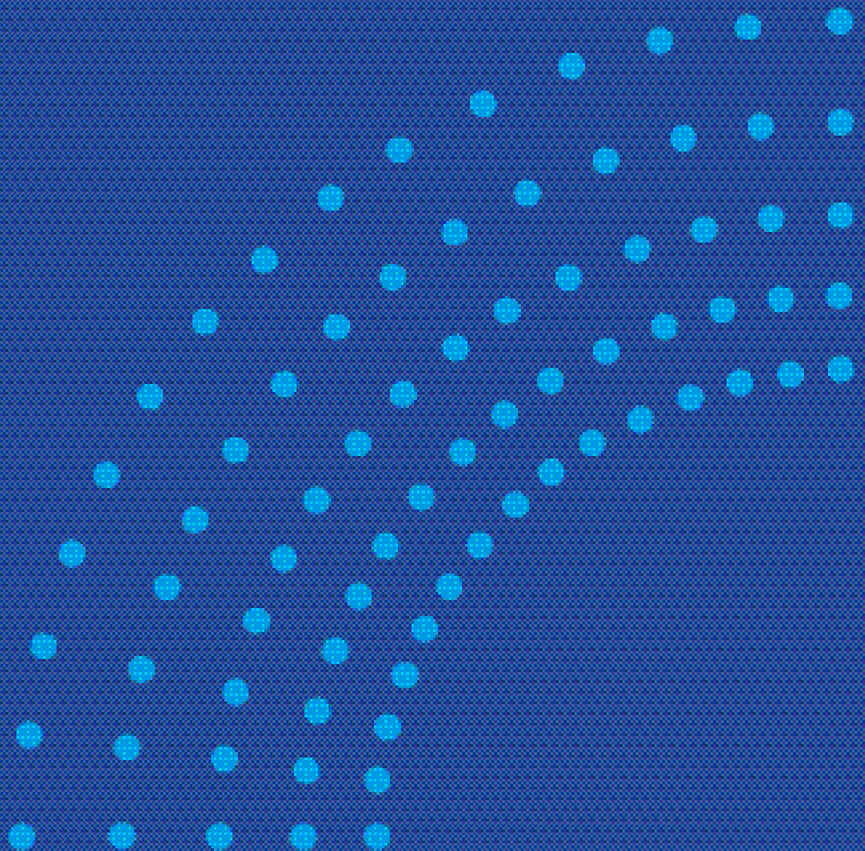
Examensarbete i Informatik

Portabilitet – en framtidssäkring?

Johan Lidqvist & Peter Sundström

Göteborg, Sweden 2007



Portabilitet - en framtidssäkring?
Portabel källkod som medel för ökad kvalitét
JOHAN LIDQUIST
PETER SUNDSTRÖM

©JOHAN LIDQUIST & PETER SUNDSTRÖM, 2007

Report no 2007:64
ISSN: 1651-4769
Department of Applied Information Technology
IT University of Göteborg
Göteborg University and Chalmers University of Technology
Box 8718
SE – 402 75 Göteborg
Sweden
Telephone + 46 (0)31-772 4895

RAPPORT NR. 2007/64

Portabilitet - en framtidssäkring?

Portabel källkod som medel för ökad kvalitét

Johan Lidquist
Peter Sundström

Department of Some Subject or Applied Information Technology
IT UNIVERSITY OF GÖTEBORG
GÖTEBORG UNIVERSITY AND CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2007

Portabilitet - en framtidssäkring?
Portabel källkod som medel för ökad kvalitét
JOHAN LIDQUIST
PETER SUNDSTRÖM
Department of Applied Information Technology
IT University of Göteborg
Göteborg University and Chalmers University of Technology

Sammanfattning
I en värld där informationsteknologi spelar en allt större roll i våra liv, blir konkurrensen om användarna på marknaden allt tuffare. I takt med att GNU/Linux och Apples operativsystem ökar i användarantal har det blivit intressant för utvecklare att kunna erbjuda sina produkter på ett flertal plattformar. Genom att skriva portabel källkod kan man med liten eller minimal revision erbjuda mjukvara på flera plattformar. Vi har genomfört en studie där vi deltagit i ett projekt som syftat till att göra en applikation tillgänglig på flera plattformar. Genom aktionsforskning har vi på ett aktivt sätt deltagit i ett portningsprojekt för insamlandet av empiri. Projektet resulterade i att ett flertal problem identifierades gällande portabilitet. Resultatet visar att portabilitetsförbättrande programmering aldrig är bortkastad tid, även i de fall då programmets miljö aldrig kommer att förändras. De initiala besluten vid design av en applikation är kritiska för dess framtida möjlighet att portas.

Nyckelord: Portabilitet, migrering, plattformsoberoende, Linux, Windows, programmering

Förord

Vill vill rikta ett stort tack till vår handledare Lennart Petersson som på ett alltid lika positivt och motiverande sätt bidragit med råd när vi behövt dem som mest.

Vi vill även tacka samtliga anställda på företaget där experimentet utförts för den tid de tagit sig att besvara våra frågor i tid och otid. Extra stort tack till Björn Olsson som trots späckat schema tagit sig tid för oss och givit oss resurser att kunna slutföra experimentet.

Innehåll

1 Introduktion **7**
1.1 Problemområde 8
1.2 Syfte och frågeställning 8
1.3 Målgrupp 8
1.4 Avgränsningar 9
1.5 Disposition 9

2 Metod **10**
2.1 Vetenskapsteori och metodologi 10
2.1.1 Aktionsforskning 10
2.1.2 För och –nackdelar 11
2.1.3 Inriktningar inom aktionsforskning 12
2.1.4 Beslutet 15
2.2 Förstudie 15
2.2.1 Granskning av kod och befintlig programvara 15
2.2.2 Litteraturstudie 15
2.2.3 Val av API 16
2.3 Praktiskt tillvägagångssätt 16
2.3.1 Insamlande av empiri 16
2.3.2 Sammanställning, tolkning och analys av empiri . . . 17

3 Teori **18**
3.1 Portabilitet 18
3.1.1 Programmeringsspråk 20
3.1.2 C 22
3.1.3 C++ 23
3.1.4 Java 23
3.1.5 C# 24
3.1.6 Funktionsbibliotek 25
3.1.7 Programstruktur och selektiv kompilering 28
3.1.8 Abstraktion 32
3.1.9 Datautbyte 33
3.1.10 Byte order 34
3.1.11 Uppdaterings— och versionskonflikter 37
3.1.12 Internationalisering 38
3.2 Reverse Engineering och Re-Engineering 39
3.3 WO–koncept 39
3.4 Portningsprocessen 40

4 Resultat **46**
4.1 Diskussion 46
4.1.1 Designkriterier 46
4.1.2 Val av programmeringsspråk 47
4.1.3 Val av funktionsbibliotek 47
4.1.4 Flera källkodsbaser 48
4.1.5 Internationalisering 49

5 Slutsats **50**

6 Referenser **51**
6.1 Litteratur 51
6.2 Artiklar 52
6.3 Webbreferenser 52

1 Introduktion

Det finns ett flertal olika operativsystem och hårdvaruplattformar ute på marknaden. Vissa liknar varandra mer än andra. Det finns dock skillnader i systemen vilket gör att program oftast inte skrivs för flera olika operativsystem med samma källkodsbas. Detta gör att många mjukvaruutvecklare väljer ett system att utveckla mot. Deras val kan bero på olika faktorer, tex systemets användarbas eller funktioner.

Hur når man som utvecklare av mjukvara en så stor målgrupp som möjligt? Ett sätt att öka användarbasen är att göra programmen tillgängliga för flera olika system och plattformar. I vissa fall krävs få eller inga förändringar i källkoden för att göra detta möjligt. I andra fall är det inte alls lika lätt. Hur mycket jobb som krävs beror på varje individuellt projekt, många faktorer spelar in.

Ett vanligt sätt att gå tillväga för att göra program tillgängliga på flera olika plattformar, är att skriva programmen i ett så kallat "plattformsoberoende" programmeringsspråk. Plattformsoberoende i detta fallet är möjligt genom att en virtuell maskin tolkar och exekverar programmet. Den virtuella maskinen är portad till olika plattformar för att kunna exekvera programmet på den plattform den körs på. Programmeringspråk i denna kategorin inkluderar Java och C#. Att skriva program i språk som dessa, löser många problem gällande migrering då samma kod kan köras på olika plattformar utan modifiering. Så länge den virtuella maskinen finns tillgänglig på plattformen kan koden exekveras.

Det är dock inte alltid möjligt att utnyttja ett plattformsoberoende språk för sin mjukvarulösning. I vissa fall måste man tex använda sig av ett språk på lägre nivå. I dessa fall krävs det att man skriver om koden så att den kan komplileras och exekveras på andra plattformar då den är mer bunden till maskinvaran än språk på högre nivå. Man behöver porta källkoden.

Att porta ett program innebär att man skriver om och anpassar det för en annan plattform. Genom att porta ett program gör man det tillgängligt för fler användare. Detta kan skapa konkurrensfördelar genom en större användarbas.

Genom att använda språk, bibliotek och kompilatorer som finns på flera olika plattformar kan man göra sin kod lättare att porta. Man kallar detta att göra mjukvaran portabel.

1.1 Problemområde

Det operativsystem som har flest användare idag är Microsoft Windows-familjen[1]. Enligt siffror som baseras på internetanvändares val av operativsystem, finner vi att Windows-familjen andel är ca 90% av marknaden. Windows har dock några konkurrenter på marknaden som Mac OSX och olika Linux distributioner. Mac OS X och Linux har cirka 3.5% av användarna på internet vardera. I takt med att dessa operativsystem får en större andel av användarna blir producenter av mjukvara mer intresserade av att kunna erbjuda sin mjukvara på dessa plattformar. Detta kan göras möjligt på olika sätt:

- Skriva sina program i "plattformsoberoende" programmeringsspråk
- Skriva om källkoden specifikt för en annan plattform, så kallad portning.

För att kunna utvärdera portningsstrategier och teorier har vi utfört ett experiment där vi portat ett utvecklingsprojekt mellan plattformen Windows NT och Gentoo Linux. Uppsatsen går igenom våra beslut gällande val och användning av programmeringsspråk och bibliotek. Uppsatsen tar upp varför val har gjorts och resultaten rörande portningsprojektet vi utfört med fokus på hur problem kan undvikas.

1.2 Syfte och frågeställning

Syftet med denna uppsats är att avgöra vilka faktorer som är kritiska gällande portning av källkod för ett framgångsrikt resultat. Detta gäller både vid utveckling av ny och befintlig mjukvara som skall portas till andra plattformar.

Forskningsfrågan vi utgått ifrån är:

Vilka är de kritiska framgångsfaktorerna vid portning av mjukvara?

1.3 Målgrupp

Denna uppsats riktar sig till de läsare som utvecklar mjukvara som kan komma att portas i framtiden. Läsare som designar system och skriver mjukvara bör finna denna uppsats intressant då den ger rekommendationer om hur man skall designa och skriva mjukvara för att undvika framtida problem med källkoden.

[1]W3Schools Online Web Tutorials

1.4 Avgränsningar

Vi har valt att behandla problem gällande migrering av källkod mellan olika plattformar ur ett generellt programmeringsperspektiv för portabel källkod. Vi har valt att inte belysa systemspecifika skillnader mellan operativsystem, då uppsatsen fokuserar på generella aspekter gällande portabilitet. Detta innebär att problem med användarrättigheter, filsystemsstruktur och övriga konflikter som beror på operativsystemets design inte tas upp.

1.5 Disposition

Vi ger läsaren en överblick om de olika programmeringsspråk och funktionsbibliotek vi använt oss av, eller finner av intresse att belysa. Uppsatsen tar upp vad utvecklare bör tänka på för att skriva portabel källkod.

2 Metod

Detta kapitel beskriver de metoder vi använt oss av för att söka svar på vår frågeställning. Avsikten med kapitlet är att läsaren skall få inblick i hur vi arbetat för att nå vårt resultat, samt att kontrollera det utifrån insamlad empiri. Då arbetet är av teknisk och praktisk karaktär, har vi fokuserat på att finna tekniska lösningar på reella problem.

2.1 Vetenskapsteori och metodologi

Inom informatiken finns det idag ett antal vedertagna forskningsmetoder att tillämpa. I vårt val av metodologi, har vi utgått från några i förväg bestämda premisser. Frågeställningen är riktad mot ett konkret mål, vilket är att finna faktorer för framgång vid praktiskt arbete[2]. Aktionsforskning som metod för insamling av empiri ansåg vi lämpa sig väl för vår studie, då ett av våra främsta önskemål var att vi på ett aktivt sätt skulle få delta i projektet. Nedan beskriver vi de metoder och tillvägagångssätt vi använt oss av för att nå vårt resultat.

