitektura bi trebala biti dizajnirana na način da se uključe i redundantne komponente i da bude moguće zamijeniti ili obnoviti komponente bez zaustavljanja sustava.
5. Održavanje – Ako je održavanje kritičan zahtjev, arhitektura bi trebala biti dizajnirana na način da se upotrebljavaju male komponente koje se lako mogu mijenjati.

Konflikti kod arhitekture sustava: Upotreba velikih podsustava povećava performanse, ali smanjuje održavanje. Kod upotrebe malih podsustava je obrnuto, oni povećavaju održavanje, ali smanjuju performanse. Ako su i performanse i održavanje važni zahtjevi

za sustav, trebao bi se pronaći neki kompromis. Taj se kompromis može postići na način da za različite dijelove sustava upotrebljavamo drugačiji stil arhitekture.

Dizajn podsustava je apstraktna dekompozicija sustava na velike komponente (large-grain), od kojih svaka može biti sustav za sebe, tj. može se sastojati od drugih podsustava. Za opis dizajna podsustava koriste se blok dijagrami. Podsustavi su predstavljeni blokovima. Blokovi unutar blokova označavaju da je podsustav rastavljen na podsustave. Strelice među blokovima označavaju smjer toka podataka ili kontrolnih signala. Ovakav način prikaza ima nekoliko mana: ne pokazuje pravu prirodu odnosa između podsustava ni vanjska vidljiva svojstva komponenti, ali se ipak često koristi jer je jako jednostavan, isključuje detalje i može ga se lako razumjeti.

Kod donošenja odluke kako sustav podijeliti na podsustave, u obzir treba uzeti zahtjeve za taj sustav. Arhitekturu bi trebali dizajnirati na način da je svaki podsustav logički povezan sa određenom skupinom zahtjeva, tako da ako dođe do promjene u specifikaciji zahtjeva, ta se promjena neće propagirati na sve podsustave, već će biti lokalizirana na manjem području.

Odluke kod arhitekture dizajna

Arhitektura dizajna je kreativan proces kod kojeg se pokušava ostvariti takva organizacija sustava koja će zadovoljiti funkcionalne i ne-funkcionalne zahtjeve. Zato što je to kreativan proces, aktivnosti unutar tog procesa se razlikuju i ovise o tipu sustava kojeg razvijamo, o količini iskustva kojeg ima arhitekt sustava te o određenim zahtjevima za sustav. Zato se na proces dizajna ne gleda s perspektive neke određene aktivnosti, već s perspektive donošenja odluka.

Kod procesa arhitekture dizajna, arhitekt sustava mora donijeti niz odluka koje će jako utjecati na sustav i njegov razvoj. Ovisno o njihovom iskustvu i znanju, moraju donijeti odgovore na sljedeća pitanja:

1. Postoji li generička arhitektura aplikacije koja bi mogla poslužiti kao template za sustav koji se dizajnira?
2. Kako će sustav biti distribuiran na različitim procesorima?
3. Kakav stil (ili kakvi stilovi) bi bili prikladni za arhitekturu sustava?
4. Koji će biti osnovni način na temelju kojeg će se sustav strukturirati?
5. Kako će se podsustavi podijeliti u module?
6. Kako će se kontrolirati rad podsustava?
7. Kako će dizajn arhitekture biti procijenjen?
8. Kako bi se arhitektura sustava trebala dokumentirati?

Iako je svaki softverski sustav jedinstven, sustavi unutar iste aplikacijske domene imaju slične arhitekture. Mi možemo naš sustav usporediti s nekim sličnim sustavom, vidjeti što imaju zajedničko te odrediti koliko od tog znanja možemo upotrijebiti za razvoj našeg sustava.

Veliki sustavi su danas najčešće distribuirani, tj. softver je distribuiran na različitim računalima, pa je odluka kako ćemo sustav distribuirati na različitim procesorima također važan.

U praksi, sustavi nikada nisu bazirani na samo jednom stilu (modelu) arhitekture, već se uvijek radi o nekakvoj kombinaciji stilova (client-server organizacija, slojevita arhitektura).

Strukturu sustava moramo odabrati na način da sustav u konačnici zadovolji svoje funkcionalne i ne-funkcionalne zahtjeve.

Procjenjivanje sustava je u odgovor na pitanje kako sustav zadovoljava svoje funkcionalne i ne-funkcionalne zahtjeve.

Razultat arhitekture dizajna je dokument arhitekture dizajna, koji može sadržavati različite grafičke reprezentacije sustava i popratni tekst. Grafički modeli sustava pružaju različite perspektive na arhitekturu sustava (modeli arhitekture sustava):

1. Statički strukturalni model - opisuje koje podsustave ili komponente moramo razviti kao zasebne jedinice.
2. Dinamički proces model - opisuje kako je sustav organiziran u run-time procese i može se razlikovati od statičkog modela.
3. Model sučelja - definira usluge koje pružaju podsustavi putem svog korisničkog sučelja.
4. Relacijski model - pokazuje relacije, kao tok podataka, između podsustava.
5. Distribucijski model - opisuje kako su podsustavi distribuirani na različitim računalima.

Predložila se je upotreba ADL-a (Architectural Description Language) kao jezika koji bi opisivao arhitekturu sustava. Njegovi glavni elementi su komponente i konektori, koji uključuju pravila i preporuke za dobro formirane arhitekture. No, budući da s ovim jezikom mogu raditi samo stručnjaci, još uvijek se radije upotrebljava UML.

Organizacija sustava

Repository model (model zajedničke baze podataka) – Podsustavi trebaju međusobno izmjenjivati informacije da bi mogli efikasno raditi zajedno. To se može postići na dva načina: ili da postoji jedna zajednička baza podataka kojoj mogu pristupati svi podsustavi, ili pak da svaki podsustav ima svoju bazu podataka, a sa drugim podsustavima komunicira izmjenjujući poruke. Ovaj prvi način naziva se repository model, i uglavnom se koristi kod sustava koji rade s velikim količinama podataka. Dakle ovaj je model dobar za aplikacije kod kojih jedan podsustav generira podatke, a drugi ih upotrebljava. I ovaj sustav ima svoje prednosti i mane.

Ovo je dosta efikasan način za dijeljenje velikih količina podataka, jer nema potrebe za eksplicitnom transmisijom tih podataka među podsustavima. Međutim, svi podsustavi moraju pristati na isti model podataka koje im nameće zajednička baza podataka. Ovo može dovesti do problema, jer integracija novih podsustava neće biti moguća ako oni pak nisu kompatibilni s tim modelom podataka.

Podsustavi koji proizvode podatake se ne moraju brinuti kako drugi podsustavi te podatke koriste, ali prebacivanje informacija u nekakav novi model podataka bi moglo biti skupo, teško ili nemoguće.

