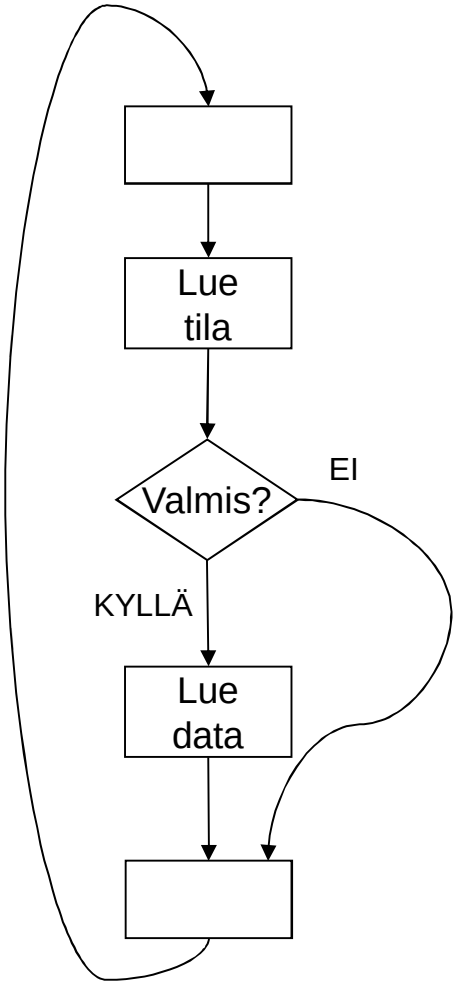
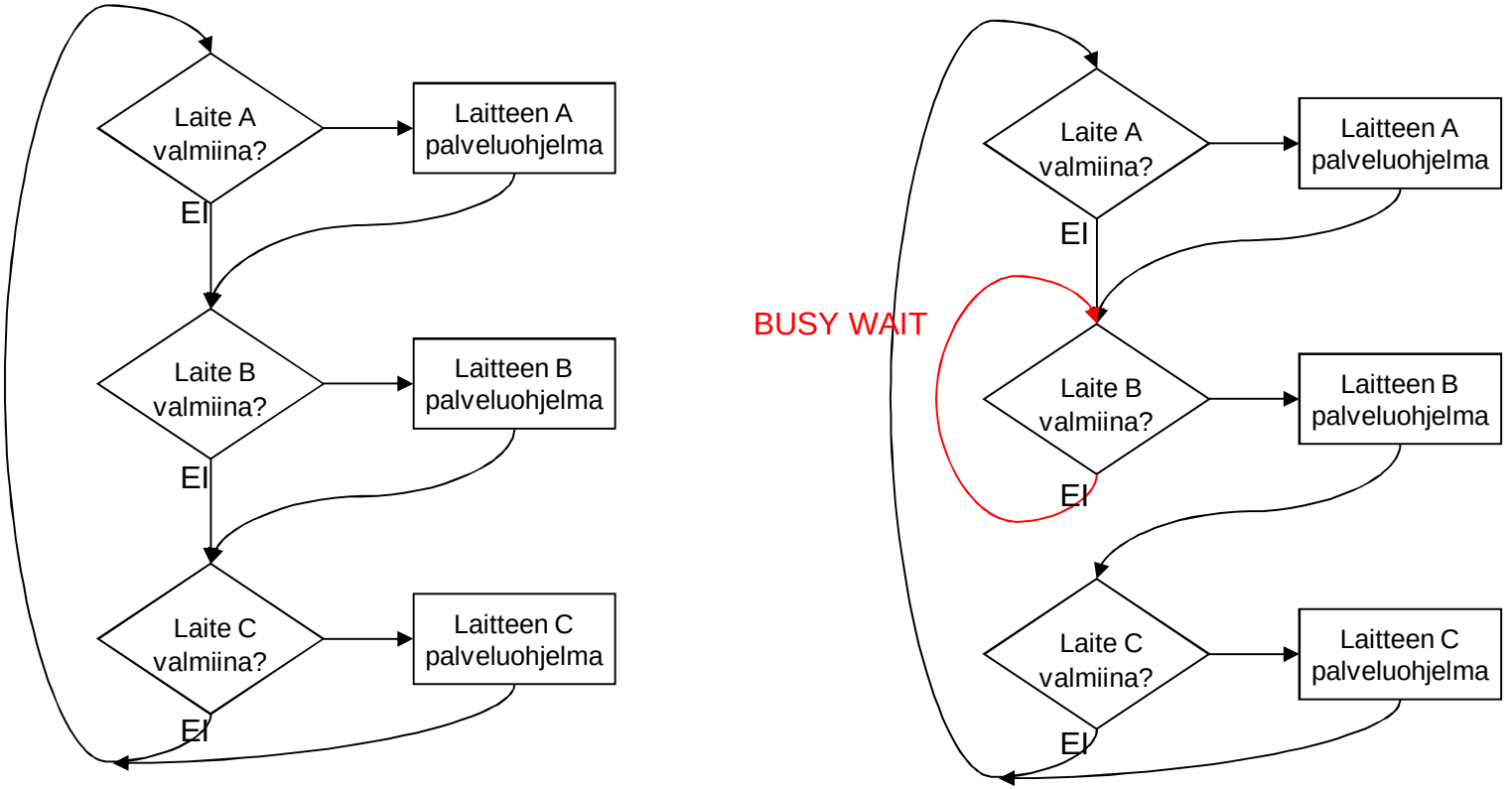
Kiertokysely

