A) Final Test-liste for AlphaCiv:

Her er den test-liste vi har fundet frem til og lavet test cases for de kraven vi fandt på s. 458 og s.462: Samtlige krav til AlphaCiv er blevet skrevet tests til, og disse tests er blevet gennemført som del af TDD processen.

Players

Red is the first player in turn
Test: redShouldBeFirstInTurn()

After red, blue player is in turn
Test: afterRedBlueShouldBeInTurn()

After blue, red player is in turn
Test: afterBlueRedShouldBeInTurn()

World Layout

Ocean tile at (1,0)
Test: shouldHaveOceanAt1_0()

Plains everywhere else
Test: worldShouldHavePlainsEverywhereElse()

Mountain tile at (2,2)
Test: shouldHaveMountainAt2_2()

Hills tile at (0,1)
Test: shouldHaveHillAt0_1()

Units

Red settler at tile(4.3)
Test: shouldInitiallyBeRedArcherAt2x0()

Blue legion at tile(3,2)
Test: shouldInitiallyBeBlueLegionAt3x2()

Red archer at tile(2,0)
Test: shouldInitiallyBeRedSettlerAt4x3()

A unit has a max movement of one
Test: unitsCanMoveOneTile()

Units can at max move 1 tile per turn
Test: unitCanMove1tilePerTurn()

Red should not be able to move Blue's units

Test: redShouldNotBeAbleToMoveBlueUnits()

Blue should not be able to move Red's units
Test: blueshouldNotBeAbleToMoveRedUnits()

Units cannot move over mountain tile
Test: unitCannotMoveOverMountain()

Units cannot move over ocean tile
Test: unitCannotMoveOverOcean()

The unit at (2,0) (archer, red) cannot move to the ocean tile at (1,0)
Test: archerAt2_0CannotMoveToOceanTileAt1_0()

Units can be moved to an empty plains tile
Test: archerCanMoveToEmptyPlainTile()

A move should result in changed position and the unit remains the same type/owner, testing movement of archer from (2,0) to (3,0)
Test: typeAndOwnerRemainsUnchangedAfterMovement()

A unit cannot move outside the defined world ((0,15),(0,15))
Test: unitCanNotMoveOutsideTheBoundaries()

Combat

Red's unit attacks and destroys Blue's unit
Test: redUnitAttacksAndDestroysBlueUnit()

Blue's unit attacks and destroys Red's unit
Test: blueUnitAttacksAndDestroysRedUnit()

Red should not be able not attack own units
Test: redShouldNotBeAbleToAttackOwnUnit()

Cities

Choose what unit a city should produce
Test: chooseCityProduction

City population stays at 1
Test: cityPopulationRemainsAtOne()

City population should initially be 1
Test: cityPopulationStartsAtOne()

Cities produces 6 production after end of round
Test: citiesProduce6ProductionAfterEndOfEachRound

After 2 turns cities should have produced 12 production with nothing in production
Test: after2RoundsCitiesShouldHave12ProductionWithNothingInProduction

Setting city production tresuary to 100 it should return 100
Test: shouldReurn100WhenSetProductionTo100()

Setting city production tresuary to 200 it should return 2x100
Test shouldReturn200AfterSetof2x100()

Red has a city at tile(1,1)
Test: shouldHaveRedCityAt1_1()

Blue has a city at tile(4,1)
Test: shouldHaveBlueCityAt4_1()

Game can place a red archer once there is enough production in city at (1,1)
Test: cityCanProduceAnArcherAfter2Turns()

Game can place a blue legion once there is enough production in city at (4,1)
Test: cityCanProduceALegionAfter3Turns()

Aging

Game starts at 4000 bc
Test: gameStartsAt4000bc()

Game ages 100 years per turn
Test: shouldAdvanceTimeBy100AtEndOfFirstRound

After turn 5 the year is 3500 bc
Test: after5TurnsGameTineShouldBe3500BC()

Units regain the ability to move after each round: setting unit move to 1
Test: unitCanMove1tileAfterTurn()

Win

Red should be the winner in 3000BC
Test: redWinsIn3000BC

B) Outline over TDD iterations

Eksempel 1

Dette eksempel omhandler testen ”Blue legion at tile(3,2)”. Vi fulgte TDD rytmen, og valgte at afprøve princippet ”Assert First”. Derfor skrev vi de assertions der skulle passe til testen (og derved den del af spillets funktionalitet), for derefter at skrive metoden getUnitAt(), således at testen ville gennemføres.

```
@Test
public void shouldInitiallyBeBlueLegionAt3x2() {
    assertNotNull("3,2 should not be null",
            game.getUnitAt(new Position(3,2)));
    assertEquals("3,2 should be an blue legion",
            GameConstants.LEGION, //legion
            game.getUnitAt(new Position(3,2)).getTypeString());
    assertEquals("3,2 should be a blue unit",
            Player.BLUE,
            game.getUnitAt(new Position(3,2)).getOwner());
}
```