2.1.1 Aktionsforskning

Begreppet aktionsforskning är ett sätt att använda vetenskapliga metoder för att lösa praktiska problem på ett sätt som bidrar till socialvetenskaplig teori och kunskap[3]. Inom begreppet aktionsforskning finns det ett antal olika inriktningar. Det som gemensamt kallas för aktionsforskning bygger på en mängd olika förhållningssätt, metoder och värderingar med rötter i olika traditioner[4]. Aktionsforskning har enligt vissa författare länge varit ett otydligt begrepp,[5] vilket medfört att forskare säger sig ägna sig åt "interaktiv", "handlingsinriktad", "particitativ' eller "praktikorienterad" forskning. Det är svårt att finna enighet ens hos aktionsforskarna själva om vad som utgör själva kärnan i aktionsforskning[6]. Aktionsforskning bör istället betraktas som ett samlingsnamn. Den samlande termen aktionsforskning är ett familjenamn för denna sortens forskning[7]. Aktionsforskning låter sig inte enkelt beskrivas som en metod eller teori utan skall snarare betraktas som en strategi för forskningen. För att definiera aktionsforskningen ytterligare är det nödvändigt att beskriva några ur aktionsforskningens karaktäristiska drag.

[2] Rapoport, R.N, *Three Dilemmas in Action Research*, 1970

[3] Ibid

[4] Ibid

[5] Reason, Peter & Bradbury, Hilary, *Handbook of action research: Participative inquiry and practice*, 2001

[6] Hansson, Agneta, *Praktiskt taget: Aktionsforskning som teori och praktik*, 2003

[7] Reason, Peter & Bradbury, Hilary, *Handbook of action research: Participative inquiry and practice*, 2001

- **Praktisk inriktning** – Aktionsforskaren tar sig an “verkliga” problem och frågeställningar hämtade ur verkliga situationer och fysiska organisationer[8].
- **Förändring** – ses som en del av forskningen. Förändringen används som ett verktyg för att lösa praktiska problem och för att öka förståelsen för företeelser i dess ursprungliga sammanhang. Själva målsättningen är att man skall sträva efter att förändra forskningssubjektet för att uppnå förståelse. Syftet är att man som forskare lär sig mer genom att aktivt delta i en förändringsprocess än man hade gjort som passiv observatör[9].
- **Cyklisk process** – Forskning som innefattar en återkopplingsprocess, ger möjlighet till förändringar som senare kan implementeras och evalueras som utgångspunkt för fortsatta studier[10].
- **Deltagande** – Deltagarna står i centrum i forskningsprocessen. Deras aktiva deltagande bygger på samarbete, ömsesidigt lärande samt gemensam kompetensutveckling[11].
- **Det hermeneutiska kunskapsidealet** – Aktionsforskning har i grunden ett emancipatoriskt och/eller kunskapsteoretiskt hermeneutiskt kunskapsideal[12].
- **Värdegemenskap hos praktiker och forskare** – Tankar om värdemässiga grunder är nära knutet till det emancipatoriska kunskapsidealet[13].
- **Helhetsförstående av problem** – Aktionsforskning skall leda till praktisk problemlösning och teoriutveckling[14].

2.1.2 För och –nackdelar

Inför valet av forskningsmetod är det viktigt att man är väl medveten om de för– och nackdelar metoden medför. Aktionsforskning har precis som andra forskningsmetoder sina för– och nackdelar. Aktionsforskning är en omtalad forskningsmetod, dels hyllad och älskad av många, men också bestridd och ifrågasatt.

[8]Hansson, Agneta, *Praktiskt taget: Aktionsforskning som teori och praktik*, 2003
[9]Ibid
[10]Ibid
[11]Ibid
[12]Denscomb, Martyn, *Forskningshandboken : för småskaliga forskningsprojekt inom samhällsvetenskaperna*, 2000
[13]Hansson, Agneta, *Praktiskt taget: Aktionsforskning som teori och praktik*, 2003
[14]Ibid

Genom aktionsforskningens historia har ett antal dilemman påtalats. Utifrån forskningsperspektivet så talas det om två uppenbara dilemman. Det första handlar om valet mellan vetenskaplig kontra praktisk relevans[15]. Det andra handlar om vem som tar initiativet till att föra fram problemet, det vill säga frågan om demokrati och kontroll i forskningsprocessen[16]. Vidare har aktionsforskningen kritiserats för att forskarna involveras i för hög grad i förändringsarbetet, vilket kan leda till brister när det gäller teorianknytning, långsiktighet och kritisk distans[17]. En utmaning för aktionsforskaren är att visa att man både kan leva upp till de krav på skriftlig kommunikation av sina resultat som forskarsamhället formulerat och samtidigt behålla sin relevans för praktikerna[18].

Förespråkarna för aktionsforskning skriver att man genom metoden får dialoger mellan olika perspektiv på deltagarorienterad och förändringsinriktad forskning, vilket sprider nytt ljus över forskningsfältet[19]. Som tidigare nämnt har forskaren fördelen uppnå en djupare förståelse som aktiv deltagare än vad denne hade uppnått som observatör.

2.1.3 Inriktningar inom aktionsforskning

Inom området aktionsforskning finns det ett flertal inriktningar. Dessa inriktningar är ett resultat av de olika mål som forskare haft vid tillämpandet av aktionsforskning. Beroende på syfte, mål och verkningsområde har forskare använt aktionsforskningen och satt sin egen prägel på den för att den skall passa just dem.

Aktionsinriktad forskning brukar härledas till pragmatismen och 1900-talets USA. I pragmatismen definieras kunskapen i nyttotermer och det är praktiska problem som är utgångspunkten för sökandet av kunskap och reflektion[20]. Pragmatismen förenade teori och praktik i en kunskapsprocess, vilket kännetecknar den aktionsinriktade forskning som idag utövas under samlingsnamnet aktionsforskning[21]. Beroende på målsättning och syfte har aktionsinriktad forskning antagit olika skepnader i form av inriktningar. Klassisk aktionsforskning är den ursprungliga formen av aktionsinriktad

[15]Rapoport, R.N, *Three Dilemmas in Action Research*, Human Relations, 1970

[16]Lundberg, Bertil & Starrin, Bengt, *Participatory research: Tradition, theory and practice*, 2001

[17]Svensson, Lennart, *Interaktiv forskning - för utveckling av teori och praktik*, 2002

[18]Lundberg, Bertil & Starrin, Bengt, *Participatory research: Tradition, theory and practice*, 2001

[19]Reason, Peter & Bradbury, Hilary, *Handbook of action research: Participative inquiry and practice*, 2001

[20]Gustavsson, Bernt, *Bildning i vår tid: Om bildningens möjligheter och villkor i det moderna samhället*, 1996

[21]Reason, Peter & Bradbury, Hilary, *Handbook of action research: Participative inquiry and practice*, 2001

forskning. Klassisk aktionsforskning kännetecknas av ett aktivt deltagande i förändringsprocessen[22]. Den klassiska aktionsforskningen utövas med syfte att förändra för att utvinna kunskap ur själva förändringsprocessen.

Ur den klassiska aktionsforskningen föddes ett annat sätt att bedriva aktionsinriktad forskning; Participatory Research. Participatory Research är en annan form av aktionsinriktad forskning vilken är något av en "motpol" till klassisk aktionsforskning. Den klassiska aktionsforskningen karaktäriseras av konsensus och konfliktundvikande, medan Participatory Research gör det motsatta. Participatory Research är en ideologisk och politiskt orienterad forskningstradition. Dessa motsatta traditioner kallas norra traditionen (Klassisk aktionsforskning) och den södra traditionen (Participatory Research)[23]. Den norra traditionen arbetar med det rådande samhällssystemet, medan det södra arbetar mot det rådande samhällssystemet.

Vidare ur Participatory Research föddes Participatory Action Research (PAR). Begreppet PAR beskrivs som en process där deltagandet är fundamentalt för handling som syftar till förändring. Det är en strategi för att utveckla både vetenskap och praktik. Här lämnas de ideologiska och politiska värderingar utanför för att skapa "värdeneutrala" resultat[24].

Tanken är att man genom PAR skall undvika den extraherande forskning som kan uppstå när forskare studerar sitt forskningssubjekt i syfte att finna svar på forskningsfrågor för att sedan presentera sitt resultat för andra akademiker/forskare. Detta innebär att kunskapen endast kommer att spridas indirekt till människorna inom problemområdet via akademiska kanaler. När forskaren/forskarna arbetar sida vid sida med praktikerna så forceras kommunikation och integration[25]. Figuren nedan är en modell av Elden & Levin[26] över hur PAR tillämpas i Skandinavien och visar hur kommunikationen mellanforskare och praktiker sker.

---

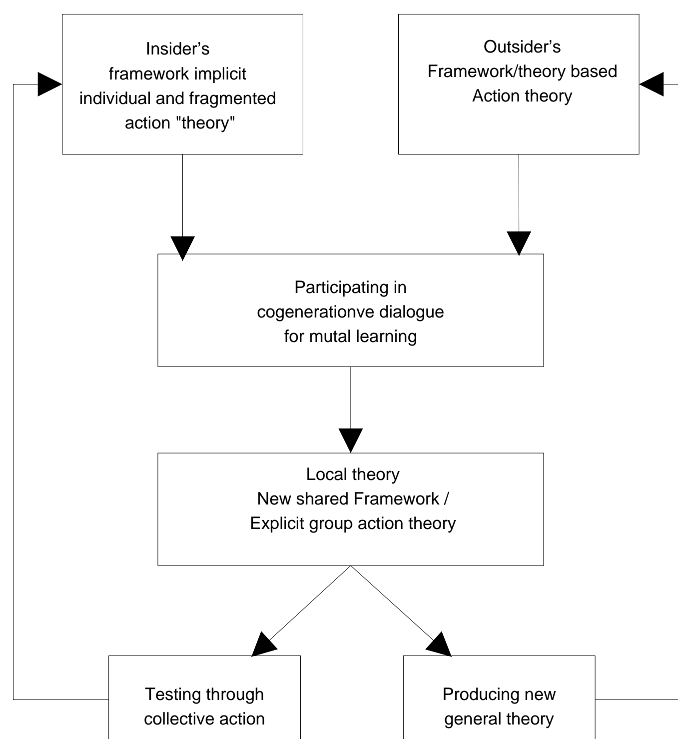
[22]Hansson, Agneta, *Praktiskt taget: Aktionsforskning som teori och praktik*, 2003

[23]Brown, D. & Tandon, R, *Ideology and political economy in inquiry: Action research and participatory research*

[24]Whyte, William Foote, *Participatory Action Research: Through Practice to Science in Social Research*, 1991

[25]Wadsworth, Yoland, *What is Participatory Action Research?*, 1998

[26]Hansson, Agneta, *Praktiskt taget: Aktionsforskning som teori och praktik*, 2003



Modellen visar forskarna som “outsiders” och deras integration med praktikerna som “insiders”. Praktikerna ses inte som datakällor, utan hjälper aktivt till i utvecklingen och medbestämmandet i varje del av forskningsprocessen[27]. Båda parterna förs samman i en dialog som i figuren kallas för “mutual learning”. Elden & Levin menar att denna dialog är nödvändig i varje form av frigörande lärande. I dialogen utvecklas en teori som de kallar för en “lokal teori”. Denna teorin förblir dock inte lokal, utan återkopplar till de båda fälten, som sedan utvärderas och prövas praktiskt i en iterativ process.

Trots de olikheter som finns i de olika forskningsmetoderna utesluter inte den ena den andra. Det är möjligt, och inte alls ovanligt att strategierna inom forskningsfamiljen vävs in i varandra för att tillfredställa de syften som finns i den forskning som tillämpningen avser[28].

Det finns ett antal ytterligare inriktningar inom aktionsforskning. Med anledning av att vi valt att tillämpa PAR kommer vi bara behandla de mest relevanta och angränsande områdena inom aktionsinriktad forskning som rör just PAR.

[27]Hansson, Agneta, *Praktiskt taget: Aktionsforskning som teori och praktik*, 2003
[28]Ibid

2.1.4 Beslutet

Då aktionsforskning syftar till att använda vetenskapliga metoder för att lösa praktiska problem, anser vi den lämplig för att nå våra mål. Vårt beslut grundar sig på de fördelar och möjligheter som metoden ger oss att evaluera olika hypoteser under arbetets gång för att sedan utvärdera dessa i ett verkligt kontext[29]. Vi tror att aktionsforskning kommer ge oss möjlighet att pröva olika portningsstrategier, med målet att finna den optimala strategin och svaret på vår frågeställning. Bland de olika inriktningar som finns inom aktionsforskning har vi beslutat oss för att tillämpa Participatory Action Research (PAR).

