F4
Testning och Parprogrammering i XP

EDA260
Programvaruutveckling i grupp – Projekt

Datavetenskap, LTH

XP's deltekniker (practices)

Kodning och design

Enkel design
Refaktorisering
Kodningsstandard
Gemensam vokabulär

Planering

Kund i teamet
Planeringsspelet
Regelbundna releaser
Hållbart tempo

Utveckling

Test-driven utveckling (TDD)
Parprogramming
Kollektivt kodägande
Kontinuerlig integration

Dessutom

Gemensamt utvecklingsrum
Nollte iterationen
Spikes

Fungerar systemet?

Verifiering – “Byggde vi systemet rätt?”

- Är systemet (tillräckligt) felfritt?
- Är systemet (tillräckligt) väldesignat? (så att vi kan modifiera det)

Validering – “Byggde vi rätt system?”

- Byggde vi det som kunden förväntade sig?
- Är användarna nöjda med systemet?

Tekniker för att förhindra fel

Testning

- kör programmet på test-indata, kontrollera att det ger rätt respons

Granskningar (av kod, specifikationer, designmodeller, ...)

- effektivt (men tidskrävande)

Obs!

- Med kodgranskning och testning kan vi hitta fel, men inte bevisa att programmet är felfritt.

Testnivåer

Hur mycket av systemet är inblandat i testerna?

- Enhetstest, t.ex. enskild klass
- Modultest, t.ex. ett paket
- Subsystem, t.ex. ett lager
- Systemtest (hela systemet)

Integrationstest

- Allt större delar av systemet testas efterhand som det utvecklas och integreras

Acceptanstest

- Tester baserade på användarnas krav

Vad kan man testa?

- Funktionen (rätt resultat)
- Användarvänlighet (“usability”)
- Prestanda
- Överlastat system (“stress testing”)
- ...

Strategier för att ta fram testfall

Blackbox testing

- Välj testfall baserat på specifikation och gränssnitt

Whitebox testing

- Välj testfall baserat på intern struktur och implementation

Sviter av testfall

Regressionstestning

- Efter en ändring körs en svit av gamla testfall igenom för att kontrollera att programvaran inte “regredierar”. Dvs man vill verifiera att det som fungerade tidigare fortfarande fungerar efter ändringen.

Automatiserad testning

- Ett verktyg kör igenom en svit av testfall och kontrollerar om de verkliga utfallen stämmer med de förväntade utfallen.

Testning och Granskning i XP

Enhetstester

- Utvecklarnas verktyg för att verifiera sin egen kod

Test First

- Testfallen driver utvecklingen framåt

Automatiserad regressionstestning

- Alla testfall måste fungera för att man skall få integrera

Acceptanstester

- Kundens krav för att en användarberättelse skall vara klar

Parprogrammering

- Kontinuerlig kodgranskning

Enhetstester (Unit Tests)

Varje klass testas

BankAccount
SEK balance() void deposit(SEK) void withdraw(SEK)

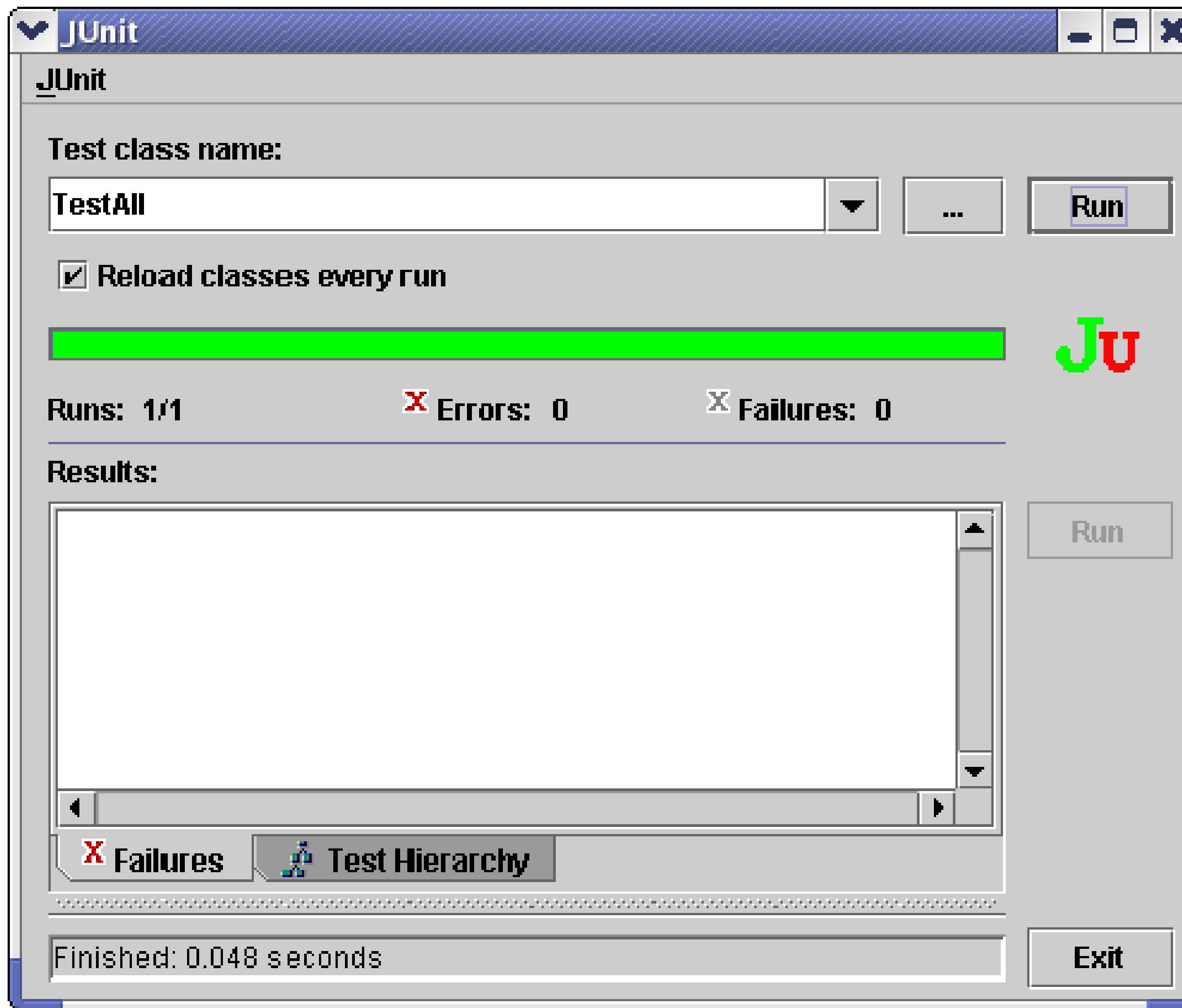
TestBankAccount
void initialBalanceZero() void simpleTransactions() void withdrawToZero() void smallOverdraft() void largeOverdraft() void depositEliminatesOverdraft() ...