Aktivnosti kao što su backup, sigurnost, kontrola pristupa ili oporavljanje od grešaka su sve centralizirane i odgovornost su menadžera baze podataka. Međutim, različiti podsustavi mogu imati različite zahtjeve za sigurnošću ili backupom, ali budući da su te aktivnosti centralizirane, svima su iste.

Lako je integrirati nove alate ako su kompatibilni s modelom podataka koji se koristi u zajedničkoj bazi podataka. Međutim, zajedničku bazu podataka može biti teško distribuirati na različitim računalima.

Client-server model (klijent-server model) – Model kod kojeg je sustav organiziran kao skup usluga i povezanih servera i klijenata koji tim uslugama pristupaju i koriste ih.

Glavne komponente ovog modela su: Skup servera koji pružaju usluge drugim podsustavima, skup klijenata koji traže usluge koje pružaju serveri, te mreža koja omogućava klijentima da pristupe tim uslugama. Mreža nije uvijek potrebna, jer se i klijent i server mogu nalaziti na istom računalu. Međutim, u praksi, većina client-server sustava je implementirana u obliku distribuiranih sustava.

Klijent ponekad mora znati imena dostupnih servera i usluga koje oni nude, međutim server ne mora znati identitet klijenata ni koliko ih ima. Klijent pošalje serveru zahtjev za uslugom koju on nudi i čeka na odgovor.

Najvažnija prednost client-server modela je da je to distribuirana arhitektura. Lako je dodavati nove servere i integrirati ih unutar sustava, ili pak update-ati neke servere bez da to utječe na čitav sustav. Međutim, da bi se mogle iskoristiti sve prednosti integracije novog servera, ponekad je potrebno napraviti neke promjene na već postojećim klijentima i serverima.

The layered model (model slojevite arhitekture) – Često se zove i model apstraktnih uređaja. On organizira sustav u slojeve gdje svaki sloj nudi određeni skup usluga. Na svaki sloj možemo gledati kao na apstraktnu 'mašinu', čiji je 'mašinski' jezik određen uslugama koje taj sloj pruža. Ovaj 'jezik' se upotrebljava da bi se implementirao novi sloj apstraktne mašine. Ovaj model podržava inkrementalni razvoj sustava. Kad se neki sloj razvije, neke od usluga koje on nudi mogu postati dostupne korisnicima. Ova arhitektura je promjenjiva i prenosiva. Sve dok sučelje ostaje nepromjenjivo, sloj se može mijenjati sa drugim, ekvivalentnim slojem. Kad se sučelje promijeni ili kad mu su dodaju nove usluge (facilities), to će utjecati samo na susjedni sloj. Mana ovog modela je da strukturiranje sustava na ovaj način može biti teško. Unutarnji slojevi mogu pružati neke osnovne usluge, kao file management, koje su potrebne svim slojevima. Kada korisnik zatraži uslugu od najgornjeg sloja, tada se ta usluga mora 'probiti' kroz donje slojeve da bi došla do sloja koji nudu neku osnovnu uslugu. Performanse također mogu biti problem. Ako postoji puno slojeva u sustavu, usluga koja je zatražena sa najgornjeg sloja, možda će morati biti interpretirana više puta u različitim slojevima dok se ne procesira. Da bi se ovi problemi izbjegli, aplikacije radije mogu komunicirati direktno sa unutarnjim slojevima nego upotrebljavati usluge koje im nude susjedni slojevi.

Stilovi modularne dekompozicije

Nakon odabira organizacije sustava, moramo odabrati način na koji ćemo podsustave podijeliti na module.

Podsustav je sustav sam za sebe, i njegov rad ne ovisi o uslugama koje pružaju drugi podsustavi. Podsustavi se sastoje od modula i imaju definirana sučelja koja koriste za komunikaciju sa drugim podsustavima.

Modul je komponenta sustava koja pruža jednu ili više usluga drugim modulima. Modul upotrebljava usluge koje mu pružaju drugi moduli. Modul se obično ne smatra samostalnim sustavom. Moduli se obično sastoje od više jednostavnijih komponenti sustava.

Postoje dvije glavne strategije koje se mogu upotrebljavati kod rastavljanja podsustava na module:

Objektno-orijentirana dekompozicija – Sustav se rastavlja na skup objekata koji međusobno komuniciraju. Moduli su ovdje objekti sa privatnim stanjem i definiranim operacijama za to stanje. Ti objekti su slabo spareni (loosely coupled) jedan s drugim i imaju dobro definirana sučelja. Objekti 'pozivaju' usluge koje pružaju drugi objekti. Kod OO dekompozicije važne su klase objekata, njihovi atributi i funkcije. Objekti se stvaraju iz klasa i mora postojati neki kontrolni model koji bi koordinirao rad objekata.

Prednosti: budući da su objekti slabo spareni, implementacija objekata se može mijenjati bez da to utječe na druge objekte. Objekti su često reprezentacije stvarnih entiteta, tako da je struktura sustava čitljiva i razumljiva. Budući da se ovi real-world entiteti upotrebljavaju u različitim sustavima, objekte se može ponovo upotrijebiti. OO programski jezici se danas često koriste.

Mane: Da bi objekti mogli koristiti određene usluge, moraju eksplicitno navesti ime i sučelje drugih objekata. Ako se promijeni sučelje, mora se provjeriti kako ta promjena utječe na sve korisnike promijenjenog objekta. Kompleksnije entitete može biti teško predstaviti objektima.

Funkcijski-orijentiran cjevovod ili model toka podataka (Function-oriented pipelining ili data-flow model) – Sustav se rastavlja u funkcionalne module koji primaju ulazne podatke i tranformiraju ih u izlazne. Moduli su ovdje funkcionalne transformacije.

Prednosti: Podržava ponovnu upotrebu transformacija. Dobar je jer većina ljudi o svom poslu razmišlja u terminima inputa i outputa. Jednostavan je za implementaciju bilo kao usporedan (paralelan) ili kao sekvencijalni sustav, lako je dodavati nove transformacije.

Mane: Mora postojati zajednički format za prijenos podataka kojeg mogu prepoznati sve transformacije. Teško je ostvariti interaktivne sustave korištenjem ovog modela.

Kontrolni stilovi

Da bi podsustavi radili kao cjeloviti sustav, mora ih se kontrolirati tako da njihove usluge stignu na pravo mjesto u pravo vrijeme.

Postoje dva generička kontrolna stila koja se upotrebljavaju u softverskim sustavima:

Centralizirana kontrola – Jedan podsustav ima potpunu odgovornost za kontrolu, pokretanje i zaustavljanje ostalih podsustava, dakle, dizajniran je kao sistemski kontroler. Može predati kontrolu drugom podsustavu, ali će očekivati da će mu je taj podsustav vratiti.