Testen fejlede selvfølgelig først, idet getUnitAt() i første omgang blot returnerede null. Første assertion ville derfor selvfølgelig fejle – Junit fejl: `"3,2 should not be null"`

Derefter fakede vi implementeringen, ved altid at have getUnit returnere et objekt af klassen Legion(). JUnit test blev kørt, og nu var det kun sidste assertion `"3,2 should be a blue unit"` der fejlede, idet den returnerede Legion tilhørte RED.

Vi rettede derefter fake-it delen, og lavede en if-clause, som checkede på positionen, og fik metoden til at returnere den korrekte Unit (Legion, med owner BLUE).

Testene gik nu igennem.

Eksempel 2

Vi har arbejdet frem efter, at der vælges en test ud fra listen. Derefter bruges nedenstående TDD rytme til at få en test fra papir til fake-it kode, og derefter til produktionskode. Som testes til det virker.

1. Tilføj en test
2. Kør alle testene og se den nye test fejle
3. Ændrer produktionskoden tilpasset den nye test
4. Kør alle testene igen og se dem alle virke
5. Lav refaktorisering for at fjerne duplikatkode.

Vi startede med at implementerer testcases der tjekker at rød fra start har en archer på Tile Position (2,0)
Mini skridt: Null test for at sikre vi ikke får null object tilbage.
Derefter tilføre til koden så en anonym klasse; unit returneres.

Et Mini skridt igen: Tester om enheden er en archer. Tilføjer at getUnitAt returner en anonym klasse med archer som typeString
Det sidste mini skridt:
Tester om enheden tilhøre RØD, tilføjer at getUnitAt returnere en anonym klasse med archer som typeString og player er rød. Ingen refaktorisering nødvendig.

Test der tjekker at der er en blå legion på 3,2
Igen som i Archer testcasen har vi et Null check og vi får Ok så der er ikke null
Så er turen kommet til enhedenstype er en Legion hvilket det er. Vi tilføjer returnering af anonym Unit klasse ved Position 2,3 med getTypeString = LEGION
vi ændrer getOwner til at returnere BLUE og vi tester igen og ser at det virker. Jubelen vil ingen ende tage.

Test der tjekker at der er en rød settler på 4,3
Som i de 2 forefgående test er Null checket det første er bliver testet. Unit er en Settler og vi tilføjer returnering af anonym Unit klasse ved Position 4,3 med getTypeString = SETTLER. Så er turen kommet til ejerforholdet. getOwner ændres til at returnere REDo g vi tester igen og ser at det virker.

Refaktorisering – UnitImpl klasse oprettes, anonyme klasser fjernes
Efter dette refaktoriserede vi da vi havde duplikering af kode. Denne refaktorisering resulterede i at vi brugte 2-d arrays og dermed flyttede skabelsen af enhederne op i konstruktøren.
Ydermere på grund af denne ændring kunne vi fjerne fake-it kode fra flere metoder og refaktoriserede konstruktøren således at skabelsen af map'et sker i en hjælpe-metode.

C) Reflection of the TDD benefits and liabilities

Fordelene ved at bruge TDD når man koder er, at først og fremmest giver det et glimrende overblik over hvad man laver og hvad der skal laves. Når man følger de første 5 TDD trin bliver man, normalt, ikke overvældet af kode eller mister overblikket. Skal man lave ændringer i koden er det også en overkommelig sag, i modsætning til mere traditionelle tilgange til softwareudvikling. Når man tager de små trin og holder fokus vil det i sidste ende føre til, at man bruger mindre tid på udvikling og vedligeholdelse. Da man tager mange små skridt medfører TDD-processen en lang række små succesoplevelser der hæver arbejdsmoralen.

Blandt ulemperne ved TDD er, at der reelt skal skrives noget mere kode, idet man først skal skrive (programmere) sine tests, og først derefter selve produktionskoden; produktet. Derudover tager det noget mere tid at sætte sig ind i arbejdsrytmen, da den er noget forskellig fra den ”traditionelle” tilgang til udvikling. Spørgsmålet er så, om det ikke bliver opvejet i det lange løb, set i forhold til vedligeholdelse og videreudvikling.