Exempel på testmetoder

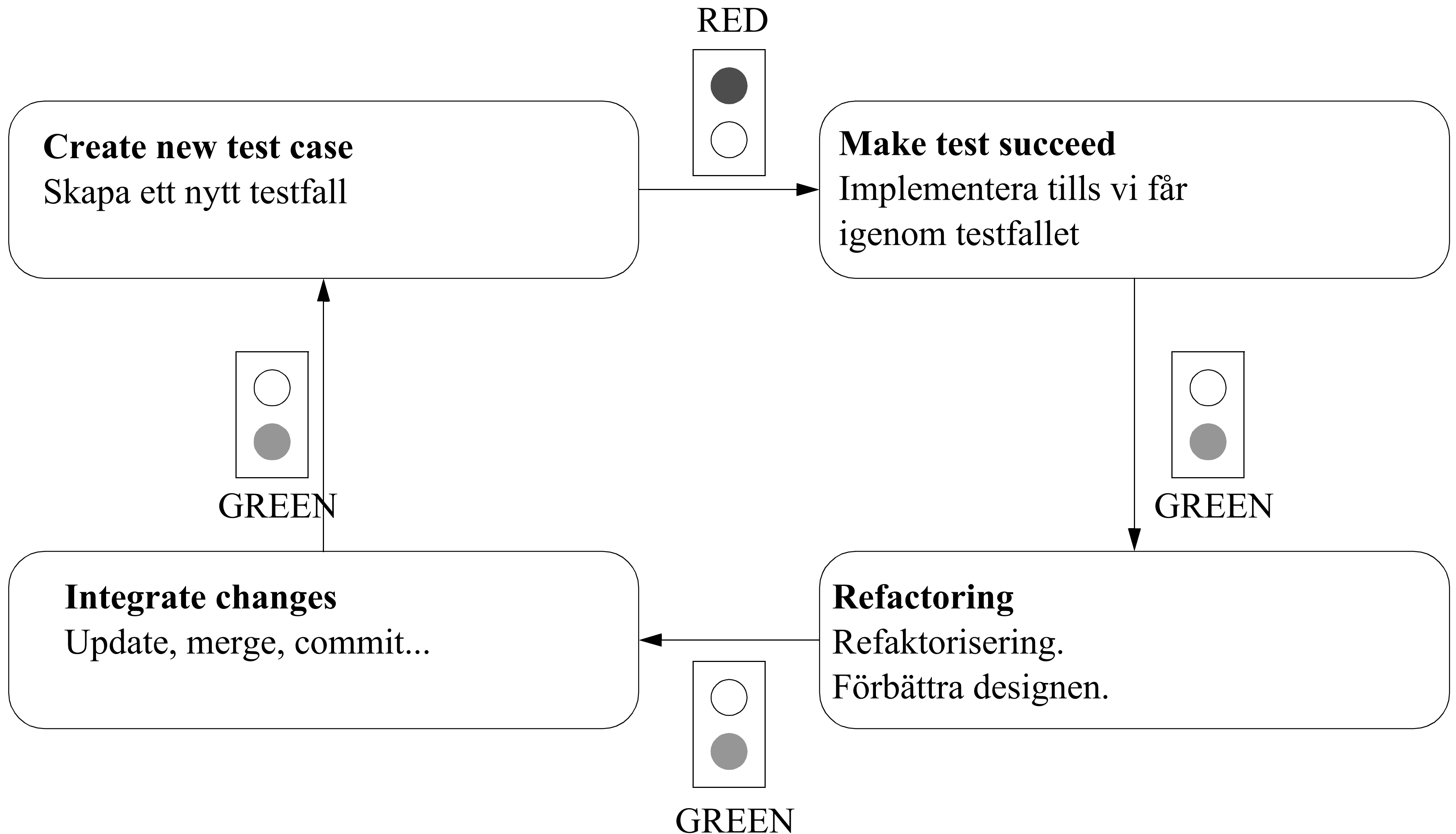
```
@Test void initialBalanceZero() {
    BankAccount account = new BankAccount();
  assertEquals(           "Incorrect initial balance",
                          new SEK(0), account.balance());
}


@Test void simpleTransactions() {
    BankAccount account = new BankAccount();
    account.deposit(new SEK(50));
    account.deposit(new SEK(75));
    account.withdraw(new SEK(32));
    assertEquals(         "Incorrect simple transactions",
                          new SEK(93), account.balance());
}
```

JUnit-verktyget



Skriv testfall före kod (Test First)



Test List

- Skriv upp alla testfall du kommer på på en todo-lista
- Komplettera med fler testfall och önskade refaktoriser-ingar efter hand som du implementerar
- Vid parprogrammering: partnern kan hålla ordning på listan

