# F4
# Testning och Parprogrammering i XP

EDA260
Programvaruutveckling i grupp – Projekt
Boris Magnusson,Datavetenskap, LTH

# XP:s Deltekniker (Practices)

| | |
|---|---|
| **1. Planering**<br>Planeringsspelet<br>Regelbundna releaser<br>Hållbart tempo<br>Kund i teamet | **2. Utveckling**<br>**Test-driven utveckling (TDD)**<br>**Parprogrammering**<br>Kollektivt ägande<br>**Kontinuerlig integration** |
| **3. Kodning och Design**<br>Enkel design<br>Refaktorisering<br>Kodningsstandard<br>Gemensam vokabulär | **4. Dessutom**<br>Gemensamt utvecklingsrum<br>Nollte iterationen<br>Spikes |

# Testning:
# - Fungerar systemet?

- **Verifiering** - ”Byggde vi systemet rätt?”
  - Är systemet (tillräckligt) felfritt?
  - Är systemet (tillräckligt) väldesignat (kan vi modifiera det) ?
- **Validering** - ”Byggde vi rätt system?”
  - Byggde vi det som kunden förväntade sig?
  - Är användarna nöjda med systemet?

# Tekniker som förhindrar fel

- **Testning**
  - kör programmet på testdata - kontrollera om det ger förväntad respons
- **Granskning** (av kod, specifikationer, design…)
  - effektivt men tidskrävande
- Vi kan bara hitta existerande fel, inte visa att något är felfritt.

# Testnivåer

- Hur mycket av systemet är inblandat i testerna?
  - Enhetstester, t ex enskild klass
  - Modultest, t ex ett paket i Java
  - Subsystem, t ex ett lager, komponent
  - Systemtest - hela systemet
- Integrationstest
  - Allt större delar av systemet testas tillsammans
- Acceptanstest
  - Tester baserat på användarnas krav.

# Vad kan man testa?

- Funktion (rätt resultat)
- Användarvänlighet (”usability”)
- Prestanda, tid och minne, bandbredd …
- Överlastat system, ”stress testning”,

# Strategier för att ta fram testfall

- **Blackbox** testning
  - Välj testfall baserat på specifikationer, API
  - Utifrånperspektiv
- Whitebox tesning
  - Välj testfall baserat på implementation, intern struktur
  - ”Test coverage” - sträva efter att all kod testas

# När körs testfall?

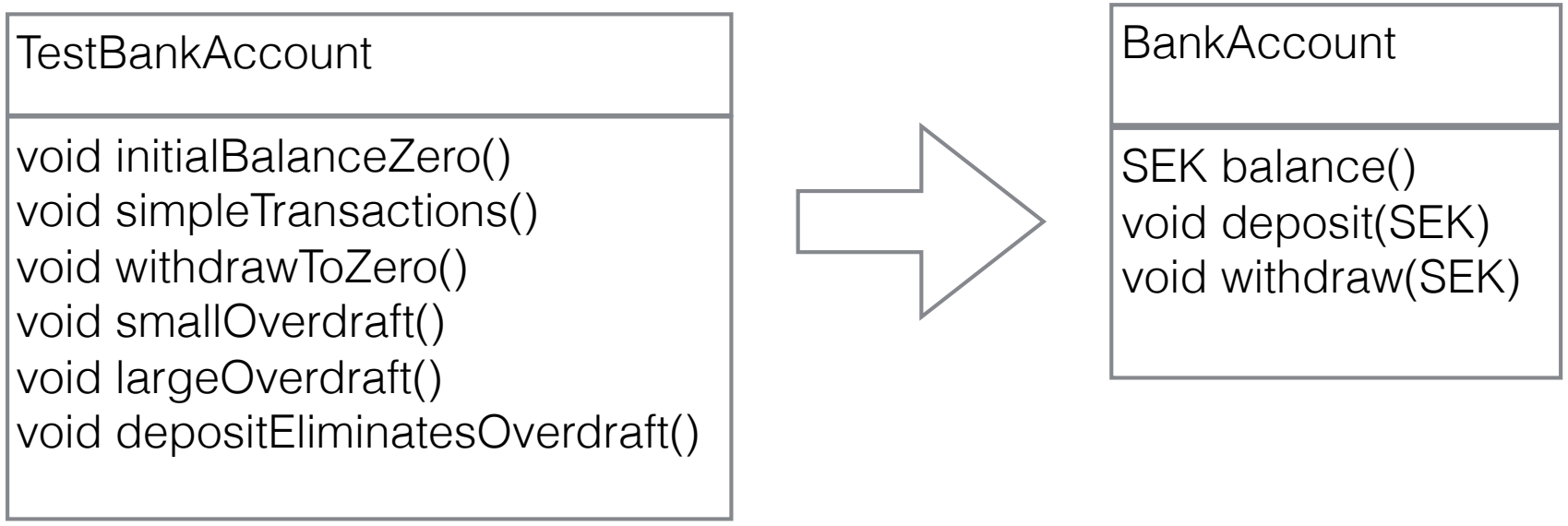
- **Regressionstestning** - nästan alltid
  - Efter en ändring, en integration,
  - Kontrollera att systemet inte ”regredierar” - återfaller till ett felaktigt beteende.
- **Automatisk testning** - mindre arbete
  - Ett verktyg (script kanske)
  - Kör igenom testfallen och verifierar (diff) att utfallet stämmer med det förväntade.