2.2 Förstudie

Det praktiska arbetet föregicks av förberedande studier. Under förstudien fokuserade vi på att finna eventuella problem inför portningen. Förstudien bestod av litteraturstudier, granskning av kod samt befintlig programvara samt analys av API:er. I förstudien fattades viktiga beslut angående:

- Val av programmeringsspråk
- Val av API
- Val av portningsstrategi
- Val av kodnotation och utvecklingsmiljö

2.2.1 Granskning av kod och befintlig programvara

Granskningen av befintlig kod var utgångspunkten under förstudien med anledning av att vi först här fick djupare insikt i vad portningen skulle omfatta. I detta stadiet började vi först med att studera relevanta dokument över systemets struktur så som; Klassdiagram, strukturdiagram och sekvensdiagram. Nästa steg i fasen var att granska källkoden för att finna eventuella hinder för portningen av programmet. Potentiella hinder för fortsatt portning är faktorer som binder applikationen till en specifik plattform genom unika systemrutiner eller API:er låsta till plattformen.

2.2.2 Litteraturstudie

I litteraturstudien studerade vi tidigare portningsprojekt av liknande karaktär så som GTK-projektet. Vidare studerade vi de olika kryptografiska algoritmer som implementationen innefattade. Eftersom de kryperingsalgoritmer vårt projekt innefattade var så kallade öppna standarder, fanns all teknisk fakta tillgänglig online, främst genom Wikipedia, men även i kryptografisk litteratur.

[29]Susman, G. – Action research: A Sociotechnical Perspective, 1983

2.2.3 Val av API

Det befintliga systemet som portningen avser, är skrivet i en blandning av programmeringsspråk. Det mest förekommande programmeringsspråket var dock C#. Av portningsstrategiska skäl beslutade vi oss för att använda oss av ett portabelt programmeringsspråk. Granskningen av systemets struktur och källkod, tillsammans med vissa önskemål från företagets sida, resulterade i en kravspecifikation för ett kryptografiskt API baserat på C eller C++. De faktorer som var avgörande vid valet av API var följande:

- Vilken typ av licens API:et var licenserad under
- Eventuella kostnader
- Användarbas och support
- Framtid och utveckling

2.3 Praktiskt tillvägagångssätt

Då uppsatsen ingår som ett avslutande moment i Systemvetarprogrammet och syftar till att praktiskt tillämpa de kunskaper som undervisats under utbildningens gång, valde vi att i största möjliga mån utöva praktiskt arbete. Genomförandet av detta moment bestod av att identifiera ett problemområde för att sedan studera det och nå ett resultat. För att nå fram till resultatet har vi följt ett antal steg:

- Val av undersökningsområde
- Identifiering av problemområde/frågeställning
- Förstudie
- Praktiskt deltagande i experiment
- Analys av empiri
- Resultat

2.3.1 Insamlande av empiri

Det experiment som utfördes under cirka tre månader ägde rum hos ett IT-säkerhetsföretag i Göteborg. Experimentet genomfördes i en grupp om 4 personer, oss själva inkluderat, på plats hos företaget. Ett problemområde hade identifierats hos företaget redan innan vi kontaktade dem, och vi enades oss om att tillsammans utföra experimentet under den givna tidsramen. Projektet gick ut på att utöka användarbasen för en prototyp av en produkt,

som företaget utvecklat på en Windowsplattform. Projektets mål var att göra den tillgänglig på ett flertal olika plattformar. Då produkten är omfattande, bestämde vi oss för att endast realisera ett antal moduler tillhörande applikationen. Projektet utfördes i grupp där vi arbetade efter Participatory Action Research–modellen. Metoden gjorde att vi kunde deltaga aktivt i projektet samtidigt som vi samlade in empiri. Utövandet av metoden gjorde att vi arbetade som praktiker i grupp parallellt med att vi studerade problemområdet som akademiker. Fältanteckningar fördes kontinuerligt i takt med att viktiga observationer gjordes och avgörande beslut fattades.

2.3.2 Sammanställning, tolkning och analys av empiri

När projektet nått sitt slut påbörjades processen med att sammanställa den empiri som samlats in under projektets gång. Detta innefattade sammanställning av fältanteckningar som gjorts löpande under projektets gång. Sammanställningen gick ut på att identifiera praktiska problem med projektet. Då alla praktiska problem rörande projektet inte nödvändigtvis var av relevans för forskningsfrågan, gjordes en utgallring av problem som inte rörde portabilitet. När alla relevanta problem var identifierade gjordes en kategorisering av problemen. De problem som var av samma karraktär, dvs de problem som hade en liknande lösning, sammanfördes under en och samma kategori för att behandlas som ett och samma problem.

I analysfasen analyserades problemen vi identifierat för att fastställa om de var specifika problem för projektet eller om problemen kunde härledas till portabilitet.

3 Teori

3.1 Portabilitet

Portning
Portning är den process som syftar till att modifiera en existerande applikation så att den kan exekvera på en ny plattform[30]. Beroende på vilken typ av applikation som skall portas och vilken typ av portning det gäller kommer arbetsbördan för att fullfölja portningen att variera (se tabell Huvudkategorier inom portning). Denna varians sträcker sig från att endast kompilera om källkoden för den nya plattformen till att designa om hela applikationen och skriva om stora stycken av koden för att skapa passform för den nya plattformen. Det är dock inte enbart tekniska omständigheter som avgör hur mycket arbete som behövs läggas ner på att porta en applikation. Hur en applikation ursprungligen designats är en avgörande faktor för hur tung arbetsbördan blir.

Vid migrering från en miljö till en annan, kan portningssubjektet ha vissa beroenden som medför ytterligare migrering. Olika tekniska lösningar kan innefatta flera olika portningskategorier och därför även involvera flera olika portningsprojekt[31]. Med vårt projekt som utgångspunkt kan vi exemplifiera detta. Då vårt projekt var att porta en applikation med kryptografiska rutiner från ett Window– till Linux–system, krävdes det att vi hittade ett motsvarande kryptografiskt bibliotek på målplattformen. Valet av programmeringsspråk får konsekvenser för val av funktionsbibliotek eftersom tillgängliga API:er beror på språket; Vi var tvugna att hitta ett lämpligt bibliotek som passade våra behov efter vårt valda programmeringsspråk. Utvecklingsverktyget var ursprungligen Microsoft Visual Studio, vilket var otillgängligt under Linux. Eftersom applikationen skulle flyttas till en ny plattform där det ursprungliga programmeringsspråket inte fungerade önskvärt, innebar det även att migrering av ramverk, bibliotek, språk och utvecklingsverktyg var nödvändigt.

[30]Porting=DevelopmentTechniques.pdf
[31]Ibid

Huvudkategorier inom portning[32]	**Exempel**
Operativsystem	Mac till Windows
OS–versioner	Mac OS 8/9 till OS X
Databaser	MSSQL till Oracle
Språk	C++ till Java
Ramverk och bibliotek	Borland till MFC
Teknologier	COM till CORBA
Utvecklingsverktyg	VC++ till CodeWarrior
Webbplattformar	IIS/ASP till WebObjects

Portabilitet

Följande sektion behandlar problematik inom olika områden gällande portabilitet. En stor del av de fakta som presenteras är hämtad från boken “The Practice of Programming”, skriven av Kerninghan & Pike.

Det innebär mycket jobb att skriva ett program som fungerar korrekt, effektivt och tillförlitligt. Så när ett program skall flyttas till en ny miljö, ligger det i utvecklarens intresse att minimalt med ändringar utförs under förändringsfasen. Det idealiska vore naturligtvis om man slapp ändra i koden överhuvudtaget. Detta ideal kallas portabilitet. Ju mindre revision av programmet vid flytt, desto mer portabelt är programmet[33].

Så varför ska vi bry oss om portabilitet? Om ett program utvecklas för en viss miljö under specifika villkor, vad finns det då för mening att lägga energi på att utveckla portabel kod? Så gott som alla framgångsrika program, nästan per definition, kommer att användas i oväntade sammanhang och miljöer[34]. Även i fall då detta inte gäller, kan kompatibilitet med programmets nuvarande miljö ändras av andra naturliga skäl. Det kan vara allt från uppgradering av operativsystem till byte av kompilator eller hårdvara. Ju mindre programmet beror på hårdvaran eller plattformen det kör på, desto mindre sannolikhet att programmet beter sig annorlunda i den nya miljön utan revidering.

Att tillämpa portabilitetsförbättrande programmering är aldrig bortkastad tid även om programmets miljö aldrig kommer att ändras under dess levnadstid. Portabla program är bättre program[35]. Den tid och kraft som läggs ner på att göra ett program portabelt leder även till att programmet blir bättre designat, bättre konstruerat och mer testat. Det tillvägagångssätt

[33] Kernighan, Brian W. & Pike, Rob, *The Practice of programming*, 1999
[34] Ibid
[35] Ibid

som portabel programmering innebär, är nära relaterat till generell god programmeringskutym[36]. Att uppnå fullt portabel kod brukar anses som en omöjlighet; *There is no such thing as an absolute portable program, only a program that hasn't been tried in enough evironments*[37]. I nedanstående stycke ges beskrivningar av kritiska områden som noga bör uppmärksammas vid utvecklandet av portabel programvara.

3.1.1 Programmeringsspråk

Denna sektion behandlar de olika programmeringsspråk som har varit av intresse för oss. Vi kommer att förklara hur de uppkom, vad som skiljer dem åt, samt illustrera skillnader i syntax och implementation. Detta för att kunna ge en tydligare bild av möjliga problem vid migrering mellan olika språk.

Standardiserade programmeringsspråk

För att skriva portabel kod krävs det att man använder ett högnivå-språk[38]. Det är viktigt att man skriver kod som följer språkets standard, där det finns sådan. Standardisering av programmeringsspråk hjälper utvecklare att skriva kompilatorer som fungerar på olika plattformar med samma källkod. Kompilatorer för ett standardiserat språk skall kunna tolka samma källkod och skapa maskininstruktioner, oavsett den plattform och operativsystem som ligger i botten. Det är inte alltid tillräckligt definierat hur kompilatorn skall översätta och tolka källkoden. Få språk har bara en implementation av sig, då språket kan ändras mellan versioner[39]. Det är heller inte ovanligt att versioner av språk skiljer sig på olika operativsystem. Kompilatorer kan även ha olika utvecklare bakom sig, vilket innebär att deras tolkning av språket kan skilja sig åt.

Så varför är inte standarderna av strikt definition? Ibland är inte standarder tillräckligt definierade av olika skäl. Där standarder inte förklarar beteende eller egenskaper hos språket tillräckligt, får utvecklare implementera dessa efter egen tolkning. Ju mer lös definitionen är, desto mer kommer kompilatorernas tolkning skilja sig gällande denna aspekt. Varför är man då inte bättre på att beskriva definitioner i vissa lägen? I vissa fall har man gjort detta medvetet för att underlätta för utvecklarna av kompilatorerna. Genom att skriva en mindre specifik definition av en egenskap, kan utvecklare ofta implementera den effektivare utan att vara bunden av hårda restriktioner[40].

[36]Kernighan, Brian W. & Pike, Rob, *The Practice of programming*, 1999
[37]Ibid
[38]Ibid
[39]Ibid
[40]Ibid

Detta gäller inte minst då skillnader i hårdvara samt tekniska problem gör det svårt att enas om en standardiserad definition som fungerar effektivt på flera plattformar. Politik kan även spela in, då oenigheter mellan inblandade parter kan leda till kompromisser i definitioner av egenskaper och funktioner. Man får inte heller glömma att programmeringsspråk och dess kompilatorer är komplexa, vilket gör att det kommer att vara tolkningsfel av standarder och buggar i implementationen[41].

En standard för ett språk utvecklas normalt inte innan konflikter av implementationer mellan utvecklare uppstår. Konflikterna måste även vara tillräckligt stor och vara av tillräcklig vikt för att rättfärdiga en kostsam och tidskrävande standardiseringsprocess. Då många språk har flera upphovsmän är det viktigt att låta standardisera språket då det leder till färre konflikter. Färre konflikter leder i regel till att utvecklingen av språket slipper stagneras under tiden. Standardisering är en kostsam process, men väl värt det för många språk och utvecklare. Utan standard blir det svårt att uppnå portabilitet hos ett språk, då det är svårt att få utvecklare att arbeta mot samma mål.

Standarder kan ge ett intryck av hårda specifikationer, men man får inte glömma att de aldrig kan definiera ett programmeringsspråk fullt ut[42]. Olika implementationer är giltliga, men skapar inkompatibilitet då kompilatorer är skrivna av personer som tolkar de aktuella standarderna på eget sätt. Detta är svårt att komma ifrån, därför är det viktigt att omrevidera standarder. En omrevidering ger chans att förbättra misstag som implementerats i standarder, då det kan vara svårt att se misstag innan utvecklare har fått chans att arbeta mot dem.