- Perinteiset ohjelmointikielet kuten C tukevat hyvin sekventiaalista ohjelmointia, jossa herätteisiin reagointi on helppoa toteuttaa pollauksella eli kiertokyselyllä
- Käyttöjärjestelmien tarjoamat rajapinnat suosivat myös sekventiaalista suoritusta
  - Esim. merkkien lukeminen tapahtuu kutsumalla funktiota, joka palaa sitten kun merkkejä on vastaanotettu
  - Paljon busy-wait tilanteita, jotka pysäyttävät koko ohjelman kierron



Busy wait



Busy wait pysäyttää koko kiertokyselysilmukan!

Foreground/background

- Pelkkää kiertokyselyä ei käytännössä voi käyttää sulautetuissa järjestelmissä, jotta järjestelmän vasteaika olisi jotenkin hallittavissa
- Keskeytysrutiineilla voidaan taata nopea vaste ulkoisiin herätteisiin ja kerätä data esim. puskuriin, josta pääohjelma käy sen hakemassa
  - Keskeytysrutiinit →foreground
    Pääohjelma → background
  - Tässä lähetymistavassa pääohjelman edelleenkin pitää pollata tapahtumia
  - Yhden tapahtuman pollauksen aikana muita tapahtumia ei tyypillisesti pystytä käsittelemään, joten järjestelmän vasteaika ei edelleenkään ole optimaalinen

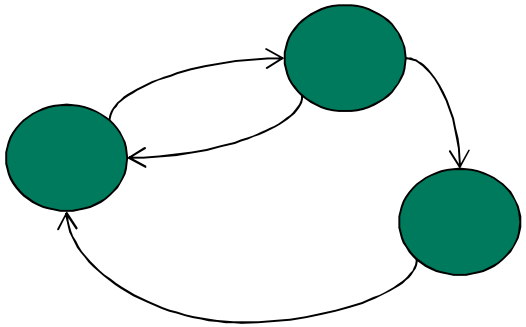
Reaktiivisuus

- Sulautetut järjestelmät ovat yleensä luonteeltaan reaktiivisia
  - Toiminnot tapahtuvat herätteiden pohjalta
- Jos herätteitä on paljon ja järjestelmä monimutkainen, niin perinteinen vuokaavioajattelu ei yleensäkään toimi
  - Busy wait-hidastaa herätteisiin reagointia
  - Pitää muistaa tutkia herätteitä oikeissa kohdissa
  - Tietoliikenneprotokollat erityisen hankalia vuokaavioajattelussa (tuppaavat olemaan monimutkaisia)