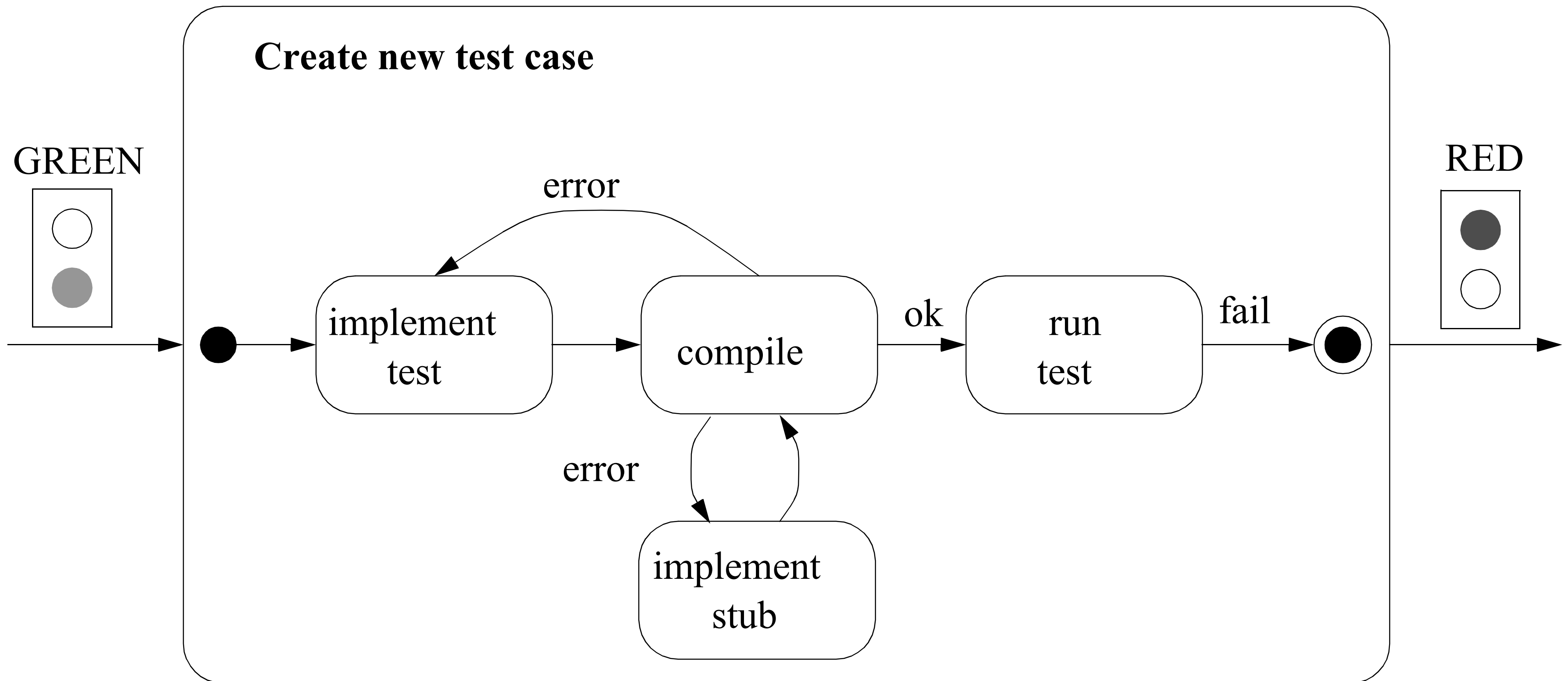
Ett Test First scenario

- saldo
- sätt in pengar
- ta ut pengar
- övertrassering
- ränta
- kontoutdrag
- ....

One step test

- Vilket testfall skall vi beta av härnäst?
- Ta ett som inte är trivialt och inte för svårt – något som man lär sig av och som för designen framåt.

Skapa ett nytt testfall



Varför vill vi se testen fallera?

