Seminar zu Programmanalysen und Programmverstehen

Prof. Dr. Erhard Plödereder           Torsten Görg
Mandy Northover

1 Organisatorisches

Aktuelle Informationen finden Sie im Web:


**Vortrag**: Eine frühzeitige Absprache mit dem Themenbetreuer ist notwendig (mindestens einen Monat vor dem Vortragstermin).
Vortragsdauer von 30 Minuten mit anschließender Diskussion.
Präsentationsfolien mindestens eine Woche vor dem Vorstellungstermin mit dem Betreuer besprechen.

**Ausarbeitung**: Umfang: 8–10 Seiten einschließlich Zusammenfassung und Literaturverzeichnis im zweispaltigen IEEE-Format.
Vorläufige Abgabe eine Woche vor dem Vortrag.
Endgültige Abgabe zwei Wochen nach dem Vortrag (Nachbesserungen darüber hinaus sind nach Vereinbarung ggf. möglich).
Stilvorlagen für LaTeX und Word stehen auf der Web-Seite zur Verfügung.

**Anwesenheit**: Anwesenheit ist bei allen Veranstaltungen Pflicht. Krankheitsbedingtes Fehlen muss mit einem ärztlichen Attest entschuldigt werden.

**Befragung**: Die Gesamtnote setzt sich aus Noten für den Vortrag, der Ausarbeitung und der Beteiligung am Seminar zusammen, wobei Vortrag und Ausarbeitung hohe Anteile besitzen. Plagiate in den Vortragsfolien oder in der Ausarbeitung führen zu der Gesamtnote 5,0.
2 Themen

Cognitive Models: Bottom-up  Cognitive models are a means of abstracting and representing the mental processes of human beings in order to understand how people comprehend and solve problems. The concept of cognitive models originated historically in cognitive science but has subsequently been applied to various disciplines, including computer program comprehension. There are several types of cognitive models, the most well-researched of which are the so-called Bottom-up and Top-down models. This seminar topic considers Bottom-up models, which assert that programmers start by analysing the source code statements in detail, and then build mental groupings or “chunks” of these statements, which form high-level, easy-to-understand, abstractions of the detailed source code.

The so-called Systematic models are one example of the models that follow a Bottom-up approach, in which developers, who aim to gain a thorough understanding of the source code, do so by tracing the data flow through the program in order to comprehend global program behaviour. These models will also be considered in this seminar topic, and contrasted with As-needed models in the following seminar topic.

Literature for Bottom-up: [27, 30, 25, 31, 19]. Literature for Systematic: [21]

Betreuer: Mandy Northover
Termin: Termin 1: Montag, 19.05., 15:45, Raum 0.463

Cognitive Models: Top-down  A contrasting approach to that of the Bottom-up models is provided by the so-called Top-down models. Brooks [5] describes the Top-down process as a “hypothesis driven one in which an initially vague and general hypothesis is refined and elaborated based on information extracted from the program text and other documentation”. All Top-down models share in common the opportunistic search for “beacons” or “cues” (like procedure and variable names) in the source code to promote program comprehension. Unlike the Bottom-up models, there is no analysis of the source code statements line-by-line. Two specific models that use an opportunistic approach are the As-needed and Constructivist learning models.

As-needed models describe how developers focus only on a particular part of the source code relevant to a specific problem they are tasked with solving. The relevant part of the source code is identified opportunistically with the help of “beacons”.

Constructivist cognitive models take a subjective view to program comprehension, arguing that there is no objective knowledge independent of that constructed by the programmer. The foundation for program comprehension is a programmer’s knowledge-base. Programmers use different strategies to actively construct their own unique knowledge-base, which can be seen as an accumulation of as-needed or opportunistic knowledge.

**Betreuer**: Mandy Northover  
**Termin**: Termin 2: Montag, 26.05., 15:45, Raum 0.463

**Program Comprehension in Practice** Several theoretical approaches have been proposed to support program comprehension in an automatic or semi-automatic way. Recent studies have shown that there is a gap between these theories and what developers really do in practice to understand some given source code [29]. It turns out that many developers do not use tools for program comprehension at all or just use simple tools like grep. They usually prefer to ask a colleague if they need detailed information. This topic views the reality of program comprehension in practice and also takes a look at the importance of collaboration and social networks beside technical approaches to solving problems.