- Hyvin monet järjestelmät voidaan mallintaa tilakoneen avulla
  - Järjestelmällä on tila
  - Siirtymät tilojen välillä tapahtuvat herätteiden perusteella
  - Järjestelmän toiminnot liittyvät joko tilaan, jossa ollaan (Mooren kone) tai sekä tilaan että herätteeseen (Mealyn kone)

Tilakoneista

- Perinteisessä tilakoneessa järjestelmän tila selviää yksinomaan tilan perusteella
  - Ohjelmoijan kannalta tilan tallentamiseen riittää yksi muuttuja
- Esimerkiksi tietoliikenneprotokollissa on erilaisia odotusaikoja, joiden toteuttaminen perinteisellä tilakoneella kasvattaisi tilojen määrän valtavan suureksi
  - Esimerkiksi TCP uudelleenlähetysviive (retransmission timeout) vaatisi yhden tilan per odotettava aikayksikkö. Jos aikaa lasketaan 100 ms yksiköissä, niin 5 sekunnin odotus vaatisi 50 erillistä tilaa
  - Tyypillisesti käytetään ns. laajennettuja tilakoneita, joissa tiloilla on omia tilamuuttujia (esim. timeoutin laskentaan)



Laajennettu tilakone

- Esimerkiksi DHCP protokolla on RFC:ssä kuvattu tilakoneena, jossa on ajastimia (T1, T2), jotka käytännössä toteutettaisiin tilamuuttujilla
- Selecting-tilassa toteutuskohtaisesti voi päättää kuinka kauan vastauksia odotetaan (tähän tarvitaan tilamuuttuja)