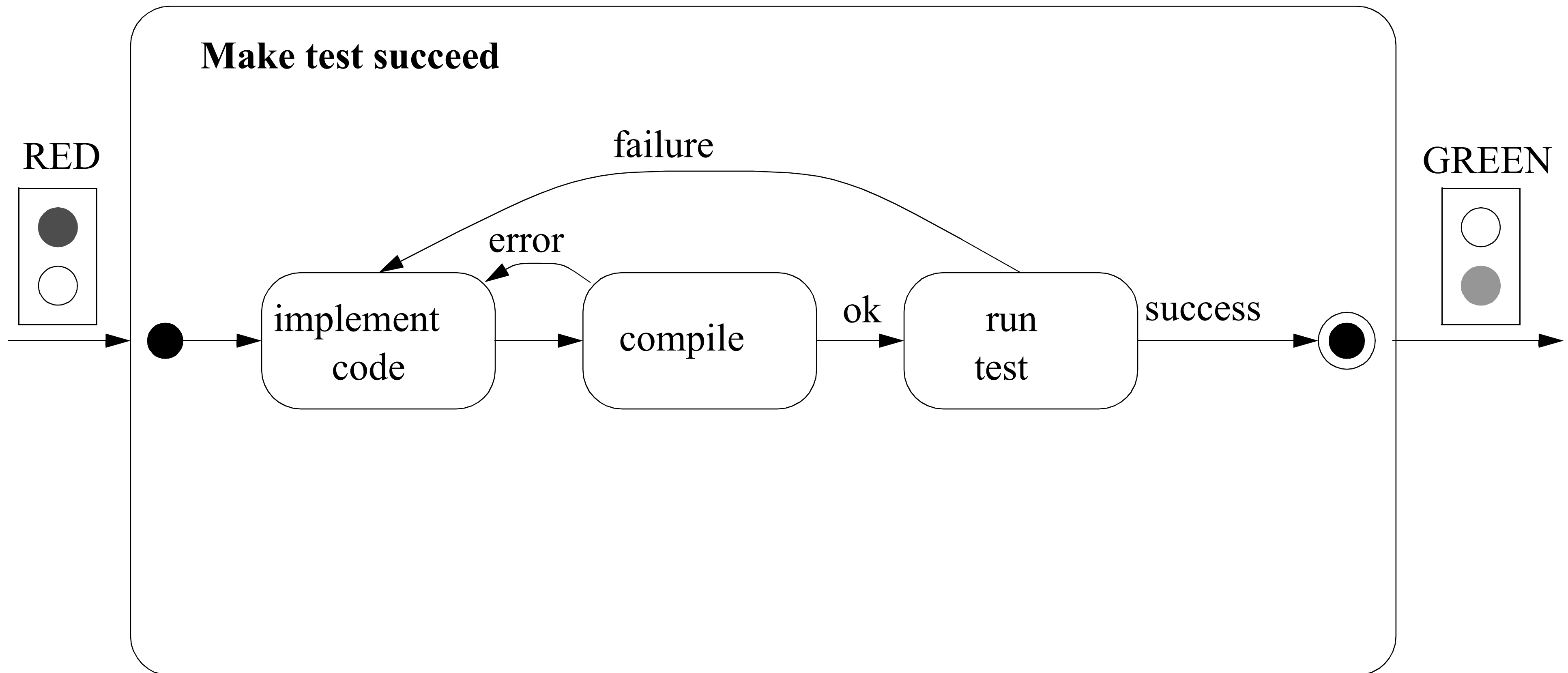
Om testen fallerar

- Kvitto på att vi tänkt rätt
- Testen testar faktiskt något nytt
- Vi vet att testfallet *kan* fallera på förväntat sätt

Men om testen går igenom utan att vi implementerat något?

- Varningssignal – har vi tänkt fel någonstans?
- Har vi kodat “i förväg” tidigare?
- Testar vi något som redan testas?
- Har vi skrivit testfallet på fel sätt?
- Men det kan också vara helt OK! T.ex., vi vill försäkra oss om att ett gränsvärde alltid kommer att fungera.

Få igenom testfallet



Fokusera på att få igenom testfallet

Om vi kommer på fler saker som borde göras...

- nya testfall
- förbättrad design
- ...

... skriv upp dem på ToDo-listan istället för att implementera dem med en gång

- därmed minimeras tiden då vi är i ett osäkert tillstånd (“rött ljus”)
- lättare att arbeta fokuserat och inte missa detaljer
- lättare att debugga då endast en sak ändrats
- lättare att hela tiden känna att man kommer framåt

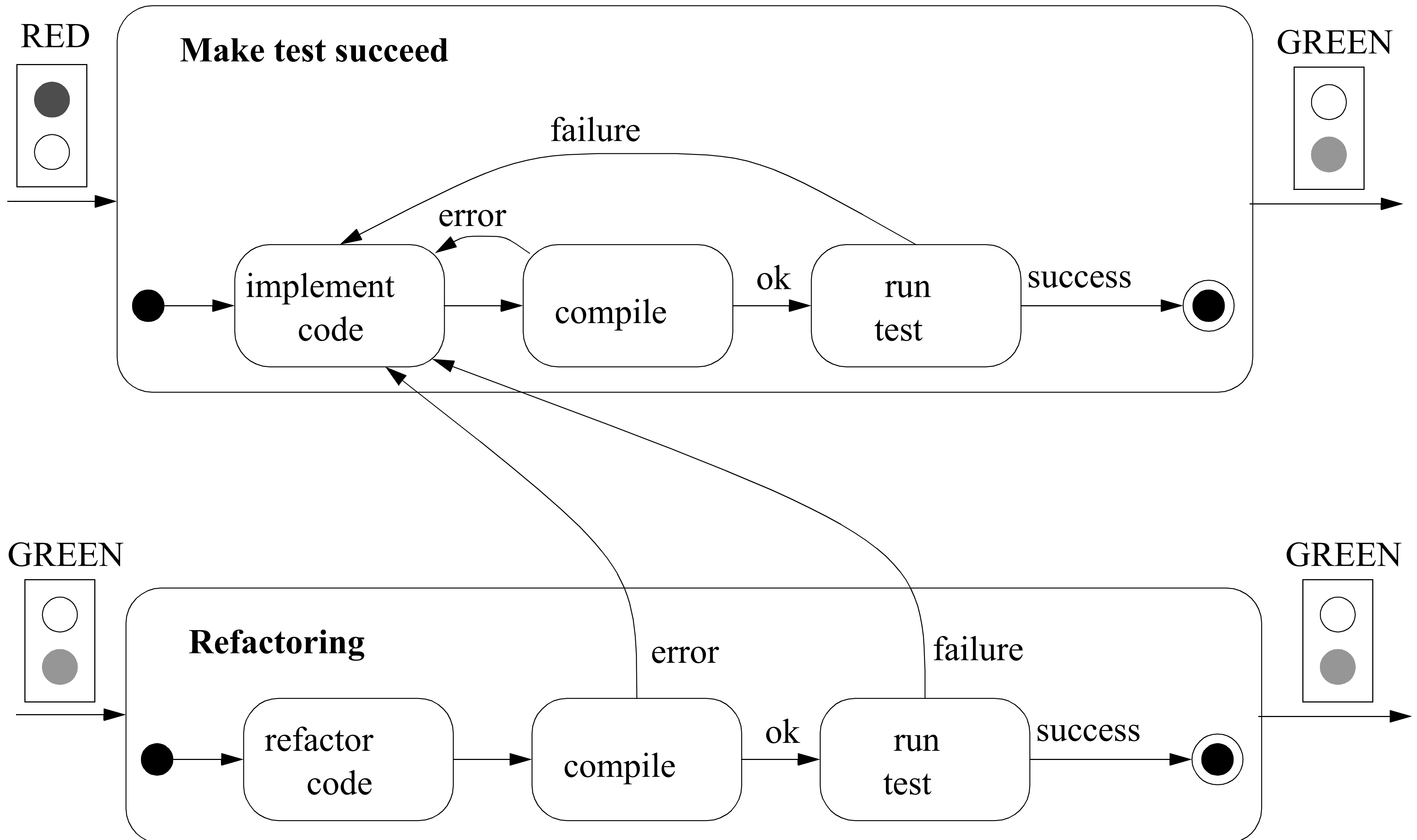
Efter att vi fått igenom ett nytt testfall

Reflektera

- Blev helheten bra?
- Har vi goda namn på klasser, metoder, variabler, ...?
- Har vi råkat få duplicerad kod?