# Testning och Granskning i XP

- **Enhetstester**
  - Utvecklarens verktyg för att verifiera sin egen kod
- Test First
  - Testfallen driver utvecklingen framåt.
- Automatiserad regressionstestning
  - Alla testfall måste fungera innan integration
- **Acceptanstester**
  - Kundens krav för att en story skall vara klar
- **Parprogrammering**
  - Kontinuerlig kodgranskning

# Enhetstester

- Varje klass testas

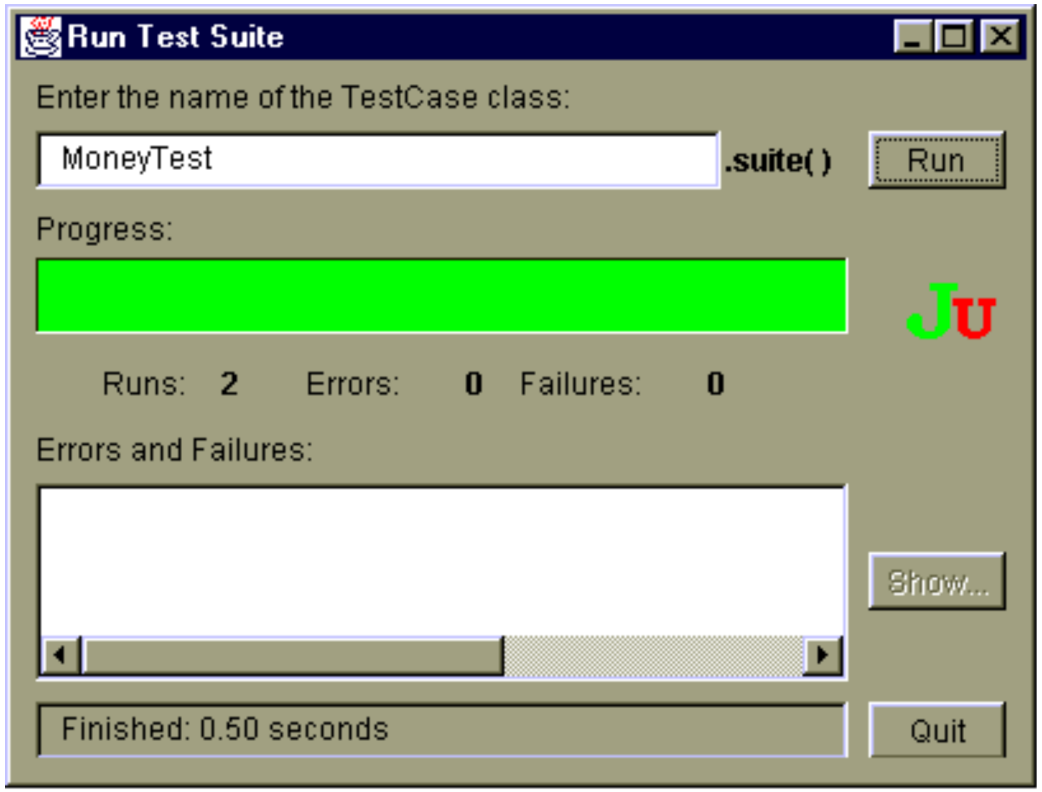


# Exempel på testmetoder

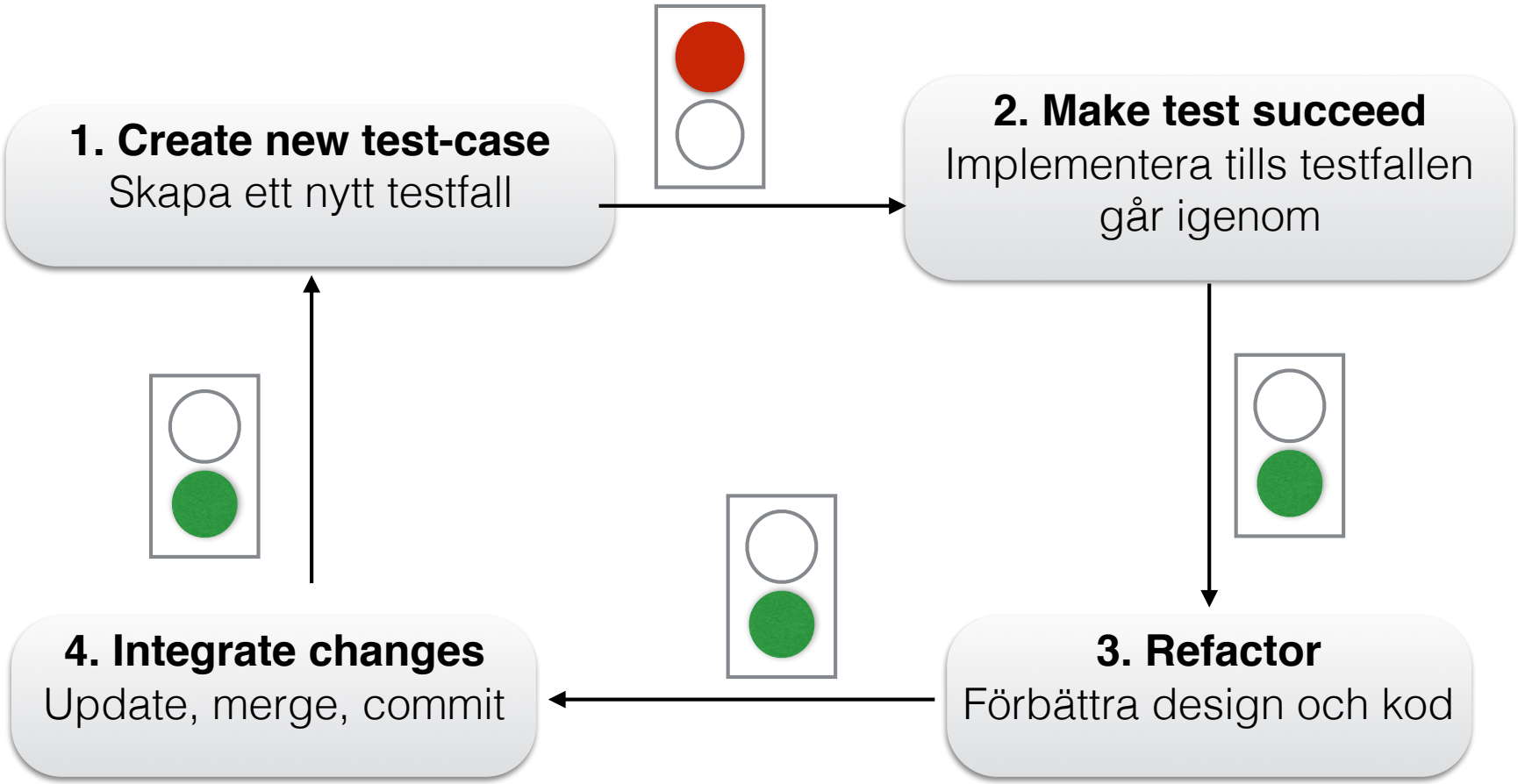
```
@Test void initialBalanceZero() {
  BankAccount account = new BankAccount();
  assertEquals(”Incorrect initial balance”,
    new SEK(0), account.balance());
}
```