Literature: [29, 32, 16, 28]  
**Betreuer**: Torsten Görg  
**Termin**: Termin 3: Montag, 02.06., 15:45, Raum 0.463

**Design Recovery: Software Clustering** Design recovery is a reverse engineering technique which attempts to recreate the design abstractions of a legacy software system in order to promote program comprehension. Many techniques for design recovery exist, but this seminar will focus specifically on a technique known as software clustering. Clustering splits a system up into smaller, more comprehensible subsystems by searching for potential clusters using either well-known criteria in the software such as high cohesion and low coupling or commonly occurring patterns. Other techniques use task-driven output such as transaction logs or co-change graphs or module dependency graphs to identify subsystem candidates. A specific use of software clustering is in identifying high-level models as part of software reflexion.

Literature: [12, 18, 37, 8, 22, 4, 24]  
**Betreuer**: Mandy Northover  
**Termin**: Termin 4: Montag, 16.06., 15:45, Raum 0.463

**Feature Localization** For maintenance, it is important to know which parts of the source code realize a particular program feature. Feature localization provides techniques that reconstruct a mapping between source code and features that are triggered by the user and exhibit an observable behaviour. These techniques are either based on static, on dynamic, or on hybrid program analyses. Eisenbarth, Koschke, and Simon propose a hybrid approach that use concept analysis - a mathematical technique to investigate binary relations [11]. Other authors provide extended techniques to address multi threaded systems [1] and distributed systems [9].

Literature: [11, 1, 9, 26]  
**Betreuer**: Torsten Görg  
**Termin**: Termin 5: Montag, 23.06., 15:45, Raum 0.463
**Visualization Tools** The strength of visual representations in aiding human comprehension is well-documented by cognitive science. It is therefore common that the various program comprehension models we have discussed so far are supported by visualization tools. The primary goal of visualization tools is to ease comprehension in practice, and to this end, various non-functional comprehension criteria exist (like legibility, simplicity, scalability and clarity) as well as various visualization views (like radial views, tree views, fish-eye views, and 3D views). Furthermore, the tools are often either categorized as static (since they analyse the source code) or dynamic (since they analyse the behaviour of the running program). In this seminar, we will consider two static software visualization tools – *SeeSoft* along with its 3D extension called *sv3D*, and *SHriMP* (Simple Hierarchical Multi-Perspective) along with its Eclipse extension called *Creole*. As the predominate approach for visualisation is based on graphs, we will also consider *Graph eXchange Language (GXL)* – a standard format for interoperability between graph-based tools.

Literature: [10, 23, 35, 36, 20, 14]

**Betreuer:** Mandy Northover  
**Termin:** Termin 6: Montag, 30.06., 15:45, Raum 0.463

**Program Slicing** During maintenance, software developers have to do modifications on software systems that may consist of a huge quantity of source code and that is usually not originally written by themselves. It is not easy to avoid new bugs getting into the code by these modifications. Program slicing is a means to calculate all influences introduced by a modification. It follows all paths that propagate data from the modified code to other instructions (forward slice). Program slicing can also be used to find the origins of data paths that influence the value of a variable at a particular location in the code (backward slice). This provides a systematic approach to analyse unexpected values calculated during the execution of a program.

Literature: [15, 38, 33]

**Betreuer:** Torsten Görg  
**Termin:** Termin 7: Montag, 07.07., 15:45, Raum 0.463

**Code Clones** Program source code often contains some high level concepts that appear as cross-cutting concerns, e.g., error handling, range checking, logging, and tracing. As the code for a particular concern follows repeated patterns, code clone detection techniques can identify cross-cutting concerns [6]. Code clone detection investigates identical or similar fragments in some given source code. Several approaches for clone detection have been developed. Most important are token-based [2], AST (abstract syntax tree)-based [3], metric-based, and PDG (program dependency graph)-based techniques.

Literature: [6, 2, 3]

**Betreuer:** Torsten Görg  
**Termin:** Termin 8: Montag, 14.07., 15:45, Raum 0.463
Literatur


[12] **Eisenbarth, Thomas; Koschke, Rainer; Plödereder, Erhard; Girard, Jean-Francois; Würthner, Martin**: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen. In: *1. Workshop Software Reengineering Bd. 7*, Bad Honnef: Universität Koblenz-Landau, Mai 1999 (Fachberichte Informatik), 17–26


7