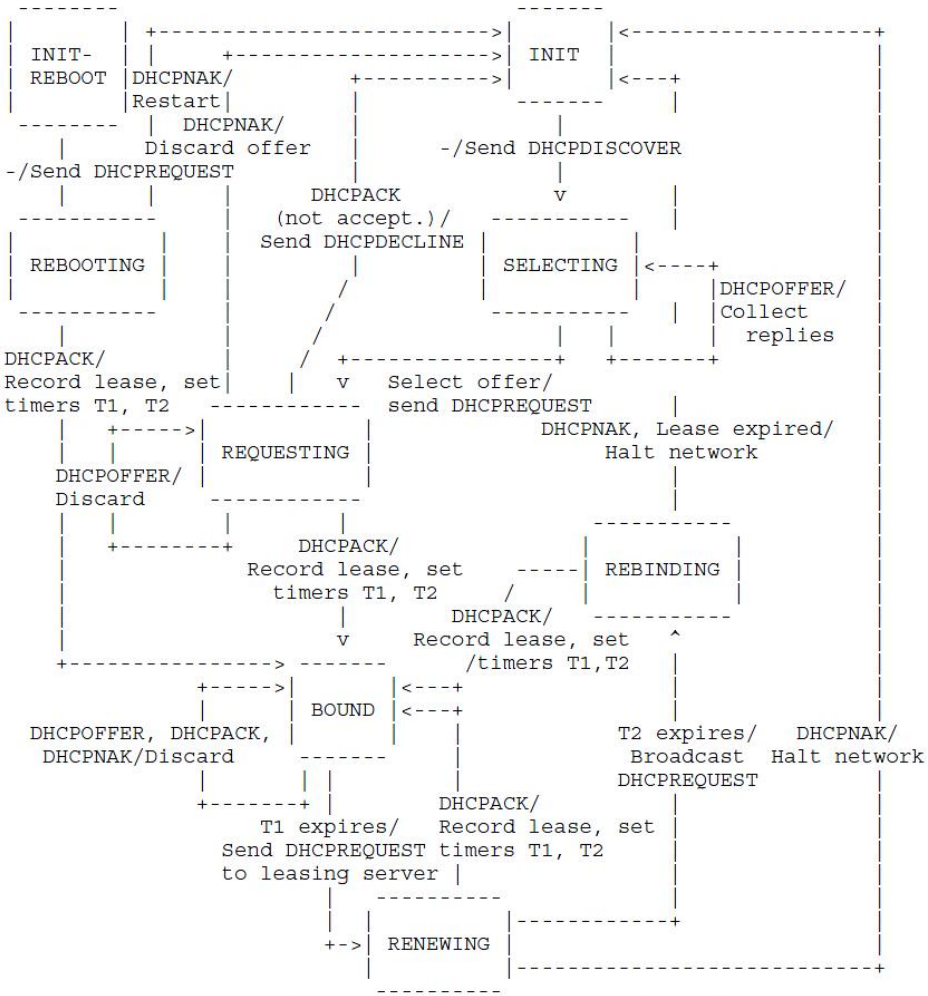
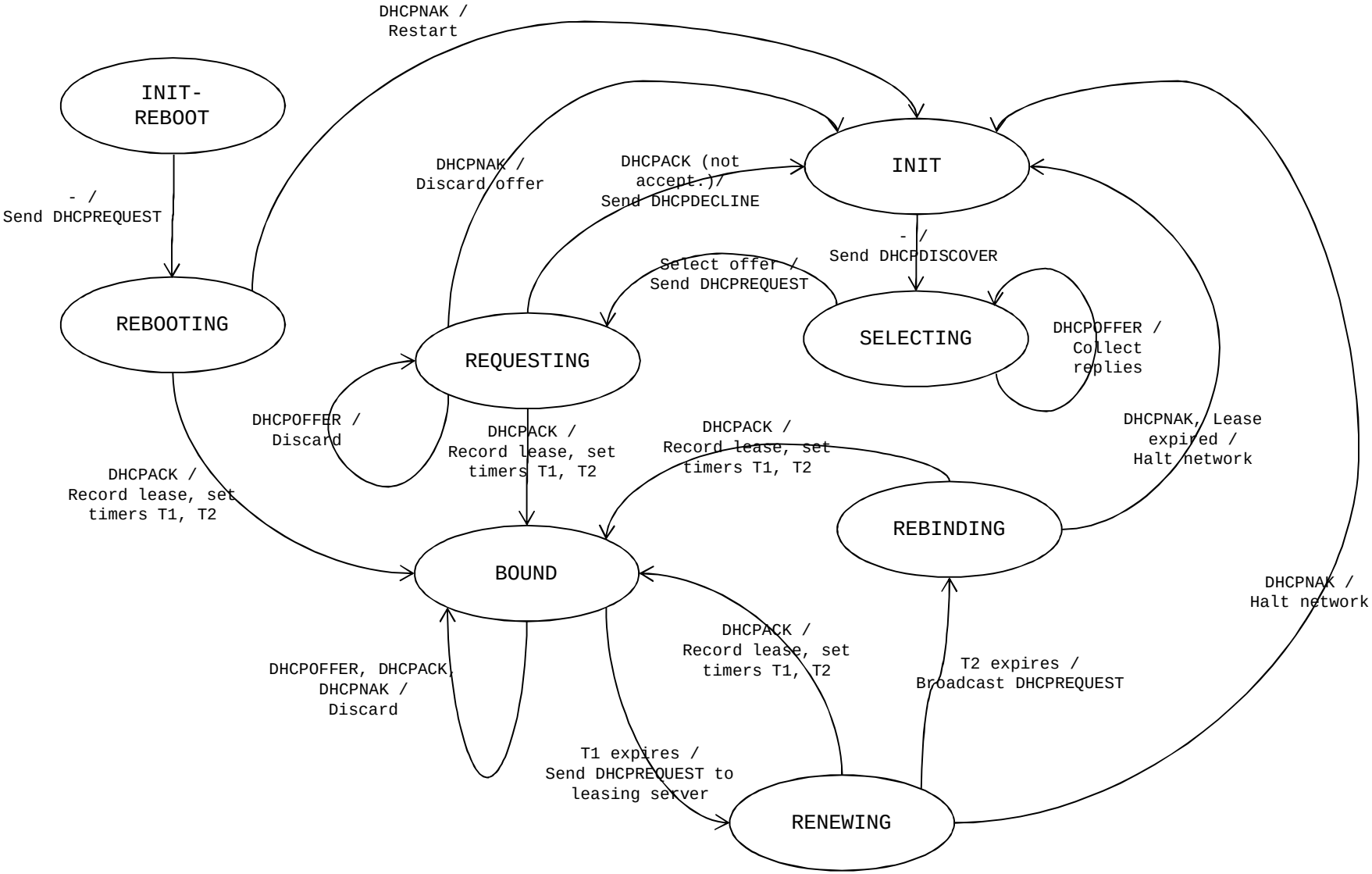