Olika kompilatorer ser dina program på olika sätt, vilket gör att det är viktigt att testa sin källkod med olika kompilatorer när man skall skriva portabel kod[43]. Bara för att källkod går igenom en kompilator garanterar inte att den går igenom en annan. Genom att anpassa sin källkod för flera kompilatorer blir källkoden mer generell och därmed mer portabel. I vissa fall kommer inte källkod att gå igenom flera kompilatorer även fast den följer standarden, detta visar att utvecklare tolkar språkets standard olika.

[41]Kernighan, Brian W. & Pike, Rob, *The Practice of programming*, 1999
[42]Ibid
[43]Ibid

3.1.2 C

C är ett av det mest förekommande programmeringsspråket idag. Språket uppfanns av Dennis Ritchie på 70-talet när han jobbade på Bell Labs[44]. Språket är ett funktionsorienterat, imperativt programmeringsspråk som härstammar från Algol-familjen. C är en vidareutveckling av det typlösa programmeringsspråket B. C har haft ett stort inflytande på datorindustrin, då det är kraftfullt och relativt litet. C utvecklades tillsammans med UNIX, vilket ger språket ett starkt fäste inom de operativsystem som baseras på- eller är kloner av UNIX. C-kompilatorer finns på de flesta större operativsystemen vilket gör C till ett portabelt språk. C är standardiserat enligt ISO och ANSI, vilket har bidragit till att C-kompilatorer har portats till flertalet plattformar.

C har haft ett stort inflytande på datorindustrin då det har egenskaper och funktioner som talar för dess styrka. Med C kan man skriva applikationer som är kompakta och som exekveras relativt snabbt[45]. Applikationer kan finjusteras för optimerad prestanda och minnesanvändning vilket gör språket kraftfullt. C är speciellt i den bemärkelse att man kan operera direkt på bitar, bytes och minnespekare, till skillnad från språk på högre nivå som Java och C#.

C har dock en stor svaghet, vilken gör att språket kan vara svårt att hantera. Språket skyddar inte programmerare mot misstag i programmeringen. Till skillnad mot språk som Java och C# sköts inte underhållsrutiner av språket, vilket gör att det kan vara svårt att skriva säkra program utan buggar så som minnesläckor och segmenteringsfel. Källkod skriven i C kan även vara svår att debugga[46].

Program 1 Ett enkelt C-program

```
#include <stdio.h>

int main(void)
{
    printf(``hello, world\n'');
    return 0;
}
```

[44]Ullman, Larry & Liyanage, Marc, *C Programming*, 2005
[45]Ibid
[46]Kelly, Al & Pohl, Ira, *A book on C*, 1998

3.1.3 C++

C++ är ett objektorienterat programmeringsspråk som är standardiserat genom ISO och ANSI. Tack vare att C++ är ett högnivåspråk på lägre nivå, är det möjligt att skriva snabba och effektiva program med det. C++-kompilatorer finns på alla större operativsystem, vilket gör språket portabelt[47]. Detta är till stod del möjligt tack vare standardiseringen av språket.

C++ började utvecklas under 1980-talet då man kände att C hade sina begränsningar. Detta har bidragit till att C++ används mer och mer där C förut varit dominerande. Syntaxen i C++ är lik C, men språket är hårdare typat för att programmerare lättare skall kunna skriva säker kod.

Även ifall språken C och C++ liknar varandra, blir programmen skrivna i respektive språk annorlunda uppbyggda. Detta beror till stor del av att C++ stöder objektorienterad programmering. Vid portning av programkod mellan de två språken, kan problematiska konsekvenser uppstå då strukturen mellan programmen skriva i språken skiljer sig åt. Man kan använda mycket av syntaxen och källkoden ur respektive språk, men programmen behöver struktureras om. Detta innebär ofta att programmen måste designas om från grunden. Portning av program mellan dessa typer av språk kan vara problemetiskt vid komplexa system.

Program 2 Ett enkelt C++-program

```
#include <iostream>

int main()
{
    std::cout << ‘‘hello world’’ << std::endl;
    return 0;
}
```

3.1.4 Java

Java är ett objektorienterat programmeringsspråk som introducerades av Sun Microsystems i mitten av 1990-talet. Java-projektet påbörjades 1991 av James Gosling. Sun Microsystems ville implementera en Virtuell Maskin och ett programmeringsspråk som följde C/C++-notation. Det släpptes för allmänheten 1995 och erbjöd möjligheten att kunna exekvera samma kod på flera populära plattformar dit Javas virtuella maskin var portad. Java blev snabbt ett populärt språk då det är objektorienterat, plattformsoberoende

[47] Davis, Stephen Randy, *C++ for Dummies*, 2004

och kan exekveras i en webbläsare.

Sun hade vissa mål de ville uppnå med projektet; Java skulle vara ett objektorienterat språk som även skulle vara plattformsoberoende. Detta är möjligt genom att källkod kompileras till bytekod som sedan kan exekveras i en virtuell maskin. Java är portat till de flesta populära operativsystem. Detta är en av Javas största styrkor. Eftersom det går att exekvera javaprogram på alla plattformar som den virtuella maskinen finns på, slipper man de bekymmer som portning innebär. Dock så är Javas största styrka även en av dess största svagheter; På grund av att programmet behöver köras i den virtuella maskinen, blir programmen långsammare än likvärdiga program skrivna i språk på lägre nivå, som till exempel C++. Dessa problem har Sun adresserat genom att utveckla så kallade JIT-kompilatorer ("Just In Time"). Kompilatorerna kompilerar koden först när den behövs och kan tillämpa lämpliga optimeringar till programmet[48]. Detta har gjort Java-program effektivare vid exekvering, dock så är det inte möjligt att optimera program så som i programmeringsspråk på lägre nivå.

Java och C# har en så kallad "Garbage collector" vilken tar hand om minne som frigjorts av använda objekt. Genom denna hantering slipper programmeraren att manuellt hantera minnet, vilket minimerar risken för minnesläckor hos applikationer. Det blir lättare att skriva säker källkod. Nackdelarna med tekniken är att den behöver resurser av systemet vilket bidrar till sämre prestanda.

Program 3 Ett enkelt Java-program

```
public class Hello {
    public static void main(String [ ] args) {
        System.out.println(''Hello World!'');
    }
}
```

3.1.5 C#

C# är ett objektorienterat programmeringsspråk som är en utveckling av C och C++. Språket har utvecklats av Microsoft som en del av deras .NET-plattform. Språket är influerat av Java, det märks inte minst genom exekvering av kod i en virtuell maskin. Till skillnad från Java så kompileras dock inte källkoden till bytekod, utan till Microsoft Intermediate Language-kod

[48]Wikipedia – Wikipedia, the free encyclopedia

(MSIL)[49] Målet med C# var att det skall vara enkelt, modernt och vara ett mångsidigt objektorienterat programmeringsspråk.

C# är godkänt av ISO och ECMA som ett standardiserat programspråk.

Program 4 Ett enkelt C#-program

```
class ExampleClass
{
        static void Main(
        {
                System.Console.WriteLine(‘‘Hello, world!’’);
        }
}
```

3.1.6 Funktionsbibliotek

Då de flesta programmeringsspråk opererar i olika miljöer där det så gott som alltid är önskvärt med någon form av interaktion med den kringliggande miljön, krävs vissa definitioner av hur språket skall interagera med omgivningen. Exempel på interaktion kan vara allt från grundläggande tjänster som operativsystemet erbjuder som I/O (Input/Output) eller mer avancerad funktionalitet som rendering av 3D-grafik. Denna typen av funktionalitet erbjuds ofta i form av färdiga funktionsbibliotek. Dessa funktionsbibliotek kallas beroende på programmeringsspråk och implementation för en Headerfil eller ett API (Application Programming Interface). Header–filer och API:er står definitionsmässigt utanför själva programmeringsspråken, men är definierade för språket och förväntas fungera i den omgivning som stöds. Syftet med dessa funktionsbibliotek är att erbjuda funktioner för viss typ av funktionalitet som sker på ett standardiserat vis.

Headerfiler används för att beteckna ett avsnitt i början av en källkodsfil, som innehåller information om hur resten av källkodens innehåll skall tolkas[50]. Headerfiler används i hög grad i C och C++ för att bryta ned komplexiteten i källkoden, samt för att skapa enhetlig namngivning av variabler. Trots att detta sätt att skriva kod är avsett för att förenkla för utvecklare kan de medföra problem då det är svårt att underhålla headerfiler. Det är inte ovanligt att man i vanliga headerfiler, exempelvis stdio.h i C, hittar flera deklarationer med kompileringsdirektiv för flera olika typer av kompilatorer. Detta kallas selektiv kompilering och innebär att headerfilen innehåller di-

[49] Wikipedia – Wikipedia, the free encyclopedia

[50] Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999

rektiv av typen #`if` och #`ifdef` för att kompilatorn skall kunna kompilera koden på ett korrekt sätt (se selektiv kompliering). Ett problem med headerfiler är att de kan orsaka konflikter när det gäller namngivning av funktioner. Viktigt är att se till att det inte finns namn på funktioner i headern som förekommer i koden, annars drabbas man av en så kallad namnkonflikt.

Exempel på en enkel headerfil med funktionalitet för att addera två tal och returnera summan[51]:

Program 5 En enkel header–fil

```
header_example.h

int add(int a, int b)
{
    return a + b;
}
```

Genom att importera headerfilen kan man nyttja dess funktionalitet. Detta sker på den första raden i Program 6[52]:

Program 6 Importering av en header–fil

```
#include <header_example.h>

int add(int, int);

int add3(int a, int b, int c)
{
    return add(a, add(b, c));
}
```

Ett API är källkod som innehåller en uppsättning funktioner eller hjälpprogram som syftar till att ge funktionalitet inom ett visst område[53]. Det fungerar som ett gränssnitt mellan programspråket och plattformen. Själva API:et är abstrakt och fungerar som en regelbok för hur de tillhandahållna funktionerna skall användas[54]. Standardbibliotek för varje språk utvecklas och underhålls av kompilatorutvecklaren, men det finns även många inofficiella

[51]Wikipedia – Wikipedia, the free encyclopedia
[52]Ibid
[53]Wikipedia – Wikipedia, the free encyclopedia
[54]Wikipedia – Wikipedia, the free encyclopedia

bibliotek att tillgå i de flesta språk.

Ett problem med API:er är att de täcker ett brett spektrum av funktioner, och ofta tvingas att hantera operativsystemstjänster, vilket kan medföra portabilitetsproblem[55]. Det generella tipset när det gäller bibliotek är att hålla sig till standarderna. Med standard avses vad som anses som standard för språket i fråga. Oavsett språk så gäller; om håller man sig till de standardrutiner som finns fördefinierade för interaktion med operativsystemet och dess rutiner, så finns det goda chanser att ditt program beter sig liknande även under andra förhållanden och i nya miljöer[56].

Program 7 visar hur ett API importeras i Java för att ge tillgång till konstruktorn för en menyrad i ett grafiskt fönster:

Program 7 Importering av API i Java

```
import javax.swing.*;

public class api_test {
    private JMenuBar meny = new JmenuBar();
}
```

[55]Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999
[56]Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999

3.1.7 Programstruktur och selektiv kompilering

Strukturering av program
Vid strukturering och utveckling av portabla program finns det två större infallsvinklar att gå efter när det kommer till hur man skall strukturera sitt program[57]. Det ena är det uniona tillvägagångssättet[58]. Det innebär att man utnyttjar styrkorna hos de olika systemen portningen sker till lokalt. Man låter kompilatorn och installationsprocessen konfigurera programmet så att programmet utnyttjar systemet och dess miljö på bästa sätt, tex genom selektiv kompilering. Detta kan vara lockande då fördelarna är kan verka starka. Speciellt då källkoden och programmen kan optimeras så att de blir effektivare. Nackdelarna innebär dock att programmen blir stora samt källkoden och installationsförfarandet komplext[59].

Det andra angreppsättet är att bara utnyttja de funktioner och egenskaper som finns på målsystemen. Det handlar om att inte använda funktioner om de inte finns på alla målsystem[60]. Kerninghan och Pike kallar detta för "den korsande metoden". Genom att generalisera koden och dess utnyttjande av funktioner, blir den portabel, samt att man slipper underhålla systemspecifik källkod. Nackdelen är att bristen på universellt förekommande funktioner kan begränsa programmets möjligheter, då de kan vara av avgörande vikt för programmets funktioner. En annan nackdel är att programmets prestanda kan bli lidande, då man inte kan optimera det som när man skriver plattformsspecifik kod[61]. Källkod skriven på detta vis kan inte optimeras på samma sätt som union källkod, då den måste generaliseras och vara portabel.

