Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa

Rudolf Ferenc[1]
Juha Gustafsson[2]
László Müller[1]
Jukka Paakki[2]

[1] Research Group on Artificial Intelligence, University of Szeged
[2] Department of Computer Science, University of Helsinki

Introduction

- Experiments with two software tools:
  - MAISA - metrics tool
  - Columbus - reverse engineering tool
- Key applications:
  - Support for the assessment of the software development process
  - Studying existing systems
  - Utilizing different levels of abstraction
  - Visualization of software architectures

Maisa – background

- Early quality assurance is effective, yet automated support is little researched
- Predictions based on historical data
- Metrics calculated by MAISA can be used as input for quality evaluation methods such as GQM
- MAISA can be used together with other software tools such as Columbus so the whole software life cycle can be covered

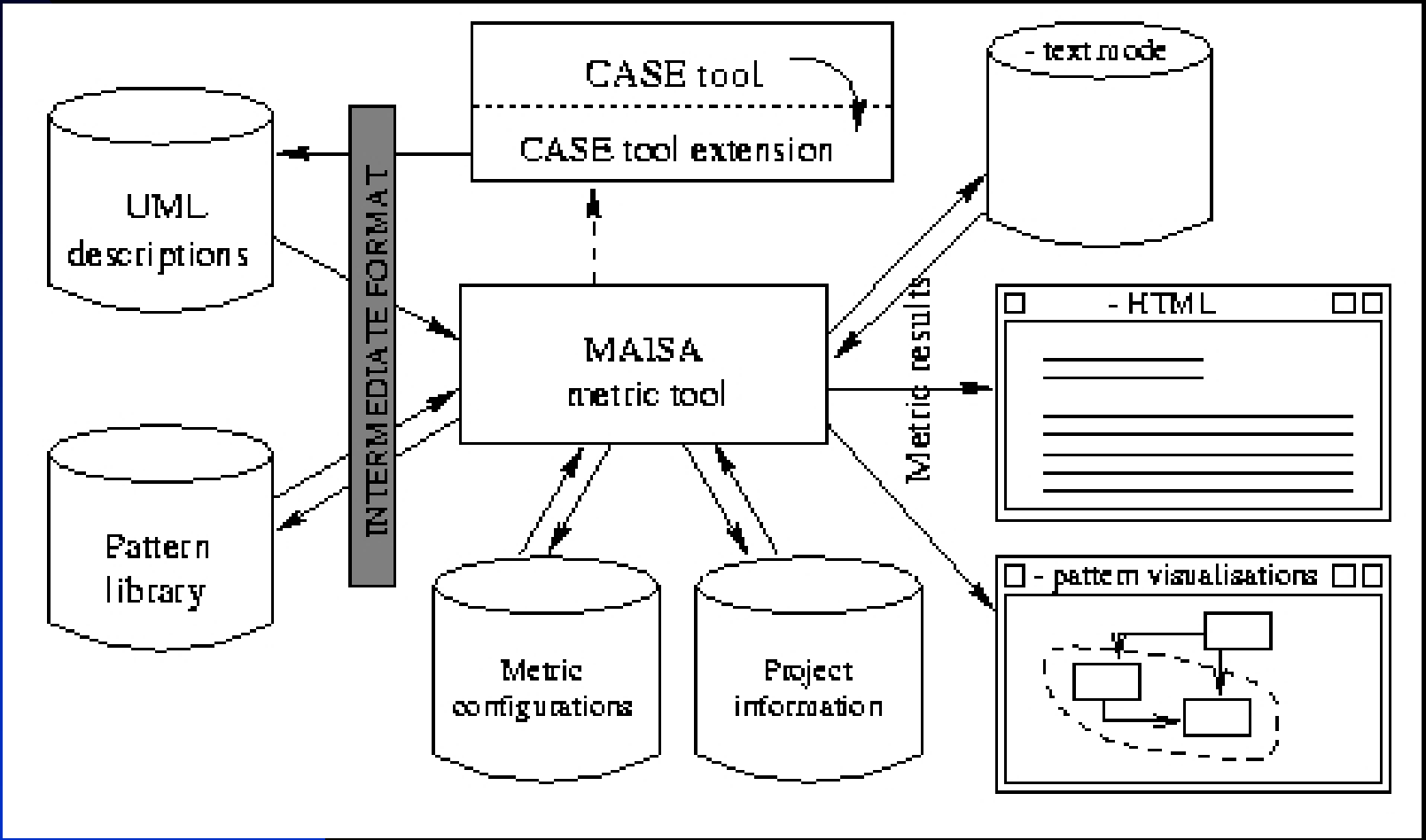
Maisa tool

- Has both GUI and a command line interface
- Three key concepts:
  - Project - a set of related UML diagrams
  - Metric configuration - a set of metrics with respective threshold values
  - Pattern library - an extendible collection of pattern data
- Data saved as text for easy access

Maisa tool (cont.)