Modeli centralizirane kontrole spadaju u dvije klase, ovisno o tome da li se kontrolirani podsustavi izvode sekvencijalno ili paralelno.

1. The call-return model – Upotrebljiv je samo za sekvencijalne sustave. Kontrola počinje na vrhu subroutine hijerarhije i kroz pozive subroutina se kreće prema dnu. Prednosti: jednostavno je analizirati kontolni tok te odrediti kako će sustav reagirati na određeni ulaz.
2. The manager model – Upotrebljava se za usporedne sustave. Jedna komponenta sustava je dizajnirana kao sistem menadžer i kontrolira pokretanje, zaustavljanje i koordinaciju ostalih sistemskih procesa. Jedna varijanta ovog modela može se primijeniti na sekvencijalne sustave.

Event-based (event-driven) control – Svaki podsustav odgovara na događaje generirane iz vana. Ove događaje mogu generirati drugi podsustavi ili mogu biti generirani iz okoline sustava.

Koriste se dva ovakva modela:

1. Broadcast models – Kod ovih modela, događaj (event) je broadcast namijenjen svim podsustavima. Svaki podsustav koji je isprogramiran da može obraditi neki događaj može i odgovoriti na njega. Ovi modeli su efikasni kod integracije podsustava koji su distribuirani na različitim računalima na mreži. Kod ovog modela, za svaki podsustav su vezani određeni događaji i kad se određeni događaj desi, kontrola se predaje podsustavu koji ga može obraditi. Podsustavi odlučuju koji su im događaji potrebni, a event i message handler osiguravaju da ih i dobiju. Svi se događaji mogu poslati svim podsustavima (broadcast), ali to najčešće nije tako. Event i message handler održavaju registar u kojemu se nalaze informacije o tome koji su događaji važni kojim podsustavima. Kada se desi neki događaj, event handler provjeri taj registar, i taj događaj proslijedi podsustavu kojemu je on važan. Prednosti: evolucija je relativno jednostavna. Novi podsustav koji bi trebao obraditi određenu klasu događaja može se jednostavno integrirati tako da se događaji koji su za njega važni jednostavno prijave event handleru. Bilo koji podsustav može aktivirati bilo koji drugi podsustav bez da zna njegovo ime ili lokaciju. Podsustavi se mogu implementirati na distribuiranim uređajima. Mane: podsustavi ne znaju kada će se (ili hoće li se) određeni događaji obraditi. Kad podsustav generira neki događaj, on ne zna kojemu je drugom podsustavu taj događaj važan. Ovo može stvoriti konflikte kada se rezultati obrade tog događaja objave.

2. Interrupt-driven models – Ekskluzivno se upotrebljavaju kod real-time sustava kod kojih se eksterno generirani događaji moraju obraditi jako brzo ili čak trenutno. Kod ovih modela definirani su različiti tipovi interrupta koji bi se mogli dogoditi, te handle za svaki od njih. Svaki tip interrupta je povezan sa memorijskom lokacijom na kojoj se nalazi adresa njegovog handlera. Kada se neki interrupt desi, odmah se predaje kotrola njegovom handleru. Prednosti: omogućava jako brz odgovor na interrupt. Mane: težak je za isprogramirati i validirati.

Referentne arhitekture

Modeli arhitekture mogu biti specifični za neku aplikacijsku domenu.

Dva tipa domain-specific modela:

1. Generalni modeli – apstrakcije većeg broja realnih sustava i obuhvaćaju njihove glavne karakteristike.
2. Referentni modeli – apstraktniji su od generalnih modela i opisuju veću klasu sustava. Oni informiraju dizajnere o generalnoj strukturi određene klase sustava. Predstavljaju idealiziranu arhitekturu koja uključuje sve funkcije koje sustav može imati. Primjeri: OSI referentna arhitektura, CASE referentni model.

12. ARHITEKTURE DISTRIBUIRANIH SUSTAVA

Gotovo svi veliki kompjuterski sustavi su danas distribuirani sustavi. Distribuirani sustav je sustav kod kojeg je obrada informacija distribuirana na više računala.

Ovi sustavi imaju niz prednosti nad centraliziranim sustavima:

1. Dijeljenje resursa – Kod distribuiranih sustava omogućeno je dijeljenje hardverskih i softverskih resursa, kao diskova, printera, fajlova ili kompajlera.
2. Otvorenost – Distribuirani sustavi su obično otvoreni sustavi, što znači da su dizajnirani na temelju standardnih protokola i omogućavaju uspješnu integraciju opreme i softvera koji su proizvedeni od strane različitih proizvođača.
3. Paralelnost (engl. concurrency, usporednost) – Kod distribuiranih sustava, više se procesa može odvijati istovremeno na različitim računalima na mreži. Ovi procesi mogu, ali ne moraju, komunicirati jedan s drugim tijekom svog normalnog rada.
4. Skalabilnost (engl. scalability) – Mogućnosti sustava mogu se poboljšati dodavanjem novih resursa koji bi se nosili sa novim zahtjevima za taj sustav.
5. Tolerantnost na greške - Kod većine distribuiranih sustava, kad dođe do neke greške, sustav još uvijek može nastaviti sa radom (iako sa slabijim performansama). Sustav će skroz prestati sa radom tek kada dođe do pada čitave mreže.

Mane distribuiranih sustava u odnosu na centralizirane sustave:

1. Kompleksnost – Distribuirani sustavi su kompleksniji od centraliziranih, i zato je teže razumjeti njihova 'izviruća' (engl. emergent) svojstva. Performanse sustava kod centraliziranih sustava ovise samo o brzini jednog procesora, a kod distribuiranih sustava ovise o bandwithu mreže i brzini procesora na njoj.
2. Sigurnost – Sustavu se pristupa sa različitih računala, a podaci koji se prenose putem mreže mogu se prisluškivati. Dakle, teže je osigurati i održavati integritet podataka.
3. Upravljanje – Računala unutar sustava mogu biti drugačijih tipova i mogu imati instalirane različite operacijske sustave. Pogreške se s jednog računala mogu propagirati na ostala računala s nepredvidljivim posljedicama. Dakle, potreban je veći trud da bi se sustav održavao i da bi se njime upravljalo
4. Nepredvidljivost – Kao što svi korisnici WWW-a znaju, distribuirani sustavi mogu biti nepredvidljivi u svojim odgovorima. Ovisno o opterećenju sustava i mreže, nekad ćemo odgovor na zahtjev dobiti brže, a nekada sporije.