De två angreppssätten skiljer sig åt i och med att den ena metoden inte skapar portabel kod, trots att det är angivet som mål. Den uniona metoden får utvecklare att skriva plattformsberoende kod, där kompilatorn sedan väljer vilka delar av källkoden som skall kompileras, beroende på vilket system den kör på och vilka funktioner som finns tillgängliga för den. När det kommer till portabilitet så är detta ett sätt att gå runt de initiala problem som portabel källkod innebär, inte att lösa dem. Man får en större kodbas att underhålla, samt källkod som inte är enhetlig. Union källkod är portabel på ett missvisande sätt. Med den korsande metoden får man en uniform kodbas som är lättare att underhålla, samtidigt som källkoden vanligtvis blir enklare och mindre komplex[62].

---

[57] Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999
[58] Ibid
[59] Ibid
[60] Ibid
[61] Mindfire solutions, *Porting: Development Techniques*, 2001
[62] Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999

Selektiv kompilering
Ett sätt att utnyttja systemspecifikt källkod på och samtidigt kunna kompilera den på ett annat system, är selektiv kompilering. Genom att instruera kompilatorn att bara kompilera källkod som går igenom på det aktuella systemet, kan man utesluta delar av koden som inte är relevant. Samma kod komplileras på ett annat system under likadana förutsättningar, skillnaden är att en annan del av koden kompileras. Ett sätt att kontrollera detta på är genom #`ifdef`-satser (Program 8):

Program 8 Selektiv kompilering med #`ifdef`, *Kerninghan, Pike - 1999, s. 199*

```
#ifdef NATIVE
    char *astring = ''convert ASCII to native character set'';
#else
#ifdef MAC
    char *astring = ''convert to Mac textfile format'';
#else
#ifdef DOS
    char *astring = ''convert to DOS textfile format'';
#else
    char *astring = ''convert to Unix textfile format'';
#endif /* ?DOS */
#endif /* ?MAC */
#endif /* ?NATIVE */
```

Problemet med koden är att den inte är portabel, trots att den kompilerar på flera olika system. Källkoden beter sig olika på olika system. Utvecklarna behöver modifiera källkoden med #`ifdef`-satser för varje ny miljö koden behöver portas till. Källkod skriven på detta sätt brukar även vara svår att underhålla eftersom information tenderar till att bli utspridd i koden. Genom att byta ut hela sektionen i Program 8 mot en generell kodrad, får man källkod som är fullständigt portabel, enklare och lika informativ:

```
char *astring = ''convert to local text format'';
```

Kod skriven på detta vis behöver inte underhållas eftersom den fungerar på alla system där en C-miljö finns tillgänglig. Inga selektiva argument behövs.

Det absolut största problemet med selektiv kompilering genom #`ifdef`, är att koden kan vara svår att testa[63]. Då en #`ifdef`-sats gör att ett program blir två olika kompilerade program då satsens villkor uppfylls, är det svårt

[63]Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999

att veta ifall alla varianter av programmet har blivit testade. Ändringar i en #ifdef-sats kan betyda att man behöver göra ändringar i en annan, något som är lätt att missa då utvecklaren inte säkert har tillgång till en miljö där satsen kan komplieras. Ändringarna kan bara bli verifierade i de miljöer och förutsättningar som #ifdef-satserna är skrivna att köras under. För varje gång en ny #ifdef-sats läggs till, blir det svårare att isolera och testa just den. Detta på grund av att villkorsreglerna som måste uppfyllas för att köra den, blir mer och mer komplex för varje ny sats[64]. Utvecklarna kan inte heller gardera sig mot fel i källkoden för de styckena då kompilatorn inte kontrollerar koden ifall inte satsens villkor uppfylls.

Ännu värre är att blanda blanda kontrollflöden för kompilering och exekvering med #ifdef-satser. Detta gör att blir källkoden svårläst (Program 9)[65]:

Program 9 Flödeskontroll med #ifdef, *Kerninghan, Pike - 1999, s. 200*

```
#ifndef DISKSYS
    for (i = 1; i <= msg->dbgmsg.msg_total; i++)
#endif
#ifdef DISKSYS
    i = dbgmsgno;
    if (i <= msg->dbgmsg.msg_total)
#endif
    {
        ...
        if (msg->dbgmsg.msg_total == i)
    #ifndef DISKSYS
            break; /* no more messages to wait for */

               about 30 more lines, with further conditional compilation

#endif
    }
```

[64]Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999
[65]Ibid

Oftast kan #`ifdef`-satser bytas ut med renare och enklare metoder. För att illustrera ett exempel så används ofta #`ifdef`-satser för att kontrollera debug-kod (Program 10):

Program 10 Kontroll av debug-kod med #`ifdef`, *Kerninghan, Pike - 1999, s. 200*

```
#ifdef DEBUG
    printf(...);
#endif
```

Trots att koden ser enkel ut, kan en vanlig `if`-sats hantera villkoret lika väl, samt ta bort de problem som selektiv kompilering innebär (Program 11):

Program 11 Kontroll av debug-kod med `if`, *Kerninghan, Pike - 1999, s. 200*

```
enum { DEBUG = 0 };
...
if(DEBUG) {
    printf(...);
}
```

Kompilatorn kommer inte att generera någon kod för det som finns i `if`-satsen ifall `DEBUG` är lika med noll. Men den kommer att ändå att gå igenom sektionen och kontrollera syntaxen för koden. Här har vi ett stort problem med selektiv kompilering. Kompilatorn kontrollerar inte källkoden som exkluderats med #`ifdef`-satser där de blir exkluderade av villkor. Enda sättet att kontrollera koden mot syntaxfel och buggar, är att kompilera och sedan exekvera den på varje system där villkoren för #`ifdef`-satserna kan uppfyllas. Detta gäller för varje #`ifdef`-sats och deras villkor i hela programmet. Och de kan vara många.

Kontroll av källkoden genom selektiv kompilering innebär att koden blir separerad i de sektionerna och skapar komplex källkod som blir svår att underhålla. Det är inte alltid det går att ha en källkodsbas för ett program som skall vara körbart under flera system, men man bör försöka hitta lösningar som fungerar på målsystemen med samma källkod. Interna interface går att ändra på. Kerninghan och Pike hävdar vikten av att hålla källkoden konsistent och fri från #`ifdef`-satser. Ifall man håller på dessa punkter blir koden mer portabel istället för mer specialiserad[66]. *Narrow the intersection,*

[66]Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999

don't broaden the union[67].

Målet är att bara använda sig av funktioner och egenskaper som finns hos de målsystem vilka programet som är under utveckling skrivs till. Då slipper man använda sig av selektiv kompliering som gör det svårt att testa och kompilera källkoden under alla tänkbara förutsättningar. Istället för att lägga till selektiv källkod, bör man skriva om koden där den är ett problem. Detta ökar portabiliteten samt programmet kommer att revideras om och förbättras, istället för att bli mer komplext.

3.1.8 Abstraktion

Att skriva portabel källkod beror inte enbart på val av programmeringsspråk och bibliotek. I vissa fall är det inte möjligt att bara ha en källkodsbas som kompilerar på flera typer av system, trots åtgärder som att undvika selektiv kompilering. Kerninghan och Pike menar att man skall separera portabel och icke-portabel kod i olika filer, så att de olika kodtyperna inte blandas. Genom att isolera systemspecifik källkod, blir det lättare att underhålla applikationer som är designade för flera system[68].

Kerninghan och Pike tar upp texteditorn Sam som ett exempel. Sam är portat till ett flertal populära plattformar, därbland Windows och Unix för att nämna några. Systemgränssnitten till dessa skiljer sig åt på många sätt, både gällande grafikbibliotek och I/O. Trots detta är den mesta koden i Sam generaliserad. Utvecklarna har isolerat den systemspecifika koden till separata filer; `unix.c` innehåller relevant kod för Unix-system och den motsvarande koden för Windows-system återfinns i `windows.c`. Filerna implementerar ett portabelt interface för operativsystemet och hjälper till att hålla kodbasen uniform. Man kan säga att Sam är utvecklat till sitt egna operativsystem som sedan är portat till de olika plattformarna. De olika generella interfacen kräver normalt mer kod än motsvanande traditionell, men portningen till ett nytt operativsystem kräver bara några 100 rader kod. Detta för att man bara skall behöva implementera ett fåtal plattformspecifika interface. Resten är redan implementerat eftersom Sams egna interface är plattformsoberoende.

Utvecklarna till Sam har fått skriva ett eget portabelt bibliotek för grafikbiblioteken det använder sig av. Detta för att kunna hålla Sam så portabelt som möjligt. Grafikbiblioteken i Windows och Unix är nästan helt orelaterade till varandra, vilket har gjort att Sams egna grafikbibliotek innehåller många rader källkod. Källkoden för gränssnittet till X-Windows i Unix, är nästan hälften så stort som resten av Sams källkodsbas. Trots att det krävs

[67]Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999
[68]Ibid

mycket arbete bakom egna gränssnitt, tjänar man på det i längden. En annan intressant effekt är att grafikbiblioteket använt separat, har gjort ett antal andra program portabla[69].

Ytterligare ett exempel på hur abstraktion fungerar är programmeringsspråkens sätt att hantera I/O. De flesta programmeringsspråken har egna bibliotek för detta. Biblioteken presenterar ett interface till filer för programmerare. De kan med hjälp av dessa öppna, stänga, läsa och skriva till filer utan att behöva bry sig om deras fysiska läge på lagringsenheten eller dess struktur[70]. Eftersom olika system har olika sätt att hantera I/O på, behöver programmerare inte hantera detta. Genom att designa ett interface till I/O, vilket portabla språk oftast har gjort, kan programmerare skriva portabel kod. Så länge språket och biblioteken med I/O-gränssnitten finns på systemet skall koden kunna kompileras och exekveras. I ett portabelt språk blir det språkutvecklarnas ansvar att hantera interaktion med systemet, så att applikationsutvecklarna inte behöver ta hänsyn till det.

Bland de utvecklare som har tagit detta koncept längst är de som har designat och skrivit Java. Med Java har de visat hur långt man kan ta konceptet abstraktion[71][72]. Ett Java program exekveras i en virtuell maskin, en simulerad maskin som kan implementeras på i stort sett vilket system som helst. Javas bibliotek och gränssnitt ger utvecklare en uniform åtkomst till funktioner för det underliggande systemet[73]. Detta betyder att användarna av Java som programmeringsspråk kan rita grafik, skapa användargränssnitt, använda nätverksfunktioner på alla plattformar som den virtuella maskinen är portad till. Javas virtuella maskiner hanterar all kommunikation med systemet och hårdvaran medan användaren kan utnyttja biblioteken och gränssnitten för att skriva portabla program. Teoretiskt skall alla kompilerade Java-program kunna exekveras på vilket system som helst där den virtuella maskinen finns, utan behov av någon ändring i programmets källkod[74].

3.1.9 Datautbyte

Portabilitetsproblem kan inträffa när man skickar eller mellanlagrar data på olika sätt i olika system. När man skickar data över något medium som till exempel nätverk, är det viktigt att man använder sig av fördefinierade standarder och protokoll för att minimera risken att datan tolkas på ett felaktigt sätt. Liknande problem kan uppstå även om data inte skickas, utan

[69]Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999
[70]Ibid
[71]Ibid
[72]Wikipedia - Java programming language
[73]Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999
[74]Ibid

bara lagrats för senare inläsning. Problemet är helt enkelt att utdata från ett program inte är av korrekt format för ett annat programs indata och konflikt uppstår. Lösningen på problemet är att sträva efter att skicka data i så enkel form som möjligt. För att skicka data i så enkel, men ändå tillräckligt informativ form, är vanligt textformat det allra bästa[75]. Textfiler är enkla att hantera och är tillgängliga på alla plattformar. Skulle utdata i form av en textfil vara i fel format för indata för ett annat program finns det gott om möjligheter att på ett smidigt sätt modifiera textfilen med enkla verktyg som till exempel Unix `grep` tillsammans med `sed`. Vidare krävs lite eller ingen dokumentation, då nödvändig information anges direkt i textfilen. Trots textfilernas enkelhet finns vissa problem då olika system tolkar textfiler annorlunda.

Ett annat vanligt sätt att föra över data mellan program är binärfiler. Som kontrast till textfiler krävs speciella verktyg för att använda sig av binärfiler. Ofta måste man utnyttja någon form av dekoder för att återställa datan till text igen[76]. Ytterligare problem relaterade till datautbyte i binär form behandlas i stycket 3.1.11.

3.1.10 Byte order

Binärdata är ett problem när det gäller portabilitet. Även fast man inte vill, så är det i vissa fall nödvändigt att använda sig av det. Binärdata har trots allt vissa fördelar som att det går snabbt att dekoda den.