Refaktorisera vid behov

Refaktorisering



Dags att integrera!

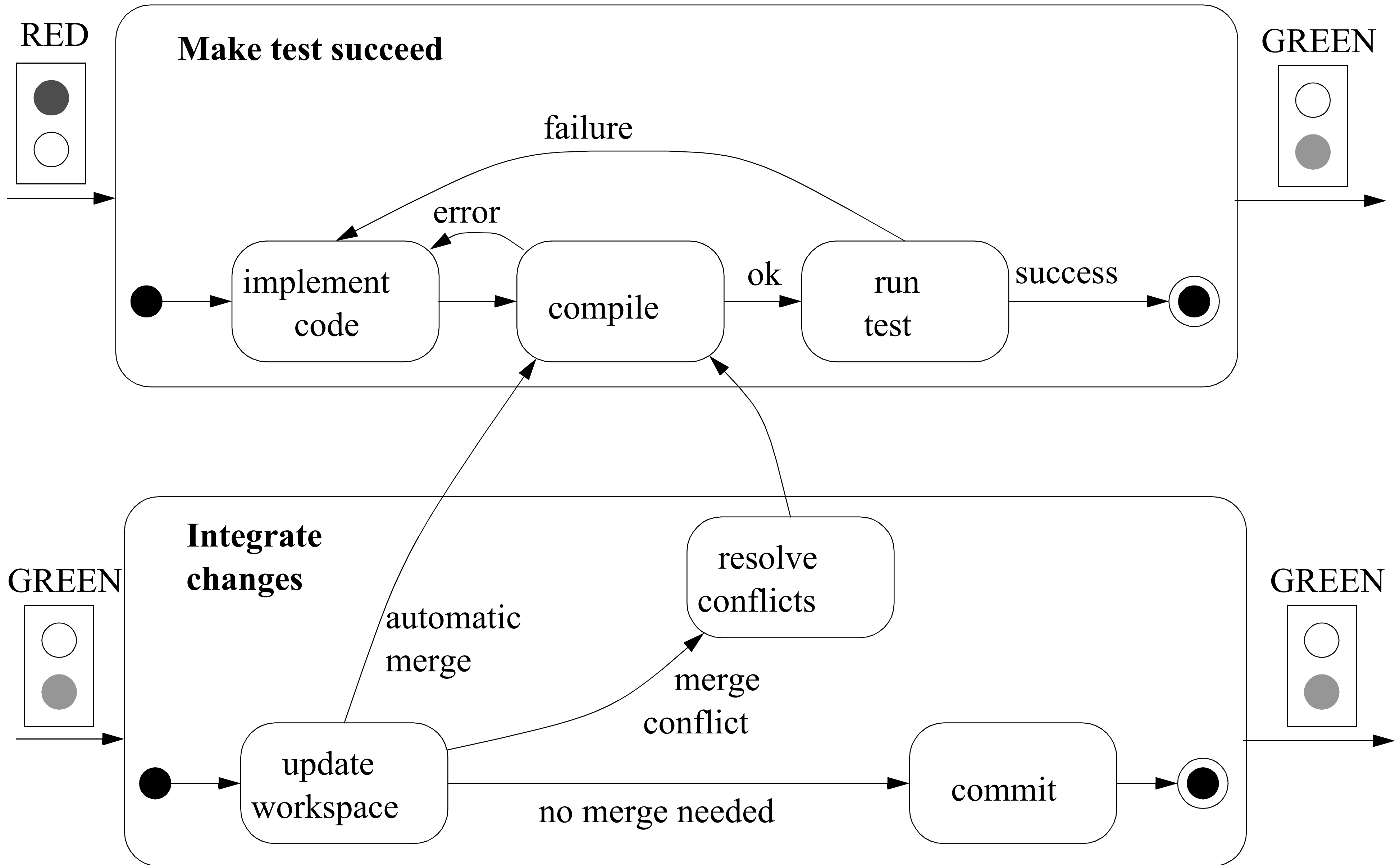
Nu har vi...

- implementerat och testat ny funktionalitet
- refaktoriserat till en enkel och tydlig design
- grönt ljus (100% av alla existerande enhetstester fungerar)