- Several types of measurements:
  - basic metrics: *What is the McCabe complexity of this diagram?*
  - pattern based metrics: *What percentage of classes participate in design patterns in this product?*
  - pattern detection: *Are there any instances of the Factory Method pattern in this diagram?*

Maisa architecture



Pattern detection in MAISA

- Pattern recognition is defined as a constraint satisfaction problem (CSP)
- Efficient search requires pruning which is currently done with the AC-3 algorithm
  - e.g. if we can only assign object of type class to a particular role, then we can prune attributes, methods etc. from the candidates
- Although class diagrams are most common, any type of diagram can be used

Pattern detection (cont.)

- The variables represent the roles of a pattern
- The variable domains are initialized to contain the identifiers in the diagram(s) in question
- Unary constraints represent conditions for a single role
  - the element in role X must be an abstract class
- Binary constraints represent conditions between two roles
  - the class in role X must be a subclass of the class in role Y

Columbus

- Tool for *reverse engineering* large object oriented systems
- Fast, fault tolerant C++ parser
- Extracts the *UML class diagram* and the *call graph*
- Supports project handling, data extraction, data representation and data storage
- Various output formats
- Extensible with *plug-in*-s

Integration of Columbus and Maisa

- Columbus offers a *C++ API* for accessing the *ASG* of the analyzed system
- Maisa is implemented in *Java*
- Solution: Columbus exports the needed information into Maisa's input file format
- The file contains the reverse engineered information in a PROLOG-like format, as facts over the main program elements (classes, attributes, etc.) and their relationships (subclassing, etc.)

Experiments

- Singleton pattern Implementation:

```
class MySingleton {
  public:    static MySingleton* getInstance();
  protected: MySingleton() {};
  private:   static MySingleton* instance;
};

MySingleton* MySingleton::instance = 0;

MySingleton* MySingleton::getInstance() {
  if (instance == 0) {
    instance = new MySingleton();
  }
  return instance;
}
```

Experiments (cont.)

- Columbus's output:

```
class("MySingleton").
method("MySingleton.getInstance()").
public("MySingleton.getInstance()").
static("MySingleton.getInstance()").
has("MySingleton","MySingleton.getInstance()").
returns("MySingleton.getInstance()","MySingleton").
method("MySingleton.MySingleton()").
protected("MySingleton.MySingleton()").
has("MySingleton","MySingleton.MySingleton()").
attribute("MySingleton.instance").
private("MySingleton.instance").
static("MySingleton.instance").
has("MySingleton","MySingleton.instance").
typeof("MySingleton.instance","MySingleton").
```

Experiments (cont.)

- Maisa's Singleton specification:

```
class("Singleton").
attribute("Singleton.instance").
has("Singleton","Singleton.instance").
typeof("Singleton.instance","Singleton").
static("Singleton.instance").
```

- Maisa's output:

```
Solution 0
  Singleton.instance = MySingleton.instance
  Singleton = MySingleton
```

Experiments – Summary

Pattern name	Description	Missing facts
Singleton	Ensures that a class has only one instance	—
Visitor	Represents an operation on the elements of an object structure	—
Builder	Separates the creation of a complex object from its representation	reads(method,attribute) writes(method,attribute)
Factory Method	Defines an interface for creating subclass-specific objects	—
Prototype	Creates objects by cloning prototypical instances	—
Proxy	Provides a placeholder for an object to control access to it	reads(method,attribute)
Memento	Captures the state of an object	—

Conclusion

- Initial experiments on the integration of Columbus and MAISA are promising
- The importance of dealing with different levels of abstraction
- Interaction with the user is essential
- Open questions include:
  - dealing with other types of patterns
  - extracting and utilizing dynamic information such as sequence diagrams