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

# jUnit -verktyget



# A) TestFirst:
# *Skriv testfall före kod*



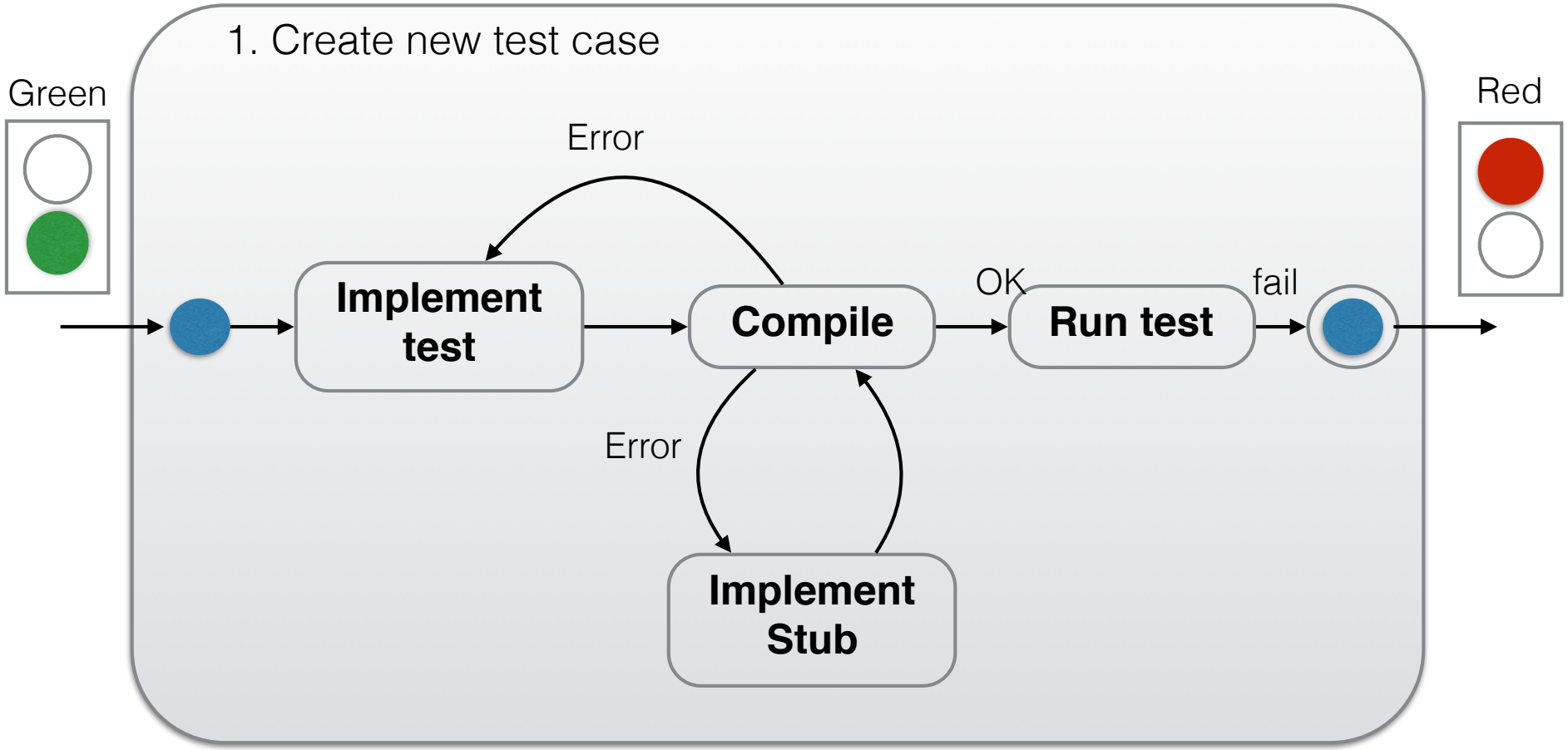
# Test ToDo-Lista

- Skriv upp alla testfall du kommer på på en ToDo-lista
- Komplettera listan löpande med fler testfall och önskade refactoriseringar när du kodar.
  - Vid parprogrammering: Partnern håller ordning på ToDo-listan.
- **Test First scenarion ToDo-lista**:
  - saldo
  - sätt in pengar
  - ta ut pengar
  - övertrassering
  - ränta
  - konoutdrag