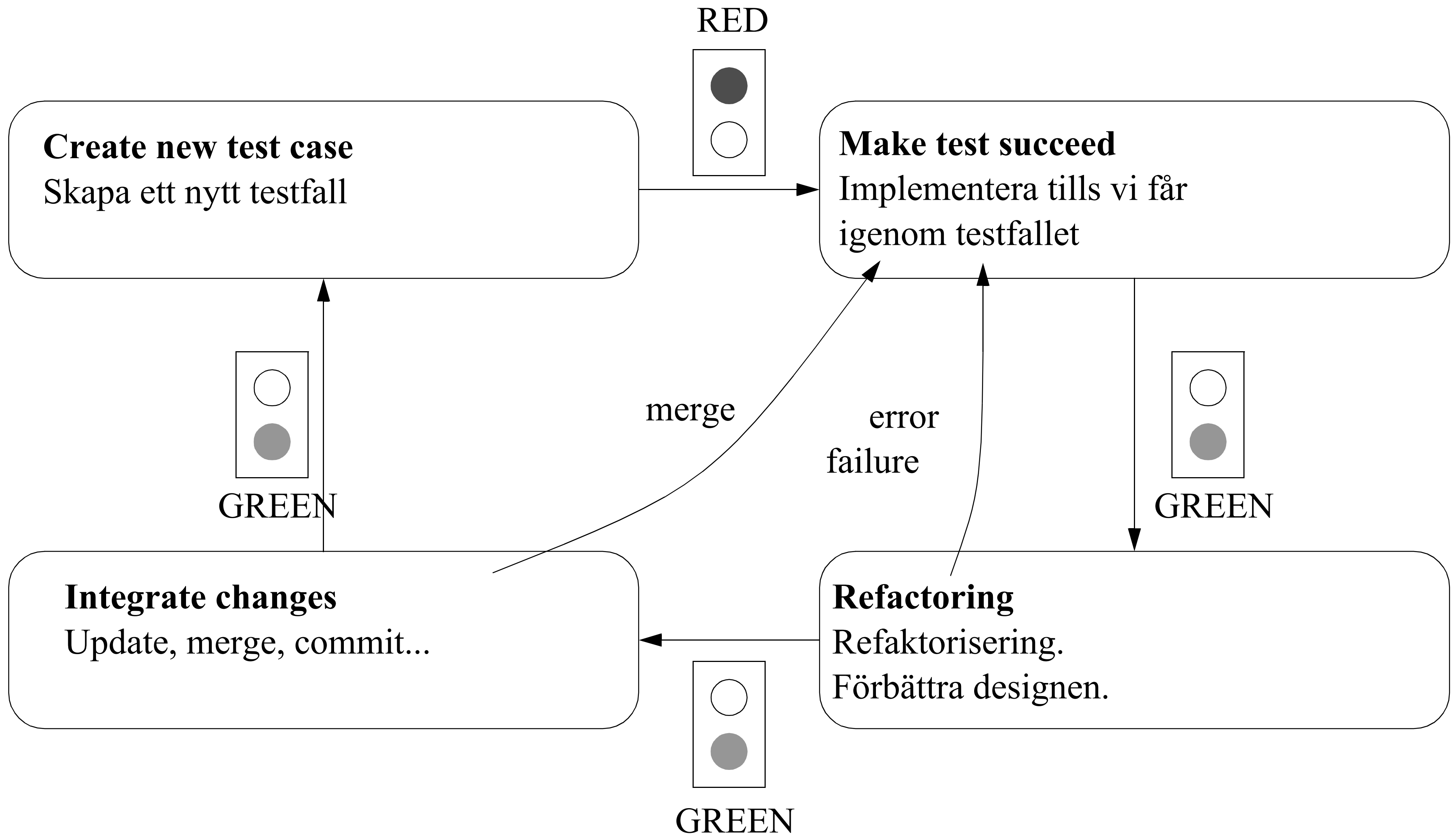
Villkor för att vi skall få integrera vår kod i det gemensamma repositoriet

- all vår kod går att kompilera utan felmeddelanden
- alla enhetstester (100%) går igenom
- vår version är baserad på den senaste versionen i repositoriet

Integrera ändringar



Test First



Fördelar med Test First

Testfallen driver utvecklingen framåt

- Hjälper oss att fokusera på en sak i taget
- Hjälper oss att minimera den tid programmet (testerna) inte fungerar
- Om något testfall slutar fungera så vet vi att det beror på våra senaste ändringar
- Det är lätt att veta när ett delproblem är färdigimplementerat

Underlättar implementationen av testfallen

- All kod blir faktiskt testad
- Test First är lättare än Test Last: Det är enklare att anpassa produktionskod till färdiga testfall än testfall till färdig produktionskod.

Underlättar design

- Test First resulterar ofta i bra gränssnitt till klasserna.
- Testerna fungerar som säkerhetsnät – vi vågar ändra koden och bibehålla och förbättra designen

Parprogrammering

Två personer vid samma maskin

- Driver och Partner
- Båda är aktiva
- Ständig kommunikation
- Rollerna växlas ofta

All produktionskod skrivs i par

- Omdelbar kodgranskning
- Inte nödvändigt vid spikes och acceptanstest

Par ändras ofta

- Många/alla har insikt i alla delar av koden
- Nya medarbetare har lättare för att komma in i projektet

Driver och Partner

Driver

- Skriver kod
- Väljer strategi (i samråd med partner)
- Beskriver sin intention för partnern

Partner (Navigator)

- Granskar kod
- Ger direkt feedback på design, metodnamn, ...
- Följer samma strategi som föraren
- Håller ordning på ToDo-listan.

Parprogrammeringstips

Byt partner ofta

Tacka alltid ja när du blir tillfrågad

Tala i Vi-termer i stället för Du:

- “Borde vi inte göra så här i stället?”

Tala i Jag-termer när det blir krångligt:

- “Jag förstår inte. Kan du förklara?”

Parprogrammeringstips, forts

Driver

- Var lyhörd inför din partner
- Rusa inte iväg utan beskriv dina idéer

Partner

- Hitta förarens rytm
- Föreslå rollbyte om ni kör fast
- men ge föraren en chans att fullfölja vald strategi

Goda parprogrammeringsvanor

Ta pauser

- Man blir trött av parprogrammering
- Ta kort paus då och då... när ni integrerat, när ni fått igenom ett nytt test, en gång i timmen, ...

Ödmjukhet

- “Egoless programming” – ni programmerar tillsammans för teamet
- Ta tillfället i akt att lära dig och att lära ut

Ha självförtroende

- Var inte rädd – din partner är där för att hjälpa dig
- Din partner vet inte allting bättre än du

Goda vanor

Kommunicera/Lyssna

- Förare – berätta hela tiden vad du tänker: “vi behöver en temporär-variabel här för iterationen...”.
- Partner – fråga, kommentera, se till att du hänger med på vad som sker. Föreslå förbättringar.