Dva glavna tipa arhitektura distribuiranih sustava:

1. Klijent-server arhitekture – Na sustav možemo gledati kao na skup usluga koje se nude klijentima koji te usluge koriste. Ovdje razlikujemo klijente i servere.
2. Arhitekture distribuiranih objekata – Ne postoji razlika između klijenata i servera. Na sustav se može gledati kao na skup objekata koji međusobno komuniciraju, a čija je lokacija nebitna.

Ova dva tipa su intra-organizacijski, dakle aplikacije su najčešće distribuirane unutar jedne organizacije.

Inter-organizacijski tipovi su peer-to-peer arhitektura (p2p) i service-oriented arhitektura.

Komponente kod distribuiranih sustava mogu biti implementirane u različitim programskim jezicima i mogu se izvršavati na potpuno različitim tipovima procesora. Modeli podataka, reprezentacija informacija i komunikacijski protokoli mogu svi biti različiti. Dakle, potreban nam je neki softver koji će osigurati komunikaciju i izmjenu podataka između svih tih različitih dijelova. Takav softver naziva se *middleware* – nalazi se između sustava i različitih distribuiranih komponenti. Middleware je obično off-the-shelf softver, dakle nije posebno razvijen za sustav na kojem se upotrebljava.

Distribuirani sustavi se obično razvijaju koristeći objektno-orijentirani pristup. Ovi su sustavi napravljeni od slapo integriranih, neovisnih dijelova, od kojih svaki može komunicirati direktno sa korisnikom ili sa bilo kojim drugim dijelom sustava.

Multiprocesorska arhitektura

Najjednostavniji model distribuiranog sustava je multiprocesorski sustav gdje se softverski sustav sastoji od više procesa koji se mogu, ali ne moraju, izvoditi na različitim procesorima. Ovaj je model čest kod velikih real-time sustava. Distribucija procesa procesorima može biti predeterminirana, a može se i izvoditi pod kontrolom nekakvog kontrolera (dispatcher) koji odlučuje koji će proces dodijeliti kojem procesoru.

Intra-organizacijske arhitekture

Klijent-server arhitekture - Na sustav možemo gledati kao na skup usluga koje serveri nude klijentima koji te usluge koriste. Klijenti moraju biti svjesni servera koji su im dostupni, ali obično ne znaju za postojanje drugih klijenata. Kod ovih arhitektura razlikujemo klijenta i servera. Aplikaciju koju razvijamo pomoću ove arhitekture možemo predočiti pomoću tri sloja:

1. Prezentacijski sloj – zadužen za prezentiranje informacija korisniku i za interakciju s korisnikom.
2. Application processing sloj – zadužen za implementiranje logike samog sustava.
3. Data management sloj – zadužen za rad sa bazom podataka.

Najjednostavnija klijent-server arhitektura naziva se two-tier klijent-server arhitektura. Aplikacija je ovdje organizirana kao server (ili više identičnih servera) i kao skup klijenata. Two-tier klijent-server arhitektura dolazi u dva oblika:

1. Thin-client model – Klijent je zadužen samo za prezentacijski softver, a server za data management i application processing. Mana ovog modela je što se stvara veliko opterećenje na serveru, a procesorska snaga na strani klijenta ostaje u velikom dijelu neiskorištena. Također imamo problema sa performansama sustava i skalabilnošću.

2. Fat-client model – Klijent je zadužen za prezentacijski softver i za application processing, a server za data management. Mane ovog modela su problemi kod sistem menadžmenta.

Budući da, bez obzira koji model izaberemo, uvijek ćemo nailaziti na probleme kod two-tier modela, koristi se još i three-tier model, kod kojeg se procesi prezentacije, application processinga i data managementa svi odvijaju na različitim procesorima. Također se još može koristiti i multi-tier model, gdje se sustavu dodaju dodatni serveri.

Arhitekture distribuiranih objekata – Ovaj model je nešto generalniji od klijent-server modela. Ovdje nema razlike između klijenta i servera. Komponente sustava su objekti koji kroz svoje sučelje pružaju određene usluge. Objekti mogu pozivati usluge koje nude drugi objekti, dakle svaki objekt je ujedno i klijent i server, te među njima zato nema razlike. Objekti mogu biti distribuirani na različitim računalima na mreži i komuniciraju putem middlewarea. Middleware se ovdje naziva *object request broker*.

Prednosti: Ovaj model omogućuje dizajnerima da odgode donošenje odluke o tome kako će se pružati usluge sustava. Objekti koji pružaju usluge mogu se izvršavati na bilo kojem čvoru na mreži. Ovo je jako otvorena arhitektura kojoj se novi resursi mogu jednostavno dodavati. Sustav je fleksibilan i skalabilan.

Mane: Glavna mana ovog modela je da je kompleksniji za dizajnirati od klijent-server sustava.

CORBA

CORBA je internacionalni set standarda za Object Request Broker (ORB) – middleware koji omogućava komunikaciju objekata koji su distribuirani na različitim računalima. Middleware koji podržava distribuirane objekte potreban je na dvije razine:

1. Razina logičke komunikacije – middleware omogućava objektima koji se nalaze na različitim računalima da razmjenjuju podatke i kontrolne informacije.
2. Razina komponenti – middleware pruža osnovu za razvoj kompatibilnih komponenti.

CORBA standarde definirani su od strane Object Management Group-a (OMG). Ovi standardi predlažu da bi se distribuirana aplikacija trebala sastojati od sljedećih komponenti:

1. Aplikacijski objekti – koji su dizajnirani i implementirani za tu aplikaciju.
2. Standardni objekti – koji su definirani od strane OMG-a za određenu domenu sustava.
3. Osnovne CORBA usluge – koje omogućavaju osnovne usluge za distribuirane sustave.
4. Horizontalne CORBA funkcije – kao što su funkcije za korisničko sučelje i sistem menadžment.

CORBA standardi:

1. Model objekta za aplikacijske objekte.

2. Object request broker koji upravlja zahtjevima za uslugama koje pružaju objekti.
3. Skup generalnih usluga koje pružaju objekti i koji su korisni za više distribuiranih sustava.
4. Skup osnovnih komponenti koje se grade iznad svih ovih usluga.

CORBA objekti su slični onima u C++. Objekti napisani u različitim programskim jezicima mogu međusobno komunicirati.

ORB (Object Request Broker) omogućava komunikaciju među objektima. Zna za sve objekte u sustavu i njihova sučelja. Sučelja se opisuju koristeći IDL (Interface Description Language).

CORBA usluge:

1. Naming and trading services – omogućavaju objektima da otkriju nove objekte na mreži i da saznaju specifikaciju drugih objekata.
2. Notification services – koriste se da bi objekti obavijestili druge objekte da se neki događaj desio.
3. Transaction services – omogućavaju oporavak od greške koja se dogodila tijekom nekog updatea. Ako update ne uspije kod nekog objekta, stanje tog objekta se može vratiti na ono stanje koje je imao prije updatea.