# One-step test

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

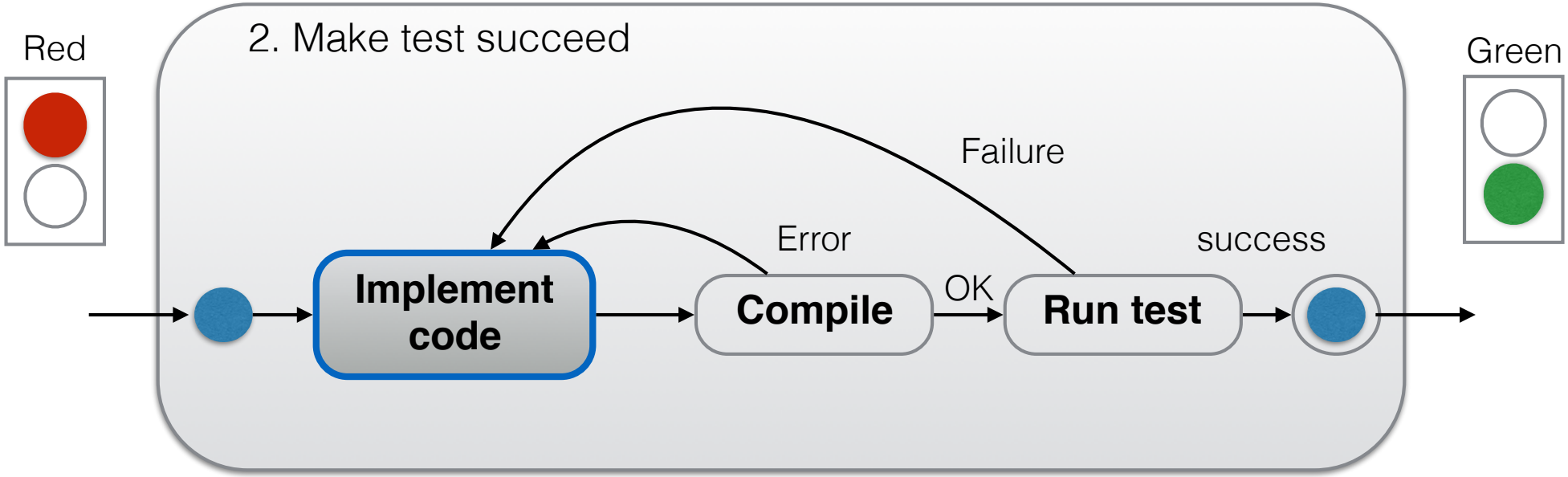
# 1. Skapa nytt testfall



# 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 implementera något?
  - Varningssignal —har vi tänkt fel någonstans?
  - Har vi kodat ”i förväg” tidigare - mer att testa?
  - Testar vi något som redan testas?
  - Har vi skrivit testfallet på fel sätt
  - Kan vara OK - fyller i tester som saknas etc.