Problemet med binärdata är hur olika typer av maskiner lagrar data i minnet. Alla maskiner har dock en sak gemensamt; 8-bitars byte. Det är när datatyper behöver större lagringsutrymme problem uppstår portabilitetsmässigt[77]. Maskinen reserverar ett antal minnesadresser och skriver sedan datan dit. Det är i vilka minnesadresser och i vilken ordning datan skrivs till dem som skiljer olika maskiner åt. Det finns två olika system för detta: Big-endian– och Little-endian–systemet[78]. De flesta datorer idag, inkluderat x86-plattformen, använder sig av Little-endian–systemet. Det finns dock andra välanvända plattformar som använder sig av Big-endian, som tex Javas virtualla maskin. Vissa maskinarkitekturer klara även båda systemen. Problem uppstår då data lagrat under ett system läses av en maskin som använder det andra. Maskinen läser då minnesadresserna i fel ordning, vilket resulterar felaktigt tolkad data.

[75]Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999
[76]Ibid
[77]Ibid
[78]Wikipedia – Endian

Ta talet 1025 som exempel. Den binära representationen av det är:

```
00000000 00000000 00000100 00000001
```

Tabellen nedan visar hur de olika systemen lagrar talet i minnet relaterat till minnesadressen:

Minnesadress	**Big-endian**	**Little-endian**
00	00000000	00000001
01	00000000	00000100
02	00000100	00000000
03	00000001	00000000

Datan lagras i minnesadresserna i olika ordning. För att illustrera problemet kan vi skriva ett program (Program 12) som lagrar en hexadecimal sträng av datatypen `long`. Datan läses in i som en `unsigned long` innehållande `0x11223344UL`. Sedan konverteras binärdatan om till en `unsigned char` från minnesadressen tillhörande `unsigned long`–variabeln. Det är här tolkningen av datan skiljer sig åt på de olika Endian–systemen.

Program 12 Datarepresentation för Big- och Little-endian–systemen. *Kernighan & Pike, s.205*

```
/* byteorder: display bytes of a long */
int main(void)
{
    unsigned long x;
    unsigned char *p;
    int i;

    /* 11 22 33 44 => big-endian */
    /* 44 33 22 11 => little-endian */

    x = 0x11223344UL;
    p = (unsigned char *) &x;
    for (i = 0; i < sizeof(long); i++)
        printf(``%x '', *p++);
    printf(``\n'');
    return 0;
}
```

Ett 32-bitars Big-endian–system ger output:en:

```
11 22 33 44
```

Ett Little-endian–system kommer att ge en annan output:

```
44 33 22 11
```

Program 12 visar att olika maskiner tolkar data olika, trots att datamanipulationen är simpel. Detta är problematiskt i flera situationer. Tänk att ett program skickar en variabel lagrad med data genom ett nätverksinterface som behöver lagras i flera minnesadresser, alltså större än en byte. Man kan då inte vara säker att datan i variabeln tolkas korrekt på mottagarmaskinen. Följande källkod kan innebära problem när variabeln läses in på en maskin med ett annat endianess–system då datan skickas över nätverket:

```
unsigned short x;
fwrite(&x, sizeof(x), 1, stdout);
```

När mottagarmaskinen tar emot datan med:

```
unsigned short x;
fread(&x, sizeof(x), 1, stdin);
```

kan vi inte vara säkra på att värdet av `x` är oförändrad. Då `x` skickats som `0x1000`, kan det mycket väl ha mottagits som `0x0010`[79].

En vanlig lösning på detta är att utnyttja selektiv kompilering. En kontroll görs som tittar på vilken typ av endian–system maskinen använder sig av, och sorterar sedan datan därefter. Detta kallas även för "byte swapping":

Program 13 "Byte swapping" med selektiv kompilering. *Kerninghan & Pike, s.206*

```
short x;
fread(&x, sizeof(x), 1, stdin);
#ifdef BIG_ENDIAN
/* swap bytes */
x = ((x&0xFF) << 8) | ((x>>8) & 0xFF);
#endif
```

Problemet med denna källkod, selektiv kompilering och dess nackdelar åsido,

[79]Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999

är att den är blir ohanterbar när många två– och fyrbytes heltal blir utbytta. Datan kommer att "swappas" mer än en gång då den passerar från ställe till ställe[80]. Det blir ännu värre för längre datatyper då det finns fler sätt än detta att flytta data.

Det finns en mer portabel lösning på problemet som kringgår "byte swapping" med selektiv kompilering. Genom att alltid bestämma vilken ordning byte skall skrivas och sedan läsas, behöver man inte bry sig om vilket endian–system maskinen använder sig av:

```
unsigned short x;
putchar(x >> 8);       /* write high-order byte */
putchar(x & 0xFF);     /* write low-order byte */
```

För att läsa tillbaka dataströmmen kan man läsa tillbaka in den, byte för byte, och sätta ihop dem:

```
unsigned short x;
x = getchar() << 8;;      /* read high-order byte */
x |= getchar() & 0xFF;    /* read low-order byte */
```

Lösningen går ut på att skriva datan i en definierad sekvens, vilket blir ett portabelt och mer generellt resultat. Kravet för att detta skall fungera, är att datan läses in i rätt ordning och hur många bytes varje objekt har[81].

Prestandamässigt kan det verka som en långsam lösning att behandla data på detta sätt, men förlusten i realtid är försumbar jämfört med kostnad av I/O.

3.1.11 Uppdaterings— och versionskonflikter

Även portabel kod kan komma att råka ut för kompatibilitetsproblem vid uppdateringar. Det kan vara mycket små ändringar i exempelvis operativsystemet eller interface till programmet som kan orsaka stora problem. Problem av detta slag kalls versionskonflikt och uppstår när program som interagerar med andra program eller system slutar fungera vid uppdatering. Uppdateringar av programvara som ursprungligen syftar till att exempelvis täta säkerhetshål, utöka funktionalitet eller på annat sätt förbättra programvaran, kan leda till att den slutar fungera ihop med annan mjukvara. Ett tydligt exempel på detta var när man uppdaterade kommandot `echo` i UNIX för att utöka dess funktionalitet[82].

[80] Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999
[81] Ibid
[82] Ibid

Ursprungligen fungerade kommandot som ett eko i konsol där det “ekar” tillbaka dess argument.

```
$ echo hello, world
hello, world
$
```

Programmet användes i många Shell-scripts där man önskade formaterad utdata. Funktionaliteten i `echo` utökades för att acceptera formateringskommandon, vilket innebar att den nya versionen av `echo` agerade på liknande sätt som en `print`–sats i C:

```
$ echo 'hello\nworld'
hello
world
$
```

Den nya funktionaliteten i `echo` var användbar, men de program som förlitade sig på att den betedde sig som tidigare versioner gjorde, kunde efter uppgradering stöta på problem. Beteendet av exempelvis program som använder sig av `echo $PATH`, var nu beroende av vilken version av `echo` som fanns tillgänglig på systemet.

3.1.12 Internationalisering

Stöd för olika språk innebär problem då olika teckentabeller och enkodningstyper inte alltid är kompatibla med varandra. Formatering av datum, tid och valutor utgör ytterligare problem. De flesta europeiska länder använder sig av ASCII-enkodningar vilka rymmer 8 bitar per tecken. För många språk räcker dock inte 8 bitar till, då deras alfabet innehåller för många tecken. Kodningarna som används i Kina, Japan och Korea utnyttjar 16 bitar per tecken[83]. Detta resulterar i portabilitetsproblem då ett dokument skrivet i 16 bitar skall läsas upp där det inte finns stöd för kodningen. I bästa fall behövs bara speciella fonter för att läsa dokumentet. I andra fall går det inte alls att läsa det.

Unicode är en teckentabell som är utvecklad för att förhindra kompatibilitetsproblem mellan olika språk. Meningen är att en enda typ av kodning skall vara aktuell för alla världens språk[84]. Unicode utnyttjar 16 bitar per tecken och representerar alla kända och vissa mindre kända språk i samma teckentabell, då den är designad för att innehålla mer än ett språk, till skillnad från ASCII-tabeller som Latin1. Detta innebär att en och samma

[83]Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999
[84]Ibid

teckentabell kan utnyttjas utan att behöva bry sig om kodningen samt hur många bitar ett tecken tar upp.

Internationaliseringsproblemen försvinner dock inte helt med Unicode. Teckentabellen introducerar andra problem som vi tagit upp tidigare tack vare att ett tecken inte får plats i en byte. Dessa problem har vi tagit upp innan i sektion 3.1.10. En vanlig lösning mot detta problemet är att skicka dokument skrivna i Unicode som en byte–ström. Tack vare att utvecklarna har tänkt på detta löses då även eventuella problem med bakåtkompatibilitet från ASCII, då Unicode tillåter program att behandla text från otolkade byte–strömmar som Unicode-text från vilket språk som helst[85].

C och C++ inkluderar funktioner för att konvertera byte–strömmar till 16-bitars tecken, så kallade "wide chars", och vice versa. Dock så får man vara uppmärksam att program skrivna med denna typ av variabler och strängar inte fungerar korrekt på maskiner som inte har stöd för teckentabellen[86].

3.2 Reverse Engineering och Re-Engineering

Reverse Engineering är den process där man försöker återskapa utvecklingen av ett program. Man söker sig från källkoden mot högre abstraktionsnivåer för att försöka finna de krav som låg till grund för det ursprungliga programmet. Reverse Engineering är en metod som ursprungligen användes av företag som ville komma ikapp sina konkurrenter. Man köpte sina konkurrenters produkter för att sedan bryta ned dem i beståndsdelar för att försöka förstå dess uppbyggnad. Utifrån dessa återskapades en egen version av samma produkt[87]. Sättet att utveckla, när man återskapar utvecklingsskeendet och dokumentationen för programmet, är en del av ett förebyggande underhåll. Arbetssättet används även när man behöver modifiera dåligt dokumenterad kod eller underhålla kod för framtida modifiering eller portning[88].

3.3 WO–koncept

WO–koncept (Write Once) är ursprungligen en marknadsföringsslogan skapad av Sun Microsystems för att illustrera fördelarna med Java och dess plattformsoberoende egenskaper[89]. WOCA (Write Once Compile Anywhere) avser att källkod som utvecklats i Java skall gå att kompilera till bytekod

[85] Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999

[86] Ibid

[87] Eklund, Sven & Fernlund, Hans, *Programkonstruktion med kvalitet – projekthantering och ISO 9000*, 1998

[88] Ibid

[89] Wikipedia - Write Once Compile Anywhere

i vilken miljö som helst där det finns stöd för språket[90]. Enligt definitionen WOCA uppfylls ett flertal språk av kraven. Viktigt att komma ihåg är att möjligheten att kompilera källkod i olika miljöer inte implicerar portabilitet. Som kontrast till WOCA Lanserade Sun ett eget koncept knutet till Java som de kallar WORA (Write Once Run Anywhere). Syftet med konceptet är att källkod kompilerad till exekverbar binärdata skall gå att köra i alla miljöer där Javas Virtuella Maskin (JVM) finns tillgänglig. Sun menar att portabilitet är något som är mätbart i procent där WORA konceptet representerar 100% portabilitet[91].

3.4 Portningsprocessen

Portningsprocessen kan delas i olika steg vilka kantas av problem och frågor som man bör ställa sig innan man påbörjar själva portningsprojektet. Frågorna i de olika stegen definierar målen för projektet.

Avgöra möjligheten för portning

Det är viktigt att förstå och klargöra målen med portningen[92]. Innan målen specificeras bör man fastställa vilken, eller vilka typer av portning, som är att vänta. Olika nivåer för portningen kan vara antingen:

- Att få en enkel applikation som fungerar tillfredsställande på den nya plattformen, för att senare bygga ut den till en fullt funktionell version.
- Att göra en fullskalig portning där applikationen i sin helhet portas till den nya plattformen.
- Att porta den existerande applikationen samt utöka den med ytterligare funktionalitet för målplattformen. Det faktum att kravspecifikationen och designen av den ursprungliga applikationen kommer att ses över ytterligare, betraktas oftast som en möjlighet att förbättra kvaliteten på programvaran.

Oavsett vilka mål som specificerats i detta skede skall man alltid hålla sig till specifikationen och de satta målen under hela projektets gång. När målen är klargjorda kan man börja undersöka huruvida en realisering av portningsprojektet är möjlig. Man bör ha i åtanke att de specifikationer som skrevs för den ursprungliga applikationen oftast bara gäller för applikationen i det ursprungliga sammanhanget. Alltså för den plattformen och de rådande omständigheterna, vilket påverkar dess specifikationer[93]. Detta innebär att originalspecifikationen för applikationen inte är realiserbar på den nya plattformen, eller ens relevant. Det kan ha funnits restriktioner

[90] Java Technology
[91] Portability and the AVK: An Interview with Bill Shannon
[92] Mindfire solutions, *Porting: Development Techniques*, 2001
[93] Ibid

hos ursprungsplattformen som påverkat specifikationerna för applikationen. Målplattformen för portningsprojektet behöver inte nödvändigtvis ha dessa egenskaper, utan kan besitta egna plattformsspecifika egenskaper som den nya specifikationen måste ta hänsyn till.

