CS3213
Architecture Overview

- ❑ Sample Workflow
- ❑ Components in the its-core
- ❑ Architectural Styles

Sample Workflow

The next slides show the intended workflow through the *Intelligent Tutoring System (ITS)*. Note that there are points of variation (static and dynamic) that depend, e.g., on the programming language of the programming assignments and the intended repair strategies. Many of the current components can be implemented in many different ways.

Workflow (Overview)

Intelligent Tutoring System

Reference Program

Submitted Program

Teacher/ Tutor

Student (Novice Programmer)

The *Intelligent Tutoring System (ITS)* uses the notion of a ***pipes-and-filter*** style to process the submitted program, and finally, to produce feedback for the students and a grading report for the tutor.