Var en “team player”

- Driver och partner är gemensamt ansvariga för koden.
- Uppstår en bugg eller dålig design – skyll inte på den andre. Hjälps åt att lösa problemet i stället.
- När något går bra – ta åt er äran tillsammans.

Mer om testning...

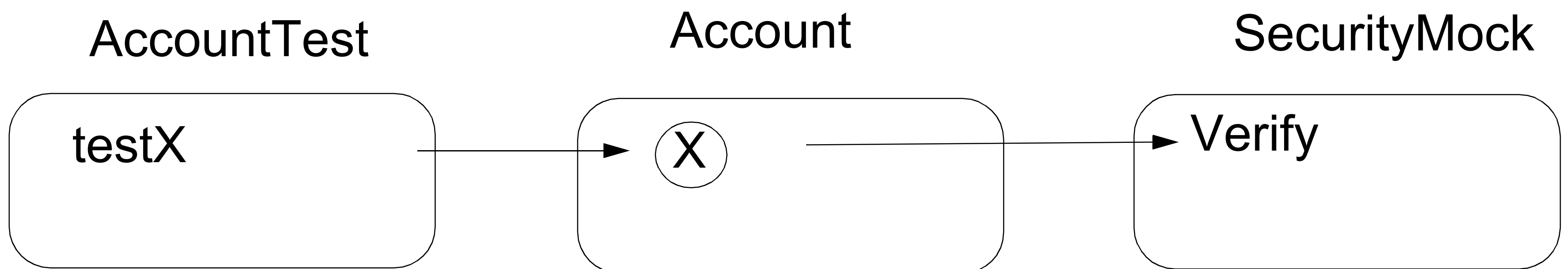
Fördelar med automatiserad testning

Säkerhetsnät vid ändringar

- Man vågar lägga till ny funktionalitet
- Man vågar förbättra designen (refaktorisering)
- Man vågar rätta buggar (snarare än koda runt dem)

Mock object

- Hur testar man ett objekt som behöver en extern/ komplicerad/långsam resurs – som kanske inte ens är implementerad än?
- Skapa en dummy – ett Mock object – för den externa resursen, som returnerar konstanter som passar testet.

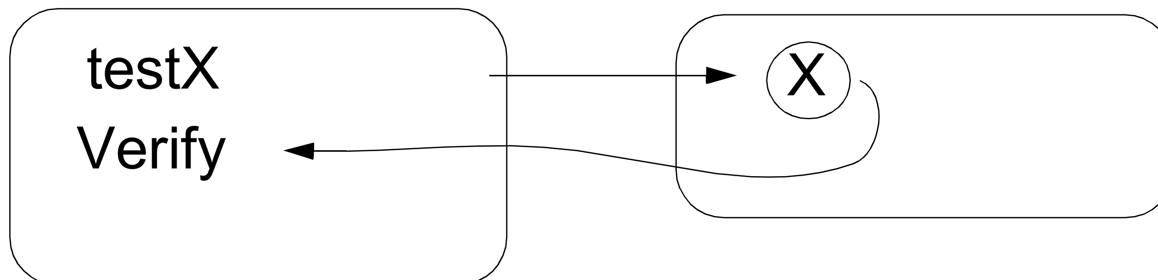


Self shunt

- Hur testar man att ett objekt anropar ett klientobjekt på rätt sätt?
- Skapa ett mock-objekt för klienten.
  - Använd testklassen själv för detta mock-objekt (self shunt).
  - Koden för att starta förloppet och fånga anropet hamnar tillsammans (i testklassen).
  - Kräver att mock-objektet anropas via interface

AccountTest
implements Security

Account



Hur mycket skall man testa?

Testningsstrategier

- identifiera typiska fall (equivalence partitioning)
- testa randvärden, t.ex. -1, 0, 1, null, length, ... (boundary values)
- se till att all produktionskod exerceras av något testfall (statement coverage)
- ...
- XP: “Test everything that could possibly break”

Hur stor del av koden är testkod?

- Det beror på tillämpningen.
- Runt hälften är en vanlig siffra.

Hierarkiska tester?

Enhetstester

- Om en klass kan testas isolerat från andra klasser kan fallerande testfall lättare avlusas
- Vid behov: koda upp “mock objects” för att simulera interaktion med andra klasser (klasser som inte finns än, funktionalitet som är komplex och som vi vill isolera oss från)

