Aspect-oriented programming (AOP)

- Typically extension of object-oriented language
- Aims to separate crosscutting concerns into their own modules, i.e. aspects

Crosscutting concerns:
- Scattered in several places
- Tangled with other concerns
- Examples: logging, authentication, caching, profiling, ...
aspect Logging {
    after: call(* Bank.do*(..)) {
        log.write(stuff);
    }
}
AOP’s inherent problem

- Pointcuts are an **implicit** mechanism
  - Classes (and aspects!) are **not aware** that their behaviour can be modified by aspects
  - May lead to unexpected/undesired behaviour as the system evolves
Cornering the problem

- **Goal:** Detect undesired behaviour caused by aspects

- **Data-flow analysis:**
  - Automatically detect interactions caused by aspects
  - Present them to the developer

- **Design by contract:**
  - Distinguish harmless from (potentially) harmful aspects
Data-flow analysis

- Expose interactions caused by aspects, both to aspect and class developers

- Focus on data dependencies:
  - What data is being modified by advice?
  - Who uses this modified data?
aspect CloseStreamAfterUse {

    after: execution(Export.write*(stream)) {

        stream.close();

    }
}

Someone unaware of the aspect might still use the stream later on.
Advice impact analysis

- Based on **definition-use analysis**
  - Find all definitions (assignments) during the execution of advice which may affect the base system (or other aspects)
  - .. as well as their corresponding uses.

- Implemented on top of the AspectBench Compiler
  - Uses AST + call graph generated from (woven) Soot representation
Incremental updates

- Analysis aimed to run in the background of an IDE
- Determine what to invalidate and reanalyse per type of source code change
- Evaluated by performing a large set of random (though realistic) changes on existing systems
Design by Contract (DbC)

- DbC, an intuitive and modular means to specify desired behaviour of an application

- In OOP, the Liskov substitution principle:
  - Preconditions: must be eq. or weaker in subtype
  - Postconditions: must be eq. or stronger
  - Invariants: must be preserved

- What if you add aspects?
Advice substitutability

- Considering all advice are around advice:
  Whenever a pointcut matches, the corresponding advice effectively substitutes the original join point.

- Liskov substitution can be easily adjusted to aspects.
  - The contracts of an advice have to comply with the contracts of the join point where it is applied.
Relation to obliviousness

- If the advice substitution principle is strictly applied, the base system can remain unaware of aspects.

- However, some aspects have to break the principle:
  - e.g. authentication: Postcondition is weakened if the user is not logged in

- Principle not meant to be strictly enforced; it simply defines the border between safe/unsafe obliviousness
Where DFA and DbC meet

- Data flow analysis + Design by contract:
  Detect which interactions are desired or undesired
Detecting (un)desired interactions

- Given an interaction (definition+use), will the altered value cause a contract violation where the new value is used?
- If the original code implements its contracts, does this still hold with the altered value?
- Look into ESC/Java2, Spec#, ...
Summary

- **Aim:** Detect undesired behaviour caused by aspects

- **Data-flow analysis:**
  - Detect interactions caused by aspects

- **Design by contract:**
  - Advice subst. principle rejects aspects with undesired behaviour

- **DFA+DbC:**
  - Detect (un)desired interactions by aspects