Figure 5: State-transition diagram for DHCP clients

DHCP tilakone (rfc2131)

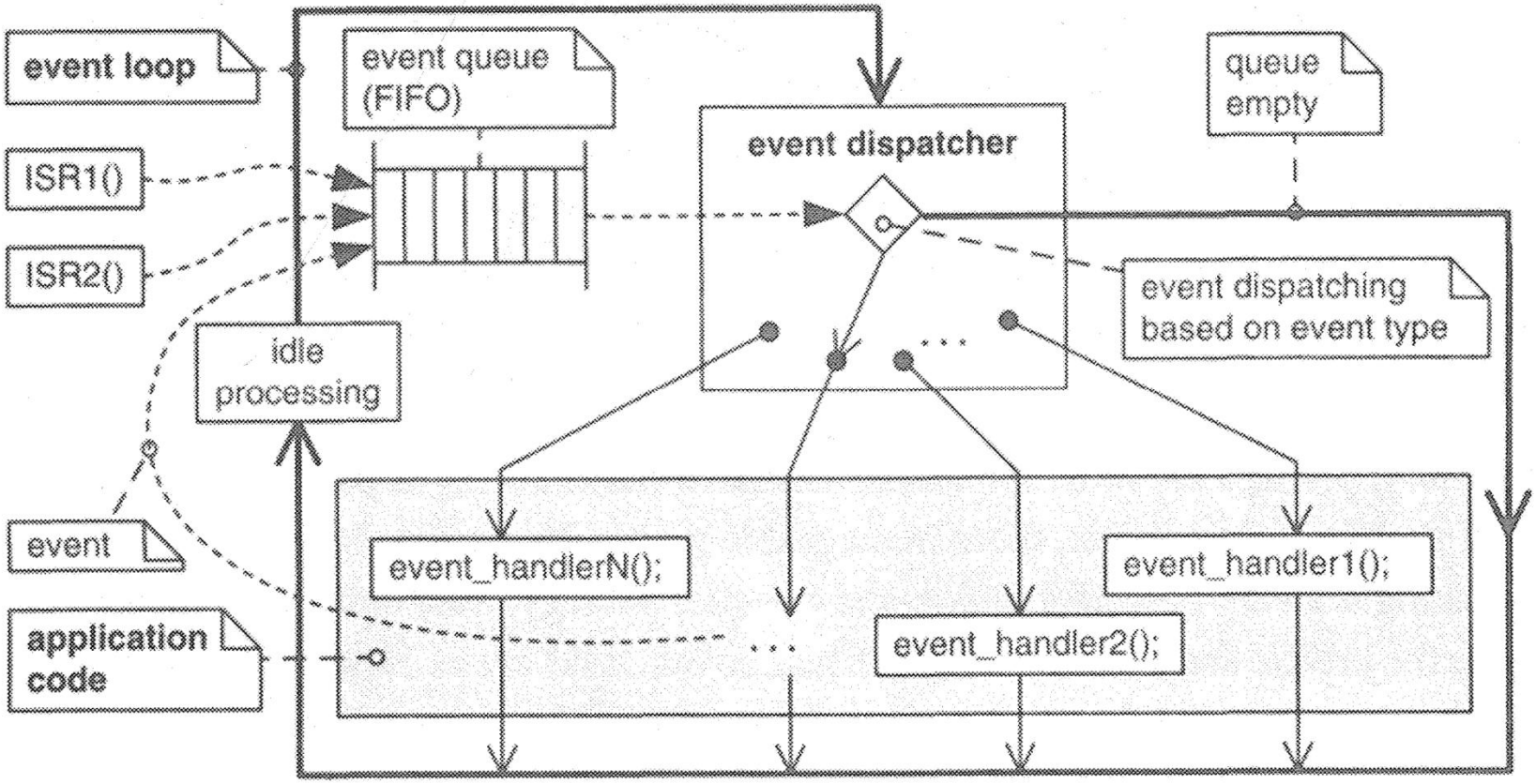


Tilakoneiden toteutustapoja

- Tilakone toteutetaan funktiolla, jolle annetaan parametrina käsiteltävä heräte
  - Herätteillä on vakiomuoto esim. tietue
  - Herätteiden tuottaminen kannattaa erottaa tilakoneesta, jolloin tilakone on yksinkertaisempi ja helpommin siirrettävissä ympäristöstä toiseen
  - Herätteet jonossa odottamassa käsittelyä