Inter-organizacijske arhitekture

Peer-to-peer arhitekture (p2p) – Peer-to-peer sustavi su decentralizirani sustavi gdje se obrada može odvijati na bilo kojem čvoru na mreži i, barem u principu, nema razlike između klijenta i servera. Sustav je dizajniran na način da iskoristi snagu za obradu i pohranjivanje koja je dostupna od strane velikog broja računala koji se nalaze na mreži. Standardi i protokoli koji omogućavaju komunikaciju između različitih čvorova na mreži su ugrađeni u samu aplikaciju, i svaki čvor mora imati kopiju te aplikacije.

Peer-to-peer modeli:

1. Logical network architecture – distribuirana arhitektura sustava. Dijeli se na decentralizirane i polu-centralizirane arhitekture.
2. Application architecture – generička organizacija komponenti od kojih se sastoji p2p aplikacija.

Kod decentralizirane arhitekture, čvorovi na mreži nisu samo funkcionalni elementi već i komunikacijski switchevi koji mogu usmjeravati podatke i kontrolne signale od jednog čvora prema drugom. Prednosti: jako redundantna arhitektura. Ima tolerantnost na pogreške i na čvorove koji se diskonektiraju sa mreže.

Kod polu-centralizirane arhitekture jedan ili više čvorova se ponašaju kao serveri da bi olakšali komunikaciju među čvorovima.

Peer-to-peer arhitektura je puno efikasnija kod inter-organizacijskih sustava od service-oriented arhitekture.

Service-oriented arhitektura – Ovi sustavi nastaju povezivanjem softverskih usluga koje pružaju različiti pružatelji usluga. Razvojem WWW-a, klijentska računala su mogla

pristupati serverima izvan vlastite organizacije. Da bi se ovaj proces olakšao, razvili su se web servisi. Organizacije koje su željele određene informacije učiniti dostupnima za različite programe, to su mogle napraviti definiranjem web servis sučelja, preko kojeg se je pristupalo tim informacijama i preko kojeg se je određivao način kako im se može pristupiti.

Sama bit usluge je da je ona neovisna o aplikaciji koja ju pruža.

Usluge su bazirane na XML baziranim standardima, mogu se implementirati na bilo kojoj platformi i pisati u bilo kojem programskom jeziku. Glavni standardi koji omogućavaju komunikaciju između web servisa su:

1. SOAP (Simple Object Access Protocol) – definira kako organizirati izmjenu podataka između web servisa.
2. WSDL (Web Services Description Language) – definira na koje se načine sučelja web servisa mogu prezentirati.
3. UDDI (Universal Description, Discovery and Integration) – definira način na koji se mogu organizirati informacije o uslugama. Te informacije koriste klijenti da bi pronašli potrebne usluge.

Razlike između ovog modela i modela distribuiranih objekata: usluge mogu pružati bilo koji pružatelji usluga, ovaj je model neovisan o providerima. Informacije o uslugama su javno dostupne. Implementacija novih usuga je lagana. Korisnici usluga plaćaju za te usluge samo kada ih koriste (npr. umjesto da se kupi neka skupa komponenta koja se rijetko koristi, možemo je iznajmljivati). Aplikacije su manje (korisno kod embedded sustava) i adaptivne.

13 Application architectures

Aplikacijski sustavi su namjenjeni potrebama raznih organizacija ili poslova, no svi poslovi ili organizacije često imaju puno toga zajedničkog.
Sljedeće generičke aplikacijske arhitekture se mogu koristiti na razne načine:

1. *Kao početna točka kod procesa dizajniranja arhitekture*
2. *Kao dizajn „checklist“*
3. *Kao uputu za organizaciju poslova unutar razvojnog tima*
4. *Za otkrivanje postojećih komponenti koje se mogu iskoristiti*
5. *Kao riječnik pri diskusiji o sličnim aplikacijama koristeći se usporedbama*

Razni tipovi aplikacija obično imaju puno toga zajedničkog pa ih svodima na 4 najproširenija tipa aplikacijske strukture.

1. *Data-processing applications* Data-procesing aplikacije su aplikacije koje se vode podacima koji se obrađuju u obrocima (komadima) bez izravne intervencije

korisnika. Određene akcije se poduzimaju ovisno o podacima koji se procesuiraju.

2. *Transaction-processing aplications* Transaction-processing aplikacije su database centrilizirane aplikacije koje procesuiraju zahtjeve korisnika cime se osvježavaju informacije unutar baze podataka.
3. *Event-processing systems* Ovo je velika klasa aplikacija čije djelovanje sustava ovisi o interpretaciji događaja (eventa) u radnom okolišu sustava. Ti događaji mogu biti unosi komandi korisnika ili promjena varijabli koje nadzire taj sustav.
4. *Language-processing systems* Language-processing sustavi su sustavi kod kojih su korisnikove namjere izrazene fromalnim jezikom (kao npr. Java). Language-processing sustav procesuira taj jezik u neki interni format a zatim interpretira tu internu reprezentaciju. Najpoznatiji takvi sustavi su kompajleri.

Data-processing systems

Ovi se sustavi usredotočuju na podatke a baze podataka na koje se oslanjaju su obično naredbe za veličinu veće od samog sustava. Data-processing sustavi su batch-processing sustavi (procesuiraju podatke u komadima) gdje se podaci unose i iznose u obliku komada (batches) iz raznih dokumenata ili baza podataka radije nego da se unose preko, a zatim iznose na te iste korisničke terminale. Ti sustavi selektiraju podatke iz nekog unosnog zapisa a zatim na osnovu vrijednosti raznih polja tog podatka poduzimaju radnje specificirane programom.
Priroda kod data-processing sustava gdje se zapisi ili transakcije procesuiraju serijski nalaže da su ti sustavi prije funkcionalno nego objektno orjentirani. Njihova sama arhitektura je relativno jednostavna no kompleksnost ovakvih sustava se nalazi u algoritmima procesuiranja podataka.

Transaction-processing systems

Transaction-processing sustavi su dizajnirani da procesuiraju korisničke zahtjeve za dohvatom informacija iz ili osvježavanjem informacija neke baze podataka. Tehnički, transakcije baze podataka su sljed operacija koje se tretiraju kao cjelina. Sve operacije unutar transakciji moraju biti dovršene prije nego promjene u bazi podataka postanu premanentne što znaci da pogreška unutar jedne operacije ne dovodi do ne konzistentnosi baze podataka.
Transaction-processing sustavi su obično interaktivni sustavi gdje korisnik postavlja asikrone zahtjeve na neku uslugu kroz neke I/O procesuirajuće komponente kod kojih se zahtjevi porcesuiraju određenom logikom. Zatim se kreira transakcija i prosljeđuje se do transakcijskog menadžera koji osigurava da se transakcija uredno izvrši i na kraju signalizira aplikaciji da je procesuiranje završeno.