# 2. Få igenom tesfallet



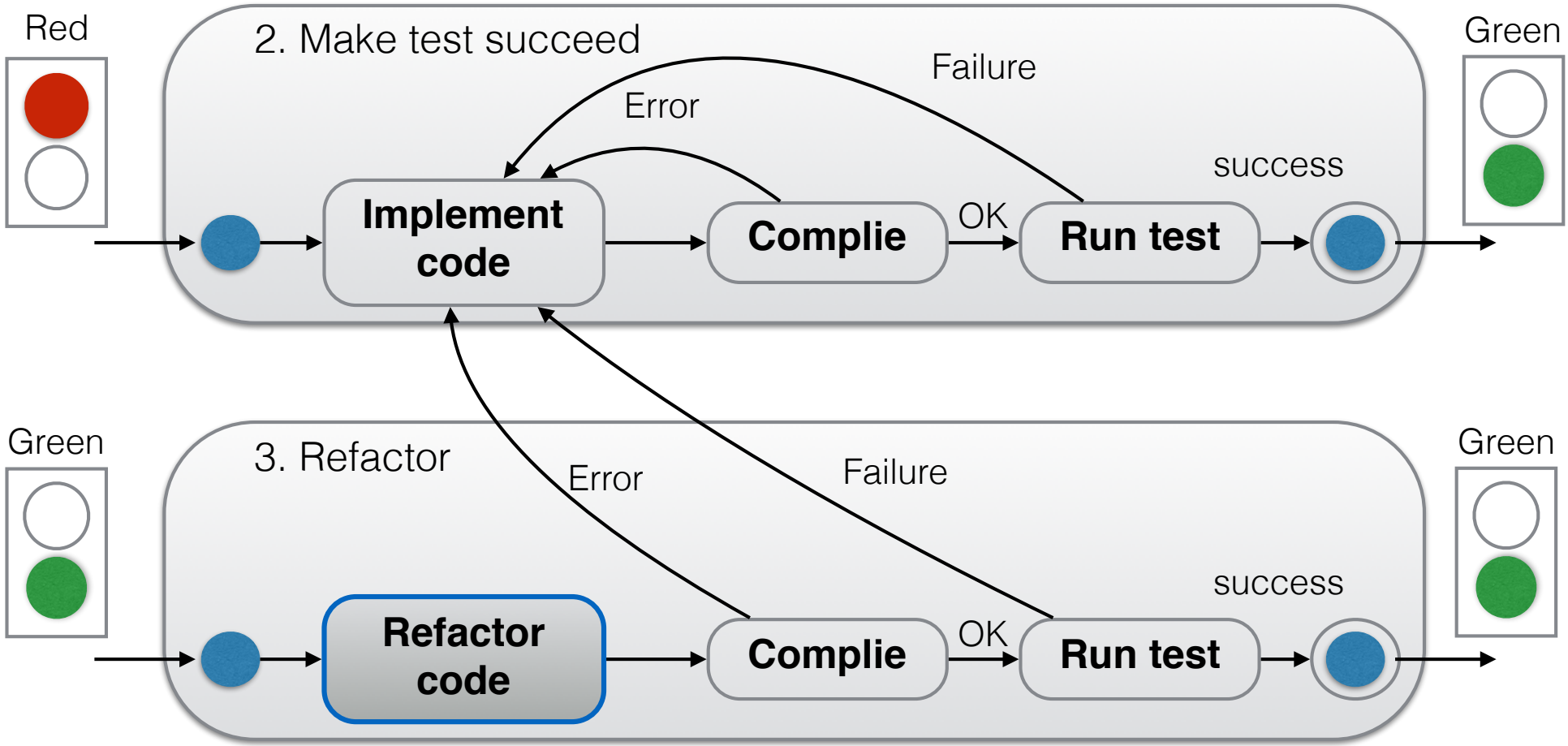
# Fokusera på att få igenom testfallet

- Om vi kommer på fler saker som borde göras:
  - nytt testfall
  - förbättrad design
  - …
- Skriv upp dem på ToDo-listan istället för att ta i dem med en gång
  - minimera tiden med ”rött ljus”, osäkerhet
  - lättare att arbete fokuserat och inte missa detaljer
  - lättare 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 bra namn på:
    - klasser,
    - metoder,
    - variabler ?
  - Har vi råkat få duplicerad kod?
- Refaktorisera där det behövs.