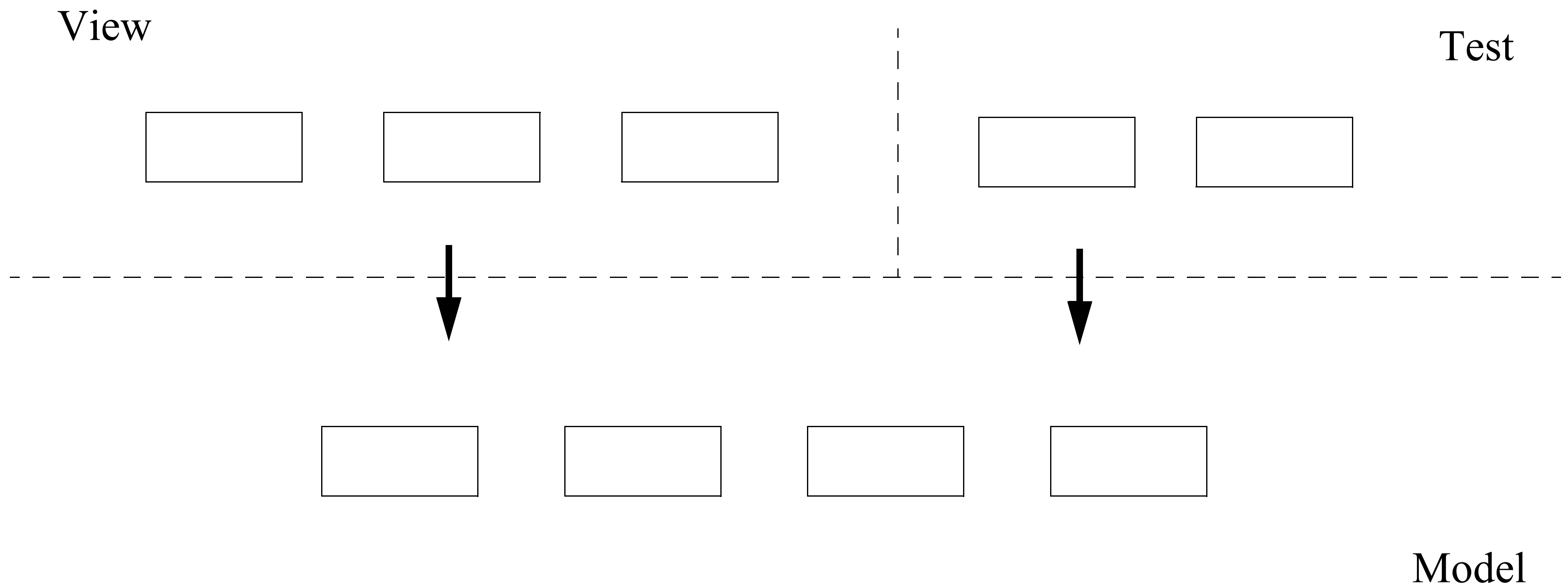
Modultester, subsystemtester, ...?

- Går utmärkt att automatisera med JUnit på samma sätt som enhets-tester.
- Men låt inte dessa tester ersätta enhetstester. Enhetstesterna behövs för att lättare isolera fel.

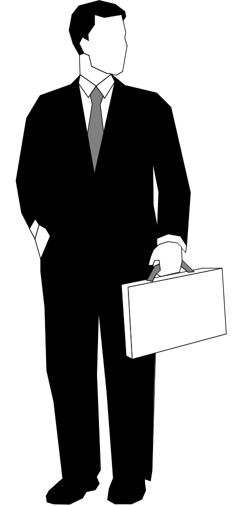
Testning av interaktiva tillämpningar

Isolera modell (Model) från användarinteraktion (View)

- Gör View-lagret så tunt som möjligt
- Modellen kan testas med automatiserade tester



Acceptanstester



Kunden

- Tänker ut ett antal testfall för varje användarberättelse
- “Vad skulle övertyga mig om att denna användarberättelse är implementerad?”
- Kan vara riktiga testdata, t.ex. från tidigare verksamhet

Projektledning

- Acceptanstesterna mäter hur projektet framskrider

Utvecklarna

- Hjälper kunden att implementera testfallen
- Automatisera så mycket som möjligt av acceptanstesterna
- Acceptanstesterna kan utvecklas parallellt med produktionskoden
- Kan utvecklas av enskild programmerare (utan parprogrammering)

Litteratur

- Kent Beck: Test-Driven Development by example. Addison-Wesley, 2003
- Bill Wake's site: http://www.xp123.com/xplor
  - The Test-First Stoplight
  - The Test/Code Cycle in XP

Lab 3: Testning och parprogrammering

Förberedelser

- Lab exercise: Test First using the JUnit Testing Framework
- Using JUnit in EDA260
- F4- föreläsningsbilder
- Laurie Williams and Robert Kessler: Pair Programming Illuminated, Addison-Wesley, 2003. Utdrag: delar av kap 1 (Introduction, p 3-5) + kap 27 (Seven Habits of Effective Pair Programmers).
- - William C. Wake: Extreme Programming Explored, Addison-Wesley, 2002. Utdrag: Delar av kap. 1 (How Do You Write a Program?, p 3-10 + 20-21).
- - chromatic: Delar av Part II: p 25-32 (Adopt Test-Driven Development, Practice Pair Programming)

Kalkylblad (spreadsheet)

	A	B	C	D	E	F	...
1	**Beställning**						
2	**Artikel**	**Antal**	**Färg**	**Storlek**	**Styckpris**	**Pris**	
3	mössa	1	röd	L	99:50	99:50	
4	strumpor	10	blå	M	19:50	195:00	
5							
6							
7					**Netto:**	294:50	
8					**Frakt:**	34:00	
9					**Brutto:**	328:50	

Exempel på testfall

[Bill Wake, http://xp123.com/xplor/xp0201/]

Basic functionality – storing values in cells

- ThatCellsAreEmptyByDefault
- ThatTextCellsAreStored
- ThatManyCellsExist
- ThatNumericCellsAreIdentifiedAndStored

Simple formulas

- ThatWeHaveAccessToCellLiteralValuesForEditing
- FormulaSpec
- ConstantFormula
- Add

Dependencies

- checkThatCellReferenceWorks
- checkThatCellChangesPropagate
- ...

checkThatCellsAreEmptyByDefault

```
@Test public void checkThatCellsAreEmptyByDefault() {
  Sheet sheet = new Sheet();
  assertEquals("", sheet.get("A1"));
  assertEquals("", sheet.get("ZX347"));
}
```

checkThatTextCellsAreStored

```
@Test public void checkTextCellsAreStored() {
  Sheet sheet = new Sheet();
  String theCell = "A21";

  sheet.put(theCell, "A string");
  assertEquals("A string", sheet.get(theCell));

  sheet.put(theCell, "A different string");
  assertEquals("A different string", sheet.get(theCell));

  sheet.put(theCell, "");
  assertEquals("", sheet.get(theCell));
}
```

checkThatCellReferenceWorks

```
@Test public void checkCellReferenceWorks () {
  Sheet sheet = new Sheet();
  sheet.put("A1", "8");
  sheet.put("A2", "=A1");
  assertEquals("cell lookup", "8", sheet.get("A2"));
}
```