U ovakvim sustavima pristup bazi podataka se obično vrši na razne načine (preko ATM-a, terminala itd.) te oni obično sadrže i neku vrstu među sklopovlja koje odgovara na svaku vrstu pristupa.

Information and resource management system

Svi sustavi koji uključuju interakciju sa bazom podataka koju svi dijele mogu se razmatrati kao transactio-based information sustavi. Informacijski sustav pruža kontrolirani pristup velikoj bazi podataka ko što su to knjižnički katalozi, plan letova itd. E-commerce sustavi su Internet-based resource management sustavi koji su dizajnirani da prihvaćaju elektroničke narudžbe artikala ili usluga a onda organiziraju dostavu tih artikala ili usluga do kupca.

Event-processing systems

Event-processing sustavi reagiraju na događaje unutar sustavnog radnog okoliša ili na interakciju korisničkog sučelja. Ključna karakteristika event-processing sustava je da su trenuci pojavljivanja događaja nepredvidljivi i da sustav mora bit u mogućnosti reagirat na te događaje u vremenu kako se oni pojavljuju.
Sustav detektira i interpretira događaje (evente). Događaji prouzročeni korisničkim sučeljem reprezentiraju implicitne naredbe usmjerene na sustav.
Real-time sustavi, koji donose odluke u realnom vremenu u odnosu na vanjske podražaje, su također event-based processing sustavi.
Editing systems (editirajući sustavi) su programi koji se vrte na PC-u ili radnoj stanici i omogućavaju korisniku da editira dokumente kao što su tekst dokumenti, dijagrami ili slike.
Editirajući sustavi imaju niz raznih karakteristika koje ih razlikuju od drugih sustava što utječe na njihov arhitekturski dizajn:

1. Editirajući sustavi su većinom singler-user sustavi pa se ne moraju baviti problemima višestrukog konkurentnog pristupa podacima i time imaju jednostavniji management podataka nego transaction-base sustavi.
2. Editirajući sustavi moraju pružiti brzu reakciju na korisnikove akcije poput „select“ i „delete“ što znači da moraju raditi nad podacima koji će se radije držati u radnoj memoriji nego na disku. To znači da ti podaci mogu biti izgubljeni ako dođe do sistemske pogreške pa bi ovakvi sustavi trebali imati neku vrstu oporavka od pogreške.
3. Editirajuće sesije su obično duže nego recimo narudžbe artikala pa je time i rizik gubitka podataka veći pa stoga mnogi editirajući sustavi sadrže uslugu oporavka koja automatski sprema podatke i vraća ih korisniku u slučaju sistemske pogreške.

Language-processing systems

Language-processing sustavi na ulazu prihvaćaju obični jezik ili neki umjetni jezik a kao izlaz generiraju reprezentaciju tog jezika. U softverskom inžinjeringu najpoznatiji takvi sustavi su kompajleri koji prevode umjetne jezike višeg nivoa u strojni kod.
Instrukcije opisuju što se treba uraditi i prevode se pomoću translatora u neku vrstu internog formata. Te instrukcije se onda interpretiraju od strane druge komponente sustava koja dobavlja instrukcije i izvršava ih koristeći (ako je to potrebno) podatke iz danog prostora. Izlaz procesa je rezultat interpretacije instrukcija nad danim podacima.
Meta-CASE alati su program generatori koji se koriste za kreiranje specifičnih CASE alata za pružanje podrške softver inžinjering metodama. Meta-CASE alati uključuju opis komponenata metode (njenih pravila itd.) napisane u jeziku za posebne svrhe koji je parsiran i analiziran kako bi konfigurirao generirani CASE alat.
Translatori u language-processing sustavima imaju generičku arhitekturu koja uključuje sljedeće komponente:

1. Leksički analizator, koji uzima ulazne nakove jezika i konvertira ih u neki interni format.
2. Tablicu simbola, koja sadrži informacije o imenima entiteta koje se koriste u tekstu kojeg se prevodi.
3. Analizator sintakse, koji provjerava sintaksu jezika koji se prevodi.
4. Sintaksno stablo, koje je interna struktura koja reprezentira program koji se kompajlira.
5. Analizator semantike, koji se koristi iformacijama iz sintaksnog stabla i tablice simbola kako bi provjerio semantičku ispravnost unesenog teksta.
6. Kod generator, koji prolazi kroz sintaksno stablo i generira abstraktni strojni kod.

16 User interface design

Pomno odabrano korisničko sučelje ima ključnu ulogu u postizanju punog potencijala određenog softvera. Pri tome treba paziti da dizajn bude prikladan vještinama, iskustvu i očekivanjima ciljane grupe korisnika. Mnoge takozvane „korisničke pogreške“ su izazvane lošim dizajnom sučelja koje ne uzima u obzir sposobnosti i razumjevanje korisnika koji se njime koriste.
Kad se donose odluke o dizajnu sučelja treba uzeti u obzir fizičke i mentalne sposobnosti ciljane grupe ljudi. Jedni od važnih faktora koje treba uračunati su:

1. Ljudi imaju ograničeno kratkoročno pamćenje – čovjek istovremeno može zapamtiti oko 7 informacija, pa stoga ako se korisnika obaspe sa puno informacija, on neće biti u mogućnosti sve zapamtiti.
2. Svi radimo pogreške, pogotovo ako se mora baratat sa puno informacija ili se čovjek nalazi pod stresom. Kada se u sustavu desi pogreška i počnu izlaziti poruke

upozorenja i oglašavanje alarma, to samo dodatno stvara stres na korisnika i povećava vjerojatnost da će korisnik donijeti krivu odluku.

3. Postoji veliki spektar različitih fizičkih sposobnosti. Neki ljudi vide i čuju bolje od drugih, neki su daltonisti, a neki imaju bolju koordinaciju. Ne treba dizajnirati sučelje prema vlastitim sposobnostima niti pretpostavljati da će se u takvom sučelju korisnici uspjeti snaći.
4. Ljudi preferiraju drugačije vrste interakcije. Netko više voli raditi sa sličicama a netko sa tekstom. Direktno upravljanje je nekome prirodnije ali drugi možda vise vole stil interakcije koji se bazira na unos komandi u sustav.