- Sisäkkäiset switch-case-lauseet
  - Ulompi valitsee tilan ja sisemmässä reagoidaan tapahtumaan
  - Toimii pienehköllä tilakoneella, mutta paisuu helposti hallitsemattoman suureksi

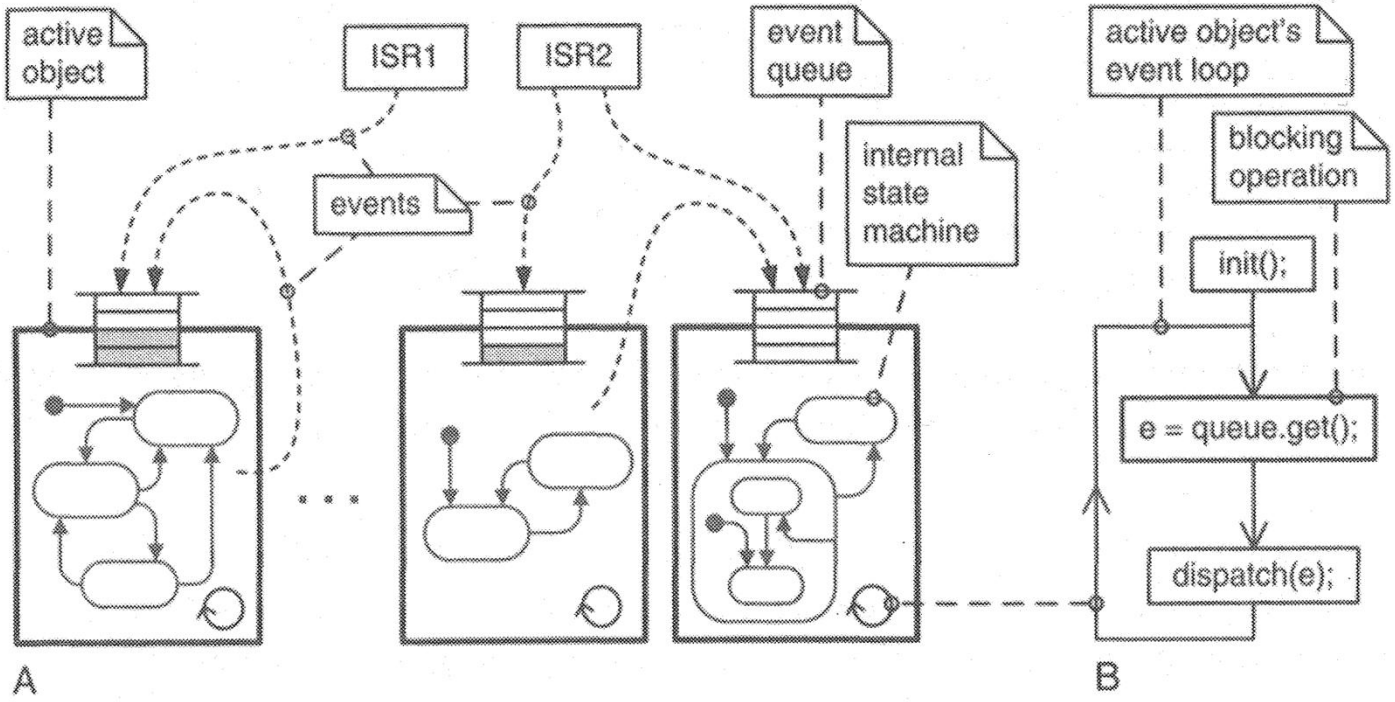
Event handler

- Käytännöllinen tapa hallita herätteitä on laittaa herätteet jonoon, josta ne sitten käsitellään järjestyksessä (tämä ajatusmalli on pohjalla esimerkiksi graafisissa käyttöliittymissä)