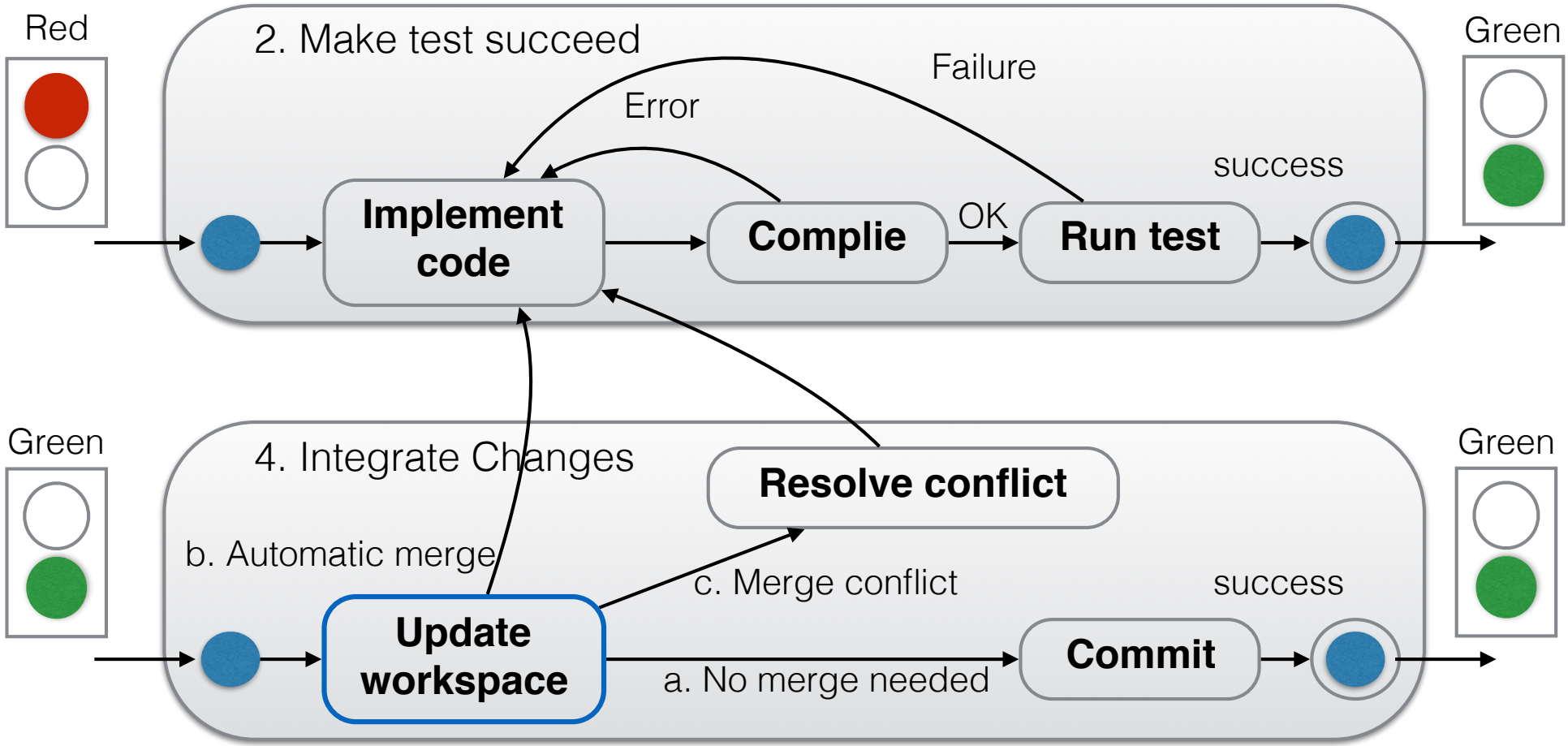
# 3. Refaktorisera



# Dags att integrera!

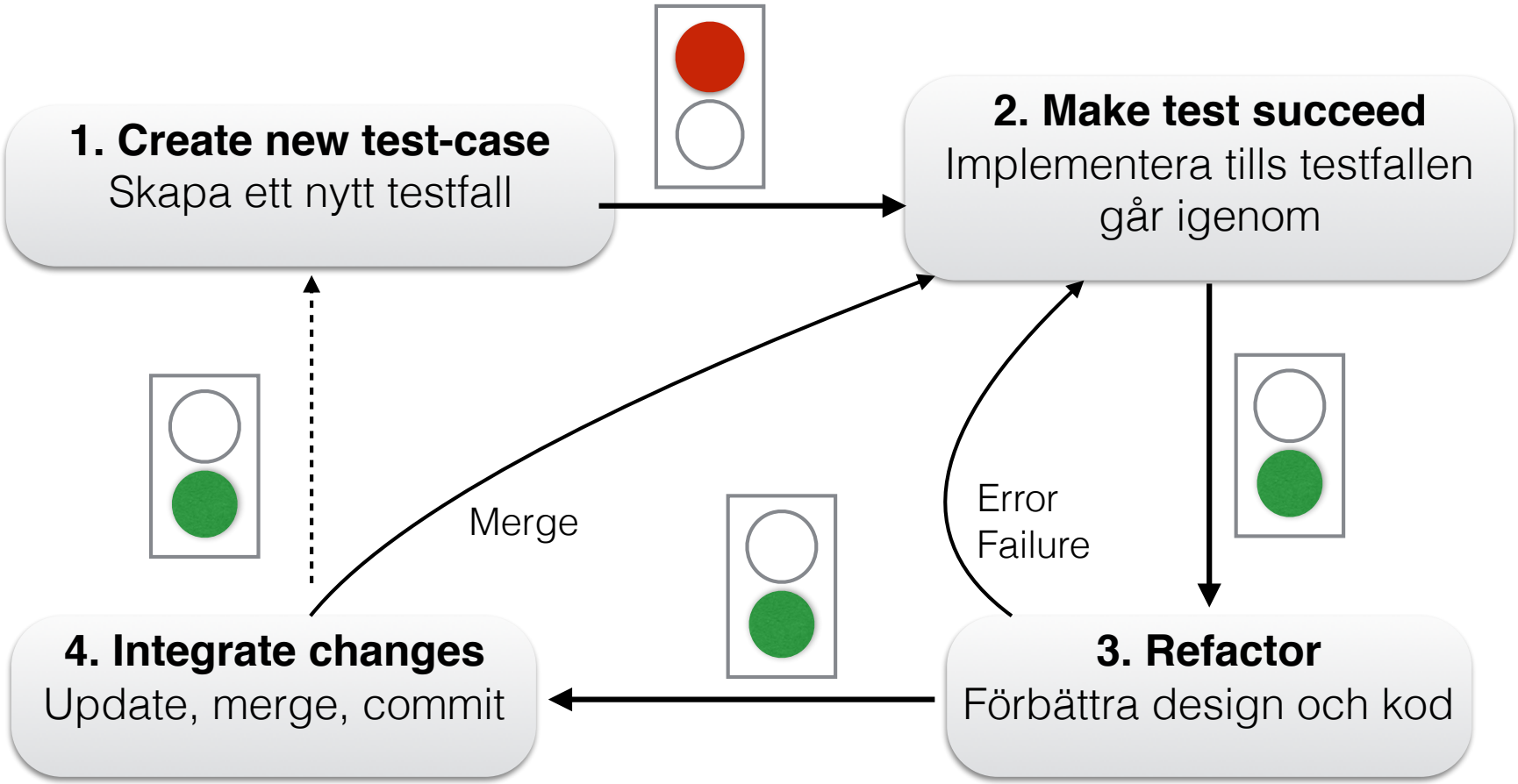
- Nu har vi:
  - implementerat och testat ny funktionalitet
  - refaktoriserat till en enkel och tydlig design
  - grönt ljus - 100% av existerande enhetstester fungerar
- Villkor för att integrera med gemensamma repositoriet:
  1. All kod kompilerar utan felmeddelanden
  2. Alla enhetstester går igenom
  3. Vår version baserad på senaste i repositoriet!

# 4. Integrera ändringar




a. Inga Commit sedan vi gjorde Update förra gången
b. Inga ändringar i samma filer
c. Ändringar i samma filer

# TestFirst


1. Create new test-case
Skapa ett nytt testfall
2. Make test succeed
Implementera tills testfallen går igenom
Merge
Error
Failure
4. Integrate changes
Update, merge, commit
3. Refactor
Förbättra design och kod

# Fördelar med Test First

- **Testfallen driver utvecklingen framåt (blir en agenda)**
  - Hjälper oss fokusera på en sak i taget
  - Hjälper oss minimera den tid programmet inte fungerar
  - När något testfall slutar fungera beror det på våra senaste ändringar
- **Underlättar implementation av testfallen**
  - All kod blir faktiskt testad (coverage)
  - Test first lättare än Test last - Enklare anpassa produktionskod till testerna än tvärt om!
- **Underlättar design**
  - Test first testar även gränssnitten till klasserna - blir bättre
  - Testerna fungera som säkerhetsnät - vi vågar ändra koden och vi vågar förbättra designen.