Ovi ljudski faktori čine princip po kojem se dizajnira sučelje i primjenjiv je na sva korisnička sučelja pa se treba koristiti kao smjernice pri dizajniranju sučelja.
Princip ***user familiarity*** sugestira da se ne treba korisnika prisiljavati na prilagođavanje sučelju samo zato što ga je na taj načinm lakše implementirati. Sučelje treba koristiti termine koji su korisniku poznati i objekti kojima sustav manipulira trebaju biti direktno povezani sa radnim okolišem korisnika.
Princip ***user interface consistency*** znači da sistemske komande i meniji trebaju imati isti format, parametri se trebaju prenositi u svim komandama na isti način a stavljanje znakova interpunkcije bi trebala biti slična. Konzistentno sučelje reducira vrijeme prilagodbe korisnika jer znanje stečeno nad jednom komandom je primjenjivo na sve ostale ili čak na srodne aplikacije.
Princip ***minimal surprise*** je prikladan zato što ljude iritira kada se sustav ponaša na način na koji ne žele. Kako se sustav koristi korisnici si stvaraju mentalni model o tome kako sustav funkcionira. Ako određena akcija u nekom kontekstu ima određeno djelovanje onda se očekuje da ista akcija ima slično djelovanje i u drugom kontekstu. Ako se desi nešto potpuno drugo korisnika se zbunjuje pa dizajneri trebaju paziti da slične akcije imaju slično djelovanje na sustav.
Princip ***recoverability*** je važan jer korisnici nehotice rade greške pri korištenju sustava. Sam dizajn može smanjiti učestalost pogreški ali ih ne može nikako potpuno isključiti pa bi trebalo uključiti dodatnu uslugu sučelja koja pruža oporavak nakon takvih grešaka. To se postiže na jedan od tri načina:

1. *Potvrdom destruktivnih akcija* Ako korisnik poduzima neku akciju koja je potencijalno destruktivna, sustav bi trebao pitati korisnika da li je to stvarno ono što želi napraviti, prije nego se uništi informacija.
2. *Pružanjem undo usluge* Undo vraća sustav u stanje koje je prethodilo određenoj akciji. Višestruke razine undo usluge su praktične jer korisnici često ne primjete odmah da su napravili pogrešku.
3. *Checkpointing* Checkpointing uključuje spremanje stanja sustava u periodičkim intervalima i omogućava sustavu da restartira od zadnjeg checkpointa. Tako da kad se desi pogreška, korisnici mogu nastaviti svoj rad iz prethodnog stanja.

Srodni princip je ***user assistance*** princip. Sučelja bi trebala imati ugrađenu korisničku podršku koja će pružati pomoć na raznim razinama. Od osnova kako započeti do detaljnih opisa sustavnih usluga. *Help system* bi trebao biti struktuiran tako da ne pretrpa korisnika sa informacijama kada zatraže pomoć.
Princip ***user diversity*** prepoznaje da za mnoge interaktivne sustave postoje različiti tipovi korisnika. Neki će biti povremeni korsinici dok će drugi biti power user-i koji će se sustavom koristiti i do par sati dnevno. Povremeni korisnici trebaju sučelje koje im pruža podršku dok intezivni korisnici žele imati kratice preko kojih mogu interaktirati što brže. Pa tako bi bilo korisno uključiti uslugu koja može povećavat font teksta, zamjeniti zvuk sa tekstom, prikazivati jako velika dugmad itd.
Svi ovi principi mogu međusobno biti kontradiktorni pa treba dobro razmisliti koja je ciljana grupa korisnika te izabrati princip koji više odgovara takvoj grupi korisnika.

Design issues

Dizajner korisničko sučelja je uvijek suočen sa dva pitanja:

1. Kako bi korisnik trebao interaktirati sa računalnim sustavom.
2. Kako bi informacije sa računalnog sustava trebale biti prikazane korisniku.

Skladno korisničko sučelje mora integrirati korisničku interakciju i prezentaciju informacija. To može biti teško ostvarivo jer dizajner mora naći kompromis između najprikladnijeg stila interakcije i prezentacije, osnovnog iskustva korisnika i opreme koja je dostupna.

User interaction

Korisnička interakcija znači unošenje komandi i sukladnih podataka u računalni sustav. Na ranim računalima to se radilo u obliku komandnih linija a današnja računala imaju uznapredovala sučelja kojima se je lakše koristiti. Schneiderman je klasificirao ove forme interakcije u pet primarnih stilova:

1. *Direct manipulation* Korisniki interaktira izravno sa objektima na ekranu. Direktna manipulacija obično uključuje pointing device (pokazivačku napravu: miš, trackball,touchscreen itd.)koji indicira object koji se manipulira i akcije koje ce se nad njim primjeniti.
2. *Menu slection* Korisnik selektira komandu sa liste mogućnosti (menija).
3. *Form fill-in* korisnik popunjava polja unutar forme.
4. *Command language* Korisnik unosi specijalne komande i odgovarajuće parametre da bi dao sustavu instruklcije.
5. *Natural language* Korisnik unosi komande u prirodnom jeziku. To se obično postiže tako da se uneseni jezik parsira i prevede u neku sistemsku komandu.

Information presentation

Predodžba informacija može biti izravna i prezentirati podatke u izvornom obliku ili pak grafički pomoću raznih grafova. Dobra smjernica pri dizajnu prezentacije informacije je da se softver potreban za prezentiranje drži odvojeno od same informacije kako promjene na samoj prezentaciji nebi iziskivale promjene i samog formata podataka.
Da bi napravili najprikladniju prezentaciju potrebno je poznavanje pozadine korisnika koji koriste sustav. Pri tome treba imati sljedeća pitanja na umu:

1. Da li korisnika interesiraju precizni podaci ili usporedba vrijednosti podataka.
2. Koliko se često podaci mijenjaju i da li male promjene trebaju odmah alarmirati korisnika.
3. Da li korisnik treba poduzeti kakve akcije u skladu sa promjenom podataka.
4. Da li korisnik ima potrebu za interakcijom nad prikazanim podacima kroz sučelje direktne manipulacije.
5. Da li su podaci koji se trebaju prikazati tekstualni ili numerički? Da li su relativne vrijednosti informacije bitne?