Active objects

- Toteutustapa, jossa jokainen tilakone on erillinen olio, jolla on oma herätejono
  - Tilakoneen ulkopuolelta tuotetaan herätteet tilakoneen jonoihin
  - Tilakone ei ota kantaa siihen mistä herätteet tulevat, joten tilakone on mahdollista testata erillisenä (testipenkissä verrataan herätteitä ja vasteita)



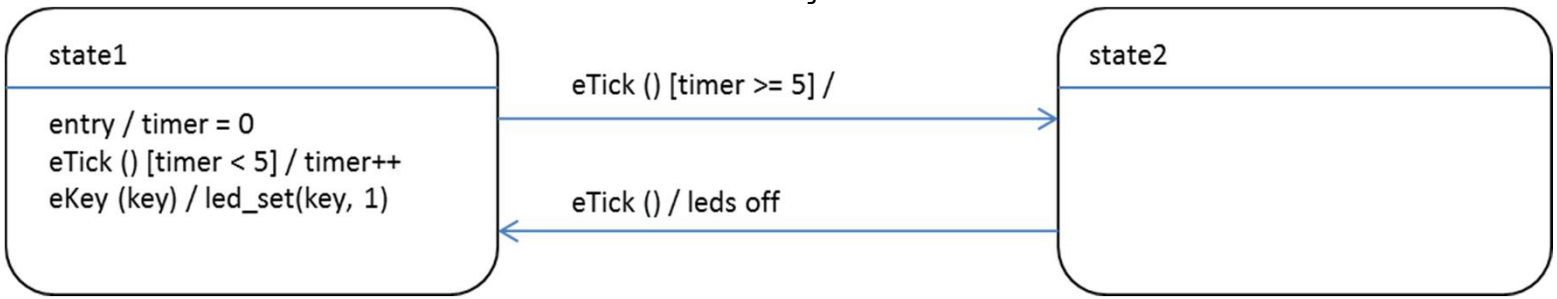
Toteutusperiaatteita

- Tilojen pitää olla mahdollisimman riippumattomia
  - Tilaan tultaessa toiminta ei saa riippua siitä mistä tilasta kyseiseen tilaan saavuttiin
    - Tällainen riippuvuus ei yleensä näy tilakaaviossa, vaan on toteutuksen ongelma
    - Tila on hajotettu osiin, jotka sijaitsevat eri puolilla lähdekoodia
- Globaalit tilamuuttujat estävät useamman kuin yhden tilakoneen instanssin ajamisen yhtä aikaa
  - Jos tarvitaan yhteisiä tilamuuttujia, niin niiden pitää sijaita tilakoneen instanssin tiedoissa
- Lähdekoodista pitää helposti pystyä tunnistamaan tilakaaviossa määritellyt tilat ja niiden toiminta
  - Lähdekoodin perusteella pitää pystyä piirtämään tilakaavio – jos ei pysty, niin yleensä ongelmia on vastassa

Yksinkertainen UML kaavio ja tilakone

```
void stState1(smi *me, const event *e)
{
    switch(e->type) {
    case eEnter:
        me->timer = 0;
        break;
    case eExit:
        break;
    case eKey:
        led_set(e->value, 1);
        break;
    case eTick:
        me->timer++;
        if(me->timer >= 5) TRAN(stState2);
        break;
    }
}
```

```
void stState2(smi *me, const event *e)
{
    switch(e->type) {
    case eEnter:
        break;
    case eExit:
        break;
    case eKey:
        break;
    case eTick:
        led_set(1, 0);
        led_set(2, 0);
        led_set(3, 0);
        led_set(4, 0);
        TRAN(stState1);
        break;
    }
}
```



UML tilakone (erittäin lyhyt oppimäärä)