# B) Parprogrammering

- Två personer vid samma maskin
  - Driver och Partner
  - Båda är aktiva
  - Ständig kommunikation
  - Roller växlas ofta
- All produktionskod skrivs i par
  - Omedelbar kodgranskning
  - Inte nödvändigt vid spikes och acceptanstest
- Par ändras ofta:
  - Mång/alla får insikt i alla delar av koden
  - Nya medarbetare har lättare för komma in i projektet

# Två roller

- **Driver** (Förare)
  - Skriver kod
  - Väljer strategi (i samråd)
  - Beskriver sina intentioner för partnern
- **Partner** (Navigator)
  - Granskar kod
  - Ger 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 (2)

- 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.

# Goda programmeringsvanor

- **Ta pauser**
  - Man blir trött av parprogramering
  - Ta korta pauser då och då, … när ni integrerat, … när ett test gått igenom, …en gång i timmen.
- **Ödmjukhet**
  - ”Egoless programming” - ni programmerar för teamet
  - Ta tillfället att lära dig och att lära ut
- **Ha självförtroende**
  - Var inte rädd - din partner är där för att hjälpa dig
  - Din partner vet inte alltid bättre än du.

# Goda vanor

- **Kommunicera/Lyssna**
  - **Förare - berätta** hela tiden vad du tänker: ”vi behöver en temporär variabel 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.
  - När något går bra - ta åt er äran tillsammans!

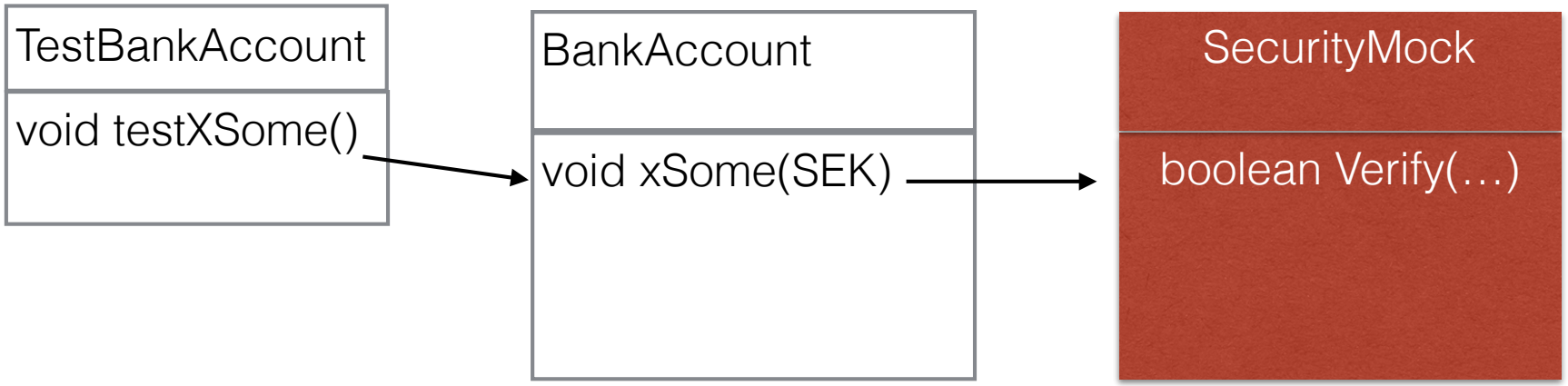
# Mer om testning …

# Fördelar med automatiserad testning

- Säkerhetsnät vid ändringar:
  - Man vågar lägga till ny funktionalitet
    - på rätt plats snarare än som ”ryggsäckar”
  - Man vågar förbättra designen
    - refactoring - ändra kod/design
  - Man vågar rätta buggar
    - snarare än koda runt dem

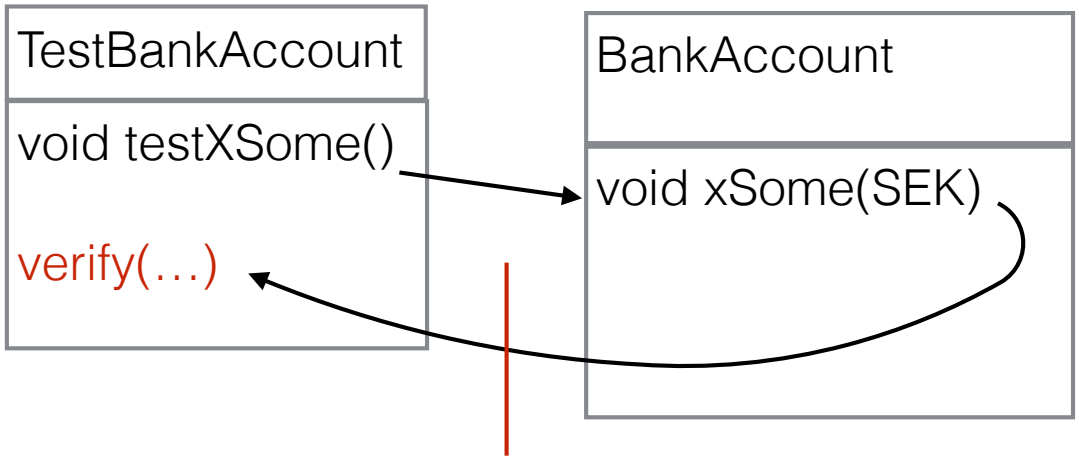
# Mock objects

- Hur testar man klasser som behöver en omgivning? - komplicerad, kanske inte implementerad ännu?
- Skapa en dummy - ett Mock object - för det som saknas.
- Konfigureras på något sätt



