SoftVarE – Thesis Topics of Paul Bittner
About Me

My name is Paul

2014 – 2020: Study Computer Science in Braunschweig

*Since then*: PhD student at SP in Ulm

My research is about helping people to

... find their code,

... keep their forks and branches synchronised,

... benefit from product-line technology with less effort.
What Do We Offer?

• weekly meetings
• reviews for each part of your thesis before submission
• thesis topics relevant to research
• possibility of participating in future publications
WHICH PART OF THE CODE IS IMPLEMENTING THAT?
Feature Traceability is the knowledge where each feature is implemented.
Software Product Lines

Feature-Modell

Domain Eng.

Feature-Auswahl

Application Eng.

Wiederverwendbare Implementierungs-artefakte

Generator

Fertiges Program
Traceability is given in \textit{software product lines} ... but in practice variability is often implemented by copying (or branching) and adapting existing code.

So how can we help developers to \textit{document} and \textit{maintain} feature traces here?
Clone-and-Own as Prominent Variability Approach
Clone-and-Own as Prominent Variability Approach

Variant 1 -> Variant 2 -> Variant 3 -> Duplicate Code

git branch

Variant 2

Bug

which part of the code is implementing that?
Topics on Feature Traceability
Commits are rarely disciplined: Multiple changed are often mixed in one commit. Can we unmix commits? Can feature traceability help?

**Tasks:**
- Literature survey: Which unmix methods exist? How do they work? How do they differ?
- Extend existing methods with feature traceability or design your own method.
- Implement method(s) to empirically evaluate them on existing git repositories (for example Linux).

```c
void jump() {
    - velocity.y -= JUMP_ACC;
    + velocity.y += JUMP_ACC;
    
    #ifdef GAMEPAD_SUPPORT
    - gamepad.vibrate(0.1f);
    + gamepad.vibrate(VIB_SEC);
    #endif
}
```

bugfix
code hygiene
Task-Oriented Commits

Today's standard for tracking source code evolution are version control systems such as Git and Subversion. These systems document edits to the code base in commits. Although intended in version control, commits are rarely disciplined in the sense that each commit captures exactly one change to a feature. Sometimes, even huge refactorings or renamings affecting the entire code base occur in a single commit. Contrary, research and tooling often require finer-grained edits. Yet version control systems are used the most in practice, so tools and research should be able to work with version control systems. We thus need an operator to divide an arbitrary-sized commit into feature-oriented commits.

Related Work: [2], [3]
**Sending Feature Traces Back in Time**

1. Some feature mapping is retrieved in the present.
   ```c
   void jump() {
     #ifdef GAMEPAD_SUPPORT
     gamepad.vibrate(0.1f);
     #endif
   }
   ```

2. Find last edit to mapped code (e.g., with git blame).

3. Introduce feature mapping in the past.
   ```c
   void jump() {
     #ifdef GAMEPAD_SUPPORT
     gamepad.vibrate(0.1f);
     #endif
   }
   ```

4. Run Feature Trace Recording to send feature mapping
   - broadcast, accumulate, refine more knowledge

---

**Topic for M**

- Sending Feature Traces Back in Time

Feature location methods recover lost or unknown feature mappings in an existing code base manually, semi-automatically, or automatically. Feature location methods may use some existing feature mappings as input. In case few or no feature mappings are known, can we assist tools with feature mappings from the future?

**Task**: Develop and implement a method for sending feature mappings into the past.

**Questions**:
- How beneficial is sending feature mappings back in time?
- Can we compute further feature mappings by sending them through time?
- How far can we send feature mappings back in time (at most, on average, at least)?
- Does traveling back to the future yield more or more accurate feature mappings?
Synchronizing and Merging Feature Mappings

Variant A

```c
void jump() {
    velocity.y += JUMP_ACC;
#if GAMEPAD_SUPPORT
    gamepad.vibrate(VIB_TIME);
    + #if GAMEPAD_WITH_SOUND
    +  gamepad.sound(JUMP_SOUND);
    + #endif
    #endif
}
#endif
#if GAMEPAD_SUPPORT && GAMEPAD_VIBRATION
    gamepad.vibrate(VIB_TIME);
    #if GAMEPAD_WITH_SOUND
    gamepad.sound(JUMP_SOUND);
    #endif
    #endif
```

Variant B

```c
void jump() {
    velocity.y += JUMP_ACC;
    -#if GAMEPAD_SUPPORT
    +#if GAMEPAD_SUPPORT && GAMEPAD_VIBRATION
        gamepad.vibrate(VIB_TIME);
    #endif
    }
```

I have to propagate my edit to Alice’s variant.

Bug: No sound without vibration anymore!
With VariantSync, developers develop multiple variants (e.g., multiple branches or forks of a project) of a software project. Thereby, VariantSync harvests feature traces by recording or mining them, or through explicit annotations by developers. Next to source code changes, also feature mappings have to be synchronized when propagating changes made to a source variant to other target variants. You may find out more about VariantSync on our [website and our introduction video](#). When propagating edits made to a variant to another variant, we may get merge conflicts. Apart from merge conflicts on source code, also feature mappings may be conflicting. It might even happen that an edit to source code can be propagated without problems but the feature mappings of the edited code differ. We thus need a reliable method for synchronizing and merging possibly conflicting feature mappings.

**Goal:** Develop, implement, and evaluate a model to synchronize edits to feature mappings.

**Questions:**
- Which merge conflicts on feature mappings may occur (next to merge conflicts on code)?
- How to resolve merge conflicts? Which conflicts can be resolved automatically?
- What are differences and benefits of state-based vs. edit-based synchronization?
Sending Feature Traces into the Future
with Feature Trace Recording

Topic for B / P

void pop() {
    storage[head--] = null;
}

void pop() {
    if (!empty()) {
        storage[head--] = null;
    }
}

Stack<T> pop() {
    Stack<T> c = clone();
    if (!empty()) {
        c.storage[c.head--] = null;
    }
    return c;
}

insert
correct: SafeStack

move
correct: null

delete
correct: ImmutableStack

context: ImmutableStack
Feature trace recording is a feature location method that monitors source code evolution. Feature traces are recorded while developers are programming (instead of recovering feature traces retroactively as variability mining does). For example, in the image above you see how edits are made to an initial version of the pop method of a Stack in Java. Upon each edit, developers specify which feature they are editing (which we call the *feature context*). Feature mappings, indicated by coloured source code) are then recorded upon the edit. We thus know to which feature the edited code belongs to.

Currently, feature trace recording was not evaluated on large commit history history so its benefits are mostly unexplored. In particular its capability of keeping feature mappings consistent across evolution as well as gathering further feature mappings over the course of evolution could be promising to gradually increase developers' knowledge on feature traceability. Moreover, feature trace recording is subject to various parameters such as quality of the feature context, how often a feature context is specified, or the granularity of the observed edits.

**Tasks:**
- Implement feature trace recording into our VariantSync framework.
- Generate variants and edit histories from software product lines (for example Linux).
- Run feature trace recording on the generated histories.
- Evaluate quality of the recorded feature traces