Förståelse för den ursprungliga applikationen
Att förstå den ursprungliga applikationen är ett måste för att kunna implementera den på en annan plattform. Utvecklarna förväntas inte ha någon djupare insikt i applikationen i detta skede utan att studerat ursprungsapplikationen mer ingående[94]. Om de ursprungliga designdokumenten finns tillgängliga används de med fördel, annars bör man försöka återskapa dem. Vid eventuellt återskapande av designdokumentation, används eventuell användardokumentation för att kartlägga funktionalitet och beteende i detalj. Detta illustreras med fördel i form av klassdiagram.

Val av utvecklingsverktyg
Att välja passande utvecklingsverktyg kan vara ett kritiskt steg i portningsprocessen[95]. Rätt beslut kan förenkla processen i hög grad samtidigt som ett dåligt och ogenomtänkt val av utvecklingsverktyg kan försvåra arbetet. Om det ursprungliga utvecklingsverktyget finns tillgängligt på målplattformen, skall det alltid övervägas först. Även om samma verktyg finns tillgängligt på målplattformen så behöver det inte vara det naturliga valet. Faktum är att det kan finnas andra verktyg som är betydligt mer mogna och kraftfulla på den nya plattformen. I fall där det finns kraftfullare verktyg hos målplattformen, bör man överväga en migrering till det nya utvecklingsverktyget. Detta kan i sin tur kan spara arbete vid portningen av själva applikationen[96].

Validera ursprunglig design
Effektiv portning handlar inte bara om att utveckla mjukvara för målplattformen. Det handlar även om att säkerställa arbetet med kommande versioner på de båda plattformarna[97]. För att uppnå detta skall den ursprungliga designen kritiskt ifrågasättas med de nya kraven för målplattformen i fokus. Att designen testas ur ett annat perspektiv gör att den nya designen blir mer generell och stabil[98].

En viktig faktor när man jobbar med samma design på olika plattformar är att sträva efter att hålla en gemensam kodbas. Att använda en gemensam kodbas är en faktor som bidrar till att portabiliteten och kvalitén ökar.

[94]Mindfire solutions, *Porting: Development Techniques*, 2001
[95]Ibid
[96]Ibid
[97]Ibid
[98]Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, 1999

I praktiken innebär det att man utvecklar till de olika plattformarna parallellt. När en ny version släpps på en av plattformarna skall den även släppas på de övriga plattformarna. Detta sätt att utveckla på bidrar till att portabel källkod skrivs och skrivs om på olika plattformar med en och samma utgångspunkt. Utvecklingsmetoden bidrar till enklare underhåll av koden[99]. Den gemensamma kodbasen möjliggör att källkod kan delas upp i plattformsspecifik och applikationsspecifik källkod. Detta gör att mycket tid kan sparas genom att man endast behöver utveckla den plattformsspecifika koden varje gång applikationen skall flyttas.

Val av portningsstrategi

Strategier som används vid portning av mjukvara kan skilja sig åt beroende på vilket projekt det rör. Det finns dock vissa tumregler, något som alla projekt bör sträva efter för en effektiv portning av källkoden. Första prioritet är att man i ett tidigt skede skall sträva efter att få koden i körbart skick på målplattformen[100]. Med körbart skick avses att källkod från ursprungsapplikationen som placeras på målplattformen så snabbt som möjligt skall gå att exekvera i någon mån. Här sker det första portningsstrategiska valet.

De två strategierna som är radikalt olika, avgör hur själva flytten av källkoden kommer att gå tillväga. Den första strategin som kallas - - (minus-minus) innebär att hela den ursprungliga källkoden flyttas till målplattformen, där man med hjälp av "trial and error" testar vilken kod som måste modifieras, och vilken som är direkt körbar. Det andra angreppssättet är att tillämpa strategin ++ (plus-plus), vilket innebär att man för över en liten, och ofta simpel del från den ursprungliga källkoden till målplattformen för att sedan bygga på funktionalitet efter hand[101].

I de fall där applikationen består av abstraktionslager (se skiktad arkitektur) skall man angripa dessa först. Detta görs genom att se till att dataströmmarna in och ut mellan de olika lagren sker på samma sätt. Eftersom det kan finnas möjliga skillnader mellan ursprungsplattformen och målplattformen, skall dessa beaktas strategiskt innan man börjar programmera. Olikheter mellan plattformarna kan bero på hårdvara, vilket operativsystem som körs, eller vilka verktyg som används. På samma sätt som man förväntas följa målen genom hela projektet, förväntas man följa den valda strategin[102].

[99]Mindfire solutions, *Porting: Development Techniques*, 2001
[100]Ibid
[101]Ibid
[102]Ibid

Hantera resurser

De resurser som används för att utnyttja programmets funktionalitet, användargränssnitt och/eller hjälpfiler bör portas separat för att passa målplattformen. Trots att användargränssnittet kan vara väl designat och lämpa sig för ursprungsplattformen, behöver det inte vara en passande design för målplattformen. Man bör utveckla användargränssnittet separat för att bibehålla målplattformens "look and feel". Viktigt är dock att inte ändra i programmets huvudfunktionalitet i denna process. De ursprungliga hjälpfilerna kan vara i ett plattformsspecifikt format och måste därför konverteras eller skrivas om till ett format som stöds av målplattformen, eller konverteras till ett portabelt format som stöder båda plattformarna[103].

Partitionera Portningen

Att partitionera portningsprojektet innebär att man bryter ner det i mindre beståndsdelar för att få en tydligare överblick av vad som kommer att ändras under portningsfasen.

Konstruktion, debugging och testning

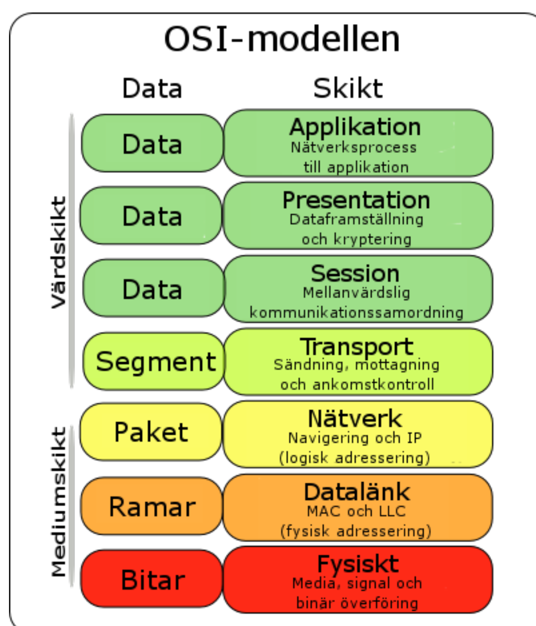
Själva processen att utveckla koden för målplattformen liknar på många sätt ett vanligt utvecklingsprojekt. Man följer i princip samma steg genom projektet, konstruktion, debugging och testning. En viktig skillnad mellan nyutveckling och portning är att utvecklingen av koden på den nya plattformen kan, och bör vara enklare om man tagit hänsyn till vissa aspekter. Majoriteten av buggarna i applikationen på den nya plattformen kommer att vara härledbara till tekniska skillnader mellan ursprungs- och målplattformen. Om man beaktat dessa skillnader initialt, samt under utvecklingens gång, kan många buggar undvikas. På samma vis kan man under testfasen lokalisera potentiella brister i källkoden, eftersom man av tidigare erfarenhet vet inom vilka områden majoriteten av problemen återfinns. Detta gör att man kan planera och fokusera testandet på ett effektivt sätt[104]. Viktigt är dock att noga överväga om man skall åtgärda kända buggar från ursprungsplattformen när man utvecklar "ny" kod på målplattformen. Att åtgärda en bugg är naturligtvis något positivt, men skapar samtidigt inkonsistest mellan de olika plattformarna[105].

Skiktad arkitektur

En väl designad applikation separerar olika lager av funktionalitet för att skapa strikta gränssnitt mellan dem[106]. Ett abstraktionslager är ett lager i en skiktad modell för att bryta ned komplexitet inom källkoden. Abstraktionslagret döljer de tekniska detaljerna bakom ett skikt av kod, vilket erbjuder

[103] Mindfire solutions, *Porting: Development Techniques*, 2001
[104] Ibid
[105] Mindfire solutions, *Testing a Software Port*, 2001
[106] Mindfire solutions, *Porting: Development Techniques*, 2001

ett enklare gränssnitt mot underliggande och överliggande lager. Ett klassikt exempel på en skiktat arkitektur är den så kallade OSI-modellen (Open Systems Interconnection).



Bilden visar en standardiserad modell för datorkommunikation. Modellen illustrerar hur olika lager frigjorda från varandra samverkar för att tillsammans uppnå ett gemensamt mål. Syftet med detta är att erbjuda en skalbar lösning. Varje lager är enbart beroende av vad som sker hos det ovanliggande och underliggande lagret, vilket innebär att funktionalitet i ett lager kan bytas ut utan att de andra påverkas. Detta innebär att man i lagret kan använda valfri teknisk lösning, förutsatt att den kommunicerar med omslutande lager på ett korrekt sätt. På exempelvis sessionsnivå, vilken ansvarar för initiering, hantering och terminering av anslutningen, innebär det att man kan använda sig av protokoll som RIP, SAP, VPN med flera för att fullfölja lagrets uppgift[107].

Packaging and Releasing

Det sista man gör innan release är att revidera dokumentation om nödvändigt. Även om applikationen är densamma som vilken dokumentationen skrevs för, kan det finnas skillnader i det praktiska användandet hos den nya plattformen. Innan release paketeras dokumentation och applikation tillsammans till en exekverbar installer eller i annat lämpligt format för den nya plattformen. Om programmet tillhör en gemensam kodbas skall den nya källkoden

[107] Wikipedia - OSI–modellen

integreras i kodbasen för arkivering och revisionskontroll. Även kod från olika källkodsbaser bör integreras. Detta för att eventuella skillnader mellan ursprungsapplikationen och den portade applikationen minimeras. Tack vare detta säkerställs likheter mellan versionerna för de båda plattformarna[108].

[108]Mindfire solutions, *Porting: Development Techniques*, 2001

4 Resultat

Detta kapitel behandlar resultatet av vårt projekt med fokus på vår ställda forskningsfråga. Vi beskriver varför vi har gjort vissa val i portningsprocessen, samt hur vi resonerat kring dessa utifrån den teori vi beskrivit i föregående kapitel.

4.1 Diskussion

I takt med att projektet fortlöpte insåg vi att vi frångick vår frågeställning mer och mer. Syftet med att försöka fastställa kritiska framgångsfaktorer vid portning av mjukvara, övergick under litteraturstudien till att börja behandla begreppet portabilitet på en mer generell nivå. Utfallet av detta har inneburit att vi granskat vårt projekt utifrån begreppet portabilitet för att finna för– och nackdelar inom ämnet.

Syftet med projektet var att porta en Windows–applikation skriven i C# till en Linux–miljö. Applikationen var skriven med plattformsspecifik källkod vilken inte var portabel. Applikationen bestod av ett antal moduler som tillsammans skapade funktionalitet för att erbjuda kryptografiska tjänster för nätverkskommunikation.

4.1.1 Designkriterier

Att porta en modul som ingår i en applikation kräver att man har god insikt i hur både applikationen fungerar och hur den använder sig av modulen. Då applikationen bestod av över 30000 rader kod placerad i ett flertal olika klasser och moduler, ansåg vi att granskning av den befintliga koden var ett ofördelaktigt tillvägagångssätt för att nå insikt i applikationens struktur. För att bilda oss en uppfattning över hur applikationen fungerade hade vi som önskemål att granska applikationens designdokument och dokumentation.

Bristande dokumentation försvårade processen med att skapa förståelse för designen av ursprungsapplikationen. Detta var ett problem i arbetet med att specificera korrekta designkrav för applikationen som skulle utvecklas på den nya plattformen. För att fastställa designkriterier för den nya applikationen granskade vi koden parallellt med att en dialog med utvecklarna av ursprungsapplikationen fördes. De designkrav som togs fram baserades till stor del på den dialog som fördes med utvecklarna. Då källkoden till stor del saknade kommentarer var granskningen av den i detta stadiet mindre givande.

4.1.2 Val av programmeringsspråk

Valet av programmeringsspråk skall bero på specifikt satta krav gällande både funktionalitet och applikationens behov. Dessa krav är beroende av projektet samt utvecklarnas preferenser och behov, eftersom alla projekt är olika. De krav vi satte på språket var att det skulle stödja objektorienterad programmering samt vara portabelt.

Kraven vi satt gjorde att vi valde bort att fortsätta utveckla applikationen i C#, då vi kände att utvecklingen av .NET–kompatibla verktyg (Mono) inte var tillräckligt mogna för våra behov på målplattformen. Trots att språket är standardiserat för att det i teorin skall kunna vara portabelt, täcker inte standardiseringen applikationerna runt språket som finns hos .NET-ramverket från Microsoft. Detta betyder att Mono inte är ett fullvärdigt alternativ gentemot .NET-ramverket i Windows–miljö, då Microsoft har patenterat lösningar i sitt ramverk. Detta innebär i praktiken att mjukvara skriven i C# inte är portabel mellan plattformar.

Java är en stark kandidat ur ett portabilitetsperspektiv då det är plattformsoberoende tack vare sin virtuella maskin. Skälet till varför vi valde bort Java var en punkt gällande vår kravspecifikation; Storlek och beroenden av programmet. Ett program skrivet i Java behöver nödvändigtvis inte vara större än ett program skrivet i andra högnivåspråk som C++, men Java–program är beroende av en JRE (Java Runtime Environment). Detta innebär att en användare som inte har en JRE på sitt system, måste installera en innan Java-programmet kan exekveras. Javas tidigare licensiering har inneburit att en JRE inte har kunnat distribueras med andra populära operativsystem än Sun Microsystems egna, vilket innebär att det är upp till användaren att installera den. Applikationen portningen avsåg skulle vara lätt att distribuera och installera, vilket gjorde att Java föll bort som val, trots språkets uppenbara fördelar gällande portabilitet.

Valet av programmeringsspråk föll på C++ då det tillfredsställde kraven gällande portabilitet, objektorientering samt tillgänglighet. Utvecklingsverktyg och kompilatorer finns portade till de flesta plattformar vilket ger en framtidssäkring för vår kodbas. Hög tillgänglighet gällande dokumentation och funktionsbibliotek var även ett skäl till varför valet föll på språket.

4.1.3 Val av funktionsbibliotek

Nästa viktiga beslut gällde valet av funktionsbibliotek för kryptografiska rutiner. Kraven gällande bibliotek utöver kompatibilitet gentemot valt programmeringsspråk var att det skulle vara väldokumenterat samt stödja ob-

jektorienterad programmering utöver den funktionalitet applikationen krävde.

Tre olika API:er valdes ut och jämfördes mot varandra; Crypto++, Botan och CryptoSys. Alla är skrivna i C++ och stöder objektorienterad programmering. Biblioteken är fria att använda, vilket var en fördel men inget krav. Ett problem med samtliga API:er var att inget av dem hade en övergriplig dokumentation, vilket gjorde valet svårt. Bristen av dokumentation gällande exempel och funktionsbeskrivningar om biblioteket skulle medföra problem, då vi endast hade grundläggande kunskaper om kryptografi. Detta fick negativa konsekvenser i utvecklingsarbetet då mycket tid fick läggas på att lära sig hantera bibliotekets funktionalitet. I efterhand anser vi att god dokumentation bör vara en viktig faktor i val av API.

Vi ansåg att CryptoSys inte var lämpligt då Linux-versionen var märkt som en beta-release. Valet mellan Crypto++ och Botan var desto svårare då de verkade vara likvärdiga gällande både funktionalitet, stabilitet och användarbas. Valet föll slutligen på Crypto++ då den använts i ett flertal kända applikationer som Bitvise WinSSHD, Autodesk Streamline och MAILguardian.

4.1.4 Flera källkodsbaser

Det faktum att ursprungsapplikationen var starkt knuten till Microsoft och dess plattform medförde att källkoden för applikationen var optimerad för Windows och inget annat. För att programmet skulle kunna göras tillgängligt på flera plattformar krävdes det att det skrevs i ett programmeringsspråk som fanns tillgängligt på flera plattformar. Migreringen av språket medförde även en migrering av API och utvecklingsmiljö. Förändringarna gjorde att den ursprungliga kodbasen inte kunde användas.

Till en början hade vi tänkt att skriva två källkodsbaser, en för Windows och en för Linux-baserade operativsystem. Ju längre vi kom i utvecklingen av vår portning, insåg vi nackdelar med detta. Vi skrev om källkoden i ett annat programmeringsspråk där resultatet blev ett portabelt program. Tack vare att vi använt oss av ett standardiserat och portabelt programmeringsspråk tillsammans med ett portabelt funktionsbibliotek, kunde vi kompilera delar av applikationen vi skrivit på ett flertal plattformar.

Vi insåg att vi hade skrivit om delar av applikationen, förbättrat dem och gjort dem portabla. Detta är någonting som borde gjorts när applikationen designades och skrevs för första gången. Syftet med projektet blev nu istället att skriva om den ursprungliga källkodsbasen med ett portabelt programmeringsspråk för att slippa underhålla och utveckla ett flertal källkodsbaser.

Detta var dock problematiskt ur andra perspektiv. Realiseringen av projektet med en ny kodbas innebar att vi själva var tvungna att skriva testprogram för att validera att modulen fungerade korrekt på båda plattformarna. Ett annat problem var att vi vid utvecklandet av modulen var tvungna att göra ett antal egna antaganden över hur resten av applikationen skulle designas för att kompatibilitet med modulen vi utvecklade skulle uppnås.

Projektet lärde oss att en gemensam källkodsbas är lättare att hantera i längden, då den sparar arbete gentemot att underhålla flera källkodsbaser. Dock uppstår nya problem med generaliserad källkod samt nya mål. Portabel källkod är per definition generell källkod. Detta innebär att det kan vara svårt att utnyttja systemets resurser optimalt då källkoden inte blir systemspecifik.

4.1.5 Internationalisering

Problemet med internationalisering uppstod i vårt projekt när char-array:er och strängar skickades mellan olika funktioner. En deklarerad `string` och `char` i C++ kan inte innehålla tecken kodade i större än 8 bitar. Det innebar att tecken i Unicode inte alltid kunde behandlas korrekt av funktioner i programmet. System som använder olika enkodningstyper fick problem med att tolka varandras strängar. Vi löste det genom att alltid skicka strängar konverterade till array:er av så kallade “wide characters”. Då `wchar` och `wstring` är datatyper som hanterar tecken kodade i större än 8 bitar, kunde de hantera strängar enkodade i UTF-8. Tyvärr blev koden inte fullt portabel då storleken på en `wchar` bestäms av kompilatorn och dess implementation av datatypen.

En bättre lösning på problemet är att alltid skicka tecken kodade i Unicode mellan program och objekt, då man alltid kan förutsätta hur datan skall behandlas när den tas emot. Problemet med detta är dock att datorn som exekverar programmet måste ha stöd för Unicode, vilket gör att detta fortfarande inte är en felsäker lösning på internationaliseringsproblemen.

5 Slutsats

Beslut som fattas i det tidiga skedet i ett utvecklingsprojekt rörande design är något som fundamentalt påverkar applikationen eller systemet i fråga för all framtid. En slarvigt designat applikation kan vara svår att ändra vid uppdateringar, vilket leder till extra jobb. Migrering av sådan programvara kan komma att kräva ytterligare jobb, eller till och med helt ny design.

Genom att följa de rekommendationer som finns för att skriva portabel och generell källkod kan man spara mycket framtida jobb när det blir dags att porta en applikation till en annan plattform. Trots att de initiala planerna för applikationer inte omfattar andra plattformar, visar projektet att planerna för applikationer ändras med tiden. Kompatibilitet med applikationers befintliga miljö kan komma att ändras av flera skäl. Det kan vara allt från uppgradering av operativsystem till byte av kompilator eller hårdvara. Ju mindre programmet beror på hårdvaran eller plattformen det kör på, desto mindre sannolikhet att programmet beter sig annorlunda i den nya miljön utan revidering.

Att tillämpa portabilitetsförbättrande programmering är att tillämpa god programmeringskutym. Den extra tid och kraft som läggs ner på portabilitet leder till bättre design, bättre konstruktion samt noggrannare testning.

De kritiska framgångsfaktorerna vid portning av mjukvara som vi identifierat är följande:

- En väl utförd förstudie med granskning av befintlig dokumentation och källkod för validering av ursprunglig design.
- Utnyttja standarder i största möjliga mån gällande programmeringsspråk och verktyg vid utvecklandet av källkod.
- Använd högnivåspråk i de fall där det är möjligt.
- Migrera befintlig och ny källkod till en uniform källkodsbas.
- Porta moduler och abstraktionslager separat.
- Modularisera systemspecifik källkod för att isolera beroenden.
- Skriv källkod som hanterar en specifik teckentabell och kodning, lämpligen Unicode, för att undvika internationaliseringsproblem.

6 Referenser

6.1 Litteratur

Hansson, Agneta, *Praktiskt taget: Aktionsforskning som teori och praktik*, Göteborg, Sociologiska Instutionen, 2003

Reason, Peter & Bradbury, Hilary, *Handbook of action research: Participative inquiry and practice*, London: SAGE, 2001

Denscomb, Martyn, *Forskningshandboken : för småskaliga forskningsprojekt inom samhällsvetenskaperna*, Lund: Studentlitteratur, 2000

Lundberg, Bertil & Starrin, Bengt, *Participatory research: Tradition, theory and practice*, Karlstad: Division for social sciences, 2001

Svensson, Lennart, *Interaktiv forskning - för utveckling av teori och praktik*, Stockholm: Arbetslivsinstitutet, cop. 2002

Gustavsson, Bernt, *Bildning i vår tid: Om bildningens möjligheter och villkor i det moderna samhället*, Stockholm: Wahlström & Widstrand, 1996

Whyte, William Foote, *Participatory Action Research: Through Practice to Science in Social Research*, Newbury Park: Sage, cop. 1991

Kerninghan, Brian W. & Pike, Rob, *The Practice of programming*, Reading: Addison–Wesley, 1999

Ullman, Larry & Liyanage, Marc, *C Programming*, Berkeley: Peachpit Press, 2005

Kelly, Al & Pohl, Ira, *A book on C*, Indianapolis: Addison–Wesley, 1998

Davis, Stephen Randy, *C++ for Dummies*, Hoboken: Wiley Publishing Inc., 2004

Eklund, Sven & Fernlund, Hans, *Programkonstruktion med kvalitet – projekthantering och ISO 9000*, Lund: Studentlitteratur, 1998

6.2 Artiklar

Kontrollerade 2007-05-28

Rapoport, R.N, *Three Dilemmas in Action Research*, Human Relations, 1970
Rapoport, 1970

Brown, D. & Tandon, R, *Ideology and political economy in inquiry: Action research and participatory research*
http://jab.sagepub.com/cgi/reprint/19/3/277?ck=nck

Wadsworth, Yoland, *What is Participatory Action Research?*, 1998
http://www.scu.edu.au/schools/gcm/ar/ari/p-ywadsworth98.html
Wadsworth, Yoland, 1998

Susman, G.I, *Action Research: A Sociotechnical systems perspective*, London: Sage Publications, 1983
Susman, 1983

Mindfire solutions, *Porting: Development Techniques*, 2001
http://www.portinggurus.com/Downloads/Articles/Porting=DevelopmentTechniques.pdf
Mindfire solutions, 2001

Mindfire solutions, *Testing a Software Port*, 2001
http://www.portinggurus.com/Downloads/Articles/Porting=TestTechniques.pdf
Mindfire solutions, 2001

6.3 Webbreferenser

Kontrollerade 2007-05-28

W3Schools Online Web Tutorials - Browser statistics
http://www.w3schools.com/browsers/browsers_stats.asp

Wikipedia -- Wikipedia, Java Programming Language
http://en.wikipedia.org/wiki/Java_%28programming_language%29

Wikipedia -- Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/C_sharp

Wikipedia -- Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Header_file

Wikipedia -- Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Library_%28computing%29

Wikipedia -- Wikipedia, the free encyclopedia
http://sv.wikipedia.org/wiki/API

Wikipedia -- Wikipedia, Endian
http://en.wikipedia.org/wiki/Endian

Wikipedia -- Wikipedia, Write Once Compile Anywhere
http://en.wikipedia.org/wiki/Write_once%2C_compile_anywhere

Java Technology
http://java.sun.com

Portability and the AVK: An Interview with Bill Shannon
http://java.sun.com/developer/technicalArticles/Interviews/AVK/index.html

Wikipedia -- Wikipedia, OSI--modellen
http://sv.wikipedia.org/wiki/OSI-modellen