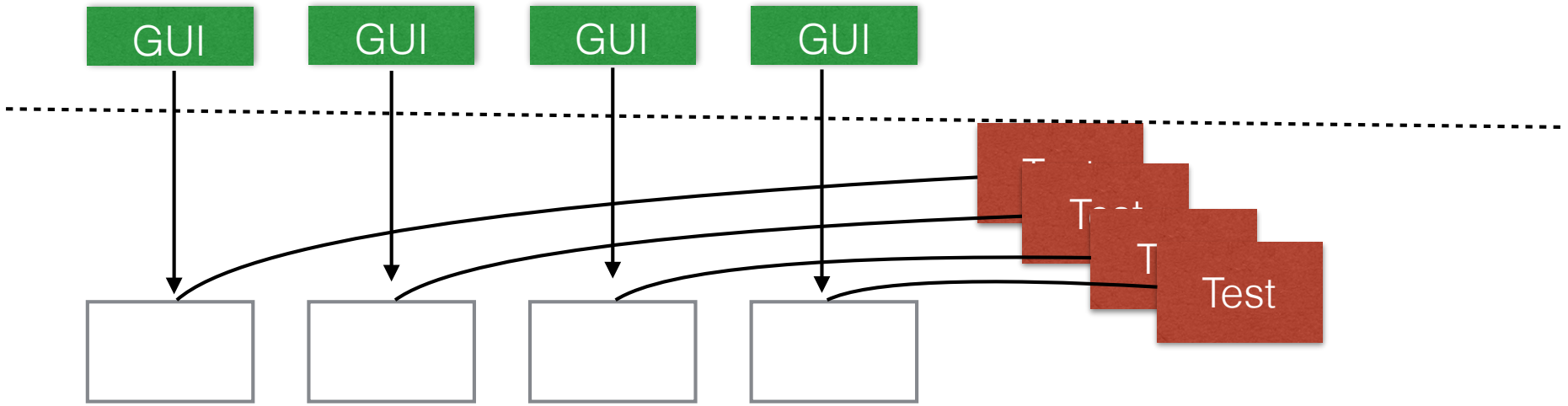
# Self shunt

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



# Testning av interaktiva tillämpningar - GUI

- Använd MVC - isolera Modellen från Interaktion (View)
  - Gör View-lagret så tunt som möjligt
  - Modellen kan testas med automatiserade tester.



# Hur mycket skall man testa?

- Testningsstrategi
  - Identifiera typiska fall
  - Testa randvärden (-1, 0, 1, maxint, null, length)
  - Testa all kod (test coverage)
  - XP: ”Test everything that could **possibly break**”
- Hur stor  del av koden är testkod?
  - Beror på tillämpningen
  - Runt hälften är en vanlig siffra

# Hierarkiska tester

- Enhetstester
  - Om en klass kan testas isolerat är det lättare att isolera vad som går fel.
  - Vid behov: koda upp ”mock objects” för att simulera interaktion med andra klasser (för komplexa, finns inte ännu, …)
- Modultester, subsystemtester …
  - Går utmärkt att automatisera med JUnit de också
  - Ersätter inte enhetstester

# Acceptanstester

- **Kunden**
  - Tänker ut ett antal testfall för varje ”Story”
  - ”Vad kan övertyga mig om att denna Story är implementerad?”
  - Kan vara riktiga testdata, t ex från tidigare verksamhet
- **Projektledning**
  - Acceptanstesterna mäter hur projektet framskrider
- **Utvecklarna**
  - Hjälper kunden implementera acceptanstesterna
  - Automatiserar så mycket som möjligt
  - Acceptanstesterna kan utvecklas parallellt med produktionskoden
  - Kan utvecklas av enskild programmerare

# Litteratur

- Kent Beck: Test-driven Development by example. Addison-Wesley. 2003
- Bill Wake's site: http://www.xp.123.com/xplor
  - The Test-First Stoplight
  - The Test/Code Cycle in XP

# Lab 3: Testning och Parprogrammering

- Förberedelser
  - Lab exercise. Test First using the JUint Testing framework
  - Using JUnit in EDA260
  - F04 - föreläsningsbilder (dessa)
  - 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 PairProgramming)
  - William C. Wake. Extreme programming Explored, Addison-Wesley, 2002. Utdrag: delar av kap. 1 (How Do You Write a Program?, p 2-10 + 20-21).
  - Chromatic: delar av Part II: p25-32 (Adopt Test-driven Development, Practice Pair Programming)

# Extra material om testning

- Ett enkelt kalkylblad som exempel

| | 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 tesfall [Bill Wake, xp123.com/xplor/xp0201]

- Basic functionality — storing values in cells
  - ThatCellsAreEmptyByDefault
  - ThatTextCellsAre Stored
  - ThatManyCellsExist
  - ThatNumericCellsAreIdentifiedAndStored
- SimpleFormulas
  - ThatWeHaveAccessToCellLiteralValuesForEditing
  - FormulaSpec
  - ConstantFormula
  - Add
- Dependencies
  - checkThatCellReferenceWork
  - checkThatCellChangesPropagate
  - …

# CheckThatCellsAreEmptyBy Deafult

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

# checkThatTextCellsAreStored

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

 sheet.put(theCell, "A string");
 assertEquals("cells are stored-1",
  "A string", sheet.get(theCell);

 sheet.put(theCell, "A different string");
 assertEquals("cells are stored-2",
  "A different string", sheet.get(theCell);

 sheet.put(theCell, "");
 assertEquals("cells are stored-3",
  "", sheet.get(theCell);
}
```

# checkThatCellReference-Works

```
@Test public void checkThatCellReferenceWorks(){
 Sheet sheet = new Sheet();

 sheet.put(”A1”, ”8”);
 sheet.put(”A2”, ”=A1”);
 assertEquals(”cell lookup”, ”8”, sheet.get(”A2”);
}
```