Ne treba pretpostaviti da korištenje grafike čini predodžbu interesantnijom, ona zauzima dosta prostora i može usporiti download stranica ako čovjek radi na sporim dial-up vezi. Informacije koje se ne mijenjaju tijekom sesije mogu se prikazati pomoću grafova ili teksta ovisno o informaciji. Tekstualni prikaz zauzima manje prostora ali je manje pregledan na prvi pogled. Treba odvojiti podatke koji se dinamički mijenjaju od onih koji se ne mijenjaju dinamički koristeći različite stilove prezentaciji. Ako se podaci polako mijenjaju i ako je potreba za preciznošću veća,treba se koristiti tekstualni prikaz a ako se podaci učestalo mijenjaju i ako je važnija usporedba podataka tada je bolje koristiti grafički prikaz.
Prikladno sa prezentacijom informacija treba dobro razmisliti koje boje koristiti za tu prezentaciju. Boje mogu poboljšati prezentaciju ali loše odabrane boje mogu učiniti sučelje ružnim. Schneiderman daje 14 ključnih smjernica za odabir boja za sučelje a najvažnije od njih su:

1. *Ograniči broj boja koje se primjenjuju i budi konzervativan u njihovoj primejni* Ne treba korsititi više od 4-5 različitih boja unutar jendog prozora i ne više od 7 u cijelom sustavno sučelju.
2. *Koristi se promjenom boje za indiciranje promjene sustavnog status* Ako sa na ekranu promjeni boja to treba indicirati da se je desio neki važan događaj unutar sustava.
3. *Koristi boje za davanje podške korisniku koji pokušava obaviti neki zadatak* Ako treba otrkiti neke anomalije zgodno je da one budu osvjetljene ili ako treba pronaći sličnosti unutar nekog teksta i slično također ih je zgodno osvjetliti.
4. *Koristi kodiranje bojama na smislen i konzistentan način* Ako u jednom dijelu sustava prikazuješ poruke upozorenja sa crvenom bojom, onda tako treba biti i u svim drugim djelovima sustava a crvenu boju nebi trebalo koristiti ni za što drugo jer bi mogla biti krivo interpretirana.

5. *Budi oprezan pri kombiniranju boja* Zbog fiziologije oka ljudi se ne mogu fokusirati na crvenu i plavu boju istovremeno. Druge kombinacije boja bi također mogle biti uznemirujuće.

The UI design process

Dizajn korisničkog sučelja je iterativan proce gdje korisnici interaktiraju sa dizajnerima i prototipovima sučelja kako bi odredili obiljžja, organizaciju i izgled korisničkog sučelja. U dizajnu sučelja postoje tri glavne aktivnosti:

1. *User analysis* U procesu korisničke analize, razvija se razumjevanje o poslovima koji korisnici obavljaju, njihovom radnom okolišu, o ostalim sustavima kojim se koriste, na koji način interaktiraju sa kolegama na svom poslu itd.
2. *System prototyping* Dizajn korisničkog sučelja i njegov razvoj je iterativan posao. Iako korisnici mogu pricačati o tome što očekuju od sučelja, teško je točno odrediti što žele dok im se ne da nešto opipljivo.
3. *Interface evoluation* Iako je očigledno da je potrebna komunikacija sa korsnikom tijekom prototypinga, treba uvijek voditi formalizirani razvoj u kojem će se prikupljati podaci o korisnikovom stvarnom iskustvu sa sučeljem.

User analysis

Kritična aktivnost u razvoju sučelja je analiza korisnika. Ako nije jasno što korisnik želi raditi sa sustavom, onda ne postoje jasna uporišta na osnovu kojih bi se izgradilo dobro korisničko sučelje. Kako bi se steklo razumjevanje o potrebama korisnika treba se koristiti tehnikama analize poslova, etnografijom, intervjuiranjem korisnika, promatranje korisnika, ili svime navedenim.

Analysis techniques

Postoje tri osnovne tehnike analize korisnika: task analysis (analiza posla kojeg obavljaju), intervjuiranje i ispitivanje, i etnografija. Analiza posla i intervjuiranje se odnose na pojedinca dok je etnografija širi pogled na to kako ljudi međusobno interaktiraju te kako si orginaziravaju svoj poslovni okoliš i na koji način rješavaju svoje probleme. Postoje razne forme analize poslova ali najčešće korištena je Hierarchical Task Analysis (HTA) koja je osnovno razvijena da pomogno oko pisanja korisničkih uputa (user manuals) ali se može koristiti da se shvati što korisnici rade da ostvare neki cilj.

User interface prototyping

Cilj prototypinga je da omogući korisniku izravno iskustvo sa sučeljem. Većina nas ima problema razmišljati o sučelju apstraktno a opisat detaljno što želimo. No ipak, kada nam se da primjer, lako nam je prepoznat karakteristike koje nam se sviđaju a koje ne.

U idealnom slučaju kada radimo prototyping sučelja trebali bi napraviti dvije faze procesa:

1. U samom početku treba napraviti prototip na papiru, imat okvirne karakteristike dizajna te ih proć zajedno sa korisnikom.
2. Onda prepraviti dizajn i postupno razvijati sofisticirani automatizirani prototip kojeg treba isporučit korisniku na testiranje i simuliranje aktivnosti.

Može se koristiti i *storyboard* u kojem niz skica ilustriraju sekvence interakcije.
Postoje tri vrste pristupa prototypingu korisničkog sučelja.

1. *Script driven approach* Ako treba jednostavno razvit ideje sa korisnikom, može se koristiti script-driven pristup gdje se kreiraju vizualni elementi kao što su dugmad i meniji na koje se povežu skripte. Na interakciju korisnika skripta prikaže samo sljedeći ekran pokazujući rezultat interakcije. Nema potrebe uključivati ikakvu aplikacijsku logiku.
2. *Visual programing language, k*ao što je visual basic, objedinjuju moćno razvojno okružje. Pružaju pristup raznim predlošcima i sustavu za razvoj korisničkog sučelja koji omogućava brzi razvoj sa svim komponentama i skriptama vezanim uz objekte sučelja.
3. *Internet-based prototyping* Solucije, vezane uz web browsere i jezike kao što je Java, nude već gotova sučelja. Funkcionalnost se dodaje na način da se povže neki segmnet Java koda sa informacijom koju se treba prikazati. Ti segmenti (zovu se apleti (applets)) se izvršavaju automatski kada se stranica učita u browser. Ovakav pristup je jako brza vrsta prototypinga korisničkog sučelja ali postoje restrikcije postavljene od strane browsera i Java sigurnosnog modela.

Interface evaluation

Razvoj sučelja je proces u kojem se ocjenjuje kosinost sučelja i kontrolira se da li su zadovoljene potrebe korisnika pa bi tako ovo trebalo biti dio verifikacije i validacije softver procesa. Sistematska evolucija korisničkog sučelja može biti skup proces koji uključuje znanstvenike i grafičke dizajnere ali postoji i niz jednostavnijih i manje skupih tehnika razvoja koje otkrivaju točne nedostatke korisničkog sučelja:

1. Ankete koje skupljaju informacije o odazivu korisnika na sučelje.
2. Posmatranjem korisnika n a poslu pri interakciji sa sustavom i razmišljanjem na glas o tome kako pokušavaju iskoristiti sustav da obave određeni zadatak.
3. Video snimkom tipičnog korištenja sustavom.
4. Uključivanjem koda softvera koji prati najčešće korištene usluge i najčešće napravljene greške u sustavu.