Plagiat i samband med programmering – problem, detektering och förebyggande pedagogiska strategier

Görel Hedin och Eva Magnusson, Institutionen för datavetenskap, LTH

Sammanfattning. **I denna artikel behandlar vi frågeställningar rörande plagiat i samband med programmering. Problematiken är relevant även för andra ämnen där man använder programmering som problemlösningsmetod eller där man använder ett begränsat formellt språk för redovisningar som t.ex. inom matematik. Vi diskuterar skillnader mellan programplagiering och uppsatsplagiering samt vilka system som finns för detektering. Utgående från relevant pedagogisk litteratur och några fallstudier från vår egen institution ger vi också förslag på ett antal pedagogiska strategier för att förebygga fusk i form av plagiat.**

I. Inledning

En av de vanligaste formerna av fusk inom universitetskurser är plagiat i samband med inlämningsuppgifter, hemtentamina o.d. Plagiat defnieras i detta sammanhang som "att lämna in någon annans arbete som sitt eget" [2]. För ämnen där sådana uppgifter består av vanlig text finns flera bra verktyg för detektering.

För många discipliner inom tekniska och naturvetenskapliga områden är dock förhållandena speciella eftersom man uttrycker sig med hjälp av begränsade formella språk. Ett exempel är matematik. Krav på logisk uppbyggnad och exakthet gör att det är viktigt att studenterna tillägnar sig och använder detta språk i stället för att använda "egna ord". Att ett svar på en tentamensfråga hamnar väldigt nära formuleringar i läroboken behöver därför inte vara en indikation på fusk. Ett annat exempel är datavetenskap. Inlämningsuppgifter och laborationer innebär här att studenterna skriver program som löser en viss uppgift. Lösningar uttrycks i form av algoritmer som implementeras i något programmeringsspråk. Dessa språk har, i jämförelse med naturliga språk, en mycket enkel syntax. Antalet olika sätt på vilka lösningen kan uttryckas är också begränsat. Små program på inledande kurser kan därför bli väldigt lika utan att plagiering förekommit. Ett program kan å andra sidan vara plagiat även om det textmässigt skiljer sig väsentligt från originalprogrammet, nämligen om det konstruerats genom en rad rutinmässiga förändringar av originalet.

I denna artikel behandlas frågeställningar angående programkodsplagiat, d.v.s. plagiat i samband med programmering. I de följande avsnitten kommer vi att beskriva hur programkodsplagiering skiljer sig från uppsatsplagiering och vilka system som finns för plagiatdetektering i program. Utgående från några fallstudier från vår egen institution ger vi till sist några preliminära rekommendationer för lärare.

Denna artikel är en förkortad version av författarnas projektrapport i den högskolepedagogiska kursen "Akademisk hederlighet – Studenter bortom plagiat", våren 2008.

II. Speciella problem för Plagiatdetektering i programkod

Programkod lagras normalt som textdokument. Om en student har kopierat sin lösning rakt av är det enkelt att detektera plagiat genom att använda ett textjämförelseverktyg. Dock är det förhållandevis enkelt för en student att kamouflera plagiat av program genom att göra små enkla ändringar, utan att förstå vad programmet egentligen gör. Parker och Hamblen definierar ett plagierat program som "ett program som har skapats från ett annat program genom ett litet antal rutinmässiga transformationer" [4].

Faidhi och Robinson har identifierat följande sex nivåer för hur ett program kan ändras när en student vill försöka undgå plagiatdetektering [3]. Varje nivå inkluderar ändringar av de tidigare nivåernas typ.

1. Ändringar av kommentarer och indentering.
2. Omnamning av identifierare, t.ex. variabelnamn.
3. Ändringar av deklarationer, t.ex. ordningsbyte.
4. Ändringar av metoder, t.ex. att införa en metod för en delberäkning.
5. Ändringar av satser, t.ex. att ersätta en while-loop med en for-loop.
6. Ändringar av villkorsuttryck, t.ex. att byta a <= b mot (a-b) <= 0.

I moderna programmeringsverktyg finns stöd för s.k. refaktorisering, d.v.s. att ändra ett program utan att påverka dess beteende. Att namna om en variabel görs t.ex. enkelt med ett enda kommando. Dagens programmeringsmiljöer underlättar således arbetet för en fuskare. Detta innebär dock inte att det är helt enkelt att undgå detektering. Förmågan att utföra olika typer av plagiat kan snarare relateras till den nivå i den s.k. SOLO-taxonomin (se t.ex. [1], kap. 3) på vilken studenten befinner sig. Plagiat på nivå 1 kan utföras av den som befinner sig på det lägsta trappsteget i SOLO-taxonomin, den prestrukturella nivån. Plagiat på nivå 2 och 3 kräver en något djupare förståelse motsvarande den unistrukturella eller den

multistrukturella nivån i SOLO. Det krävs t.ex. en hel del kunskaper i programmering för att förstå vilka namn som kan ändras och hur de kan ändras utan att programmets funktionalitet går förlorad. För att kunna utföra plagiat på nivå 4-6 krävs förmåga att tillämpa och kombinera kunskaper om semantiken för olika delar av programspråket samt förmåga att uttrycka generaliseringar i språket. Denna typ av plagiat kan därför endast utföras av dem som befinner sig på den multistrukturella eller den relationella nivån enligt SOLO.

III. System för plagiatdetektering i program

Det centrala problemet vid detektering av plagierad programkod är att försöka hitta kod där strukturen överensstämmer med originalet, trots att programtexten ytligt sett kan vara mycket olik. Det finns flera existerande system som detekterar sådana strukturella likheter. De mest kända är JPlag [5] och Moss [6]. De bygger båda på en tvåstegsmetod för att analysera programlikhet. Först *tokeniseras* programmet, d.v.s. texten transformeras från en ström av tecken till en ström av nyckelord (t.ex. for, while, ...), identifierare (t.ex. sum, count, ...), operatorer (t.ex. +, -, ...) och separatorer (t.ex. parenteser och semikolon). Därefter identifierar man delsekvenser som stämmer överens i de två program man jämför. Om det finns delsekvenser som är tillräckligt långa och lika, redovisas dessa likheter i en rapport för programjämförelsen.

Plagiatdetekteringen kan jämföras med de olika förändringsnivåerna redovisade i förra avsnittet. Vid tokeniseringen ignoreras kommentarer och "whitespace", alltså blanktecken, tabbar, och radbrytningar. Vidare representeras alla identifierare i programmet av ett slags token, utan hänsyn till identifierarnamnet. Program som modifierats enligt nivåerna 1-2 har därför samma tokenisering som originalet. De högre nivåernas förändringar innebär förändringar av tokensekvensen, och kan därmed störa plagiatdetekteringen. Det krävs dock att man inför tillräckligt många sådana förändringar för att verktygen inte skall upptäcka likheterna.

IV. Pedagogiska strategier mot plagiat

Utgående från fallstudier vid vår egen institution har vi funnit att två typer av fusk dominerar i samband med inlämningsuppgifter och projekt. Den ena typen är s.k. "free riding" d.v.s. att en student "åker snålskjuts" på övriga deltagare i en grupp och inte utför sin del av arbetet. Den andra typen är plagiat till följd av upplevd tidspress. Följande är exempel på strategier som vi använder för att förebygga dessa typer av fusk:

- *Adekvata formella förkunskapskrav*. En vanlig anledning till att en student kan frestas åka snålskjuts på andra är bristande förkunskaper.
- *Kontroll av förberedelser*. Det krävs oftast att man förbereder sig inför laborationer o.d. genom att läsa på angivna avsnitt i kursbok e.d. En lämplig strategi mot free riding är att individuellt kontrollera att förberedelserna gjorts.
- *Återkoppling till gruppaktivitet i samband med skriftlig tentamen*. Vid tentamen finns det möjlighet att återkoppla till de uppgifter som lösts i samband med de obligatoriska momenten i kursen. Genom att under kursens gång vara tydlig med att detta förekommer kan frestelsen att åka snålskjuts på andra minska.
- *God planering avseende kursmoment*. Som kursansvariga kan vi se till att momenten inom kurser schemaläggs så att studenterna inte skall hamna i tidsnöd. Detta låter självklart men kan ändå vara svårt att systematiskt genomföra inom kurser vid LTH, där vi har korta läsperioder omfattande sju veckor.
- *God lärartillgänglighet*. Studenter som "kör fast" på en inlämningsuppgift kan frestas att ta för mycket hjälp av studiekamrater. Därför är det viktigt att studenterna kan få hjälp av lärare. Detta kräver att kursansvarig och/eller andra lärare på kursen är tillgängliga och att studenterna känner till detta.
- *Individualiserade eller varierade uppgifter*. Ett råd som bl.a. återfinns i [2], kap. 2 är att man bör undvika att använda samma uppgifter år efter år. En variation på samma tema är att inte låta alla studenter/grupper lösa samma uppgift.
- *Information om vad som är tillåtet/förbjudet*. Inom civilingenjörsutbildningar uppmanas ofta studenter att samarbeta för att kunna lösa uppgifter. Det är därför angeläget att klargöra var gränsen går mellan tillåtet samarbete och fusk. Vi har försökt formulera sådana regler. De återges i kursmateral och finns även på institutionens hemsidor [7].

V. Diskussion

Som lärare har man begränsade resurser och det är alltid frågan om hur man använder dem på bästa sätt. Varje kurs har sina förutsättningar och vi ser det som viktigt att strategier och verktyg anpassas till varje kurs snarare än att förorda några patentlösningar. Samtidigt är det viktigt att reglerna är klara för studenterna.

Vi tillämpar inom kurser i datavetenskap ett antal generella strategier som kan bidraga till att förebygga fusk. Samtidigt finns det inom vårt och flera andra tekniska ämnen speciella problem med att upptäcka fusk i form av plagiat. Detekteringsverktyg är en intressant möjlighet, som behöver utredas närmare innan man kan bedöma hur användbara de är i praktiken.

References

[1] J. Biggs, *Teaching for Quality Learning at University*. Second Ed. Open Uniersity Press, 2003.

[2] J. Carroll. *A Handbook for Deterring Plagiarism in Higher Education*. Second Ed. Oxford Centre for Staff Learning Development, 2007.

[3] J.A.W. Faidhi and S.K. Robinson. “An empirical approach for detecting program similarity and plagiarism within a university programming environment”. *Computers & Education*, vol. 11(1), pp11-19, 1987.
[4] A. Parker and J.O. Hamblen. “Computer algorithms for plagiarism detection”. *IEEE Transactions on Education*, vol. 32(2), pp. 94-99, 1989.
[5] L. Prechelt, G. Malpohl, and M. Philippsen. “Finding plagiarisms among a set of programs with JPlag”. *Journal of Universal Computer Science*, vol. 8(11), pp. 1016-1038, 2002.
[6] S. Schleimer, D.S. Wilkerson, and A. Aiken. ”Winnowing: local algorithms for document fingerprinting”. *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp. 76-85, 2003.
[7] http://www.cs.lth.se/Education/LTH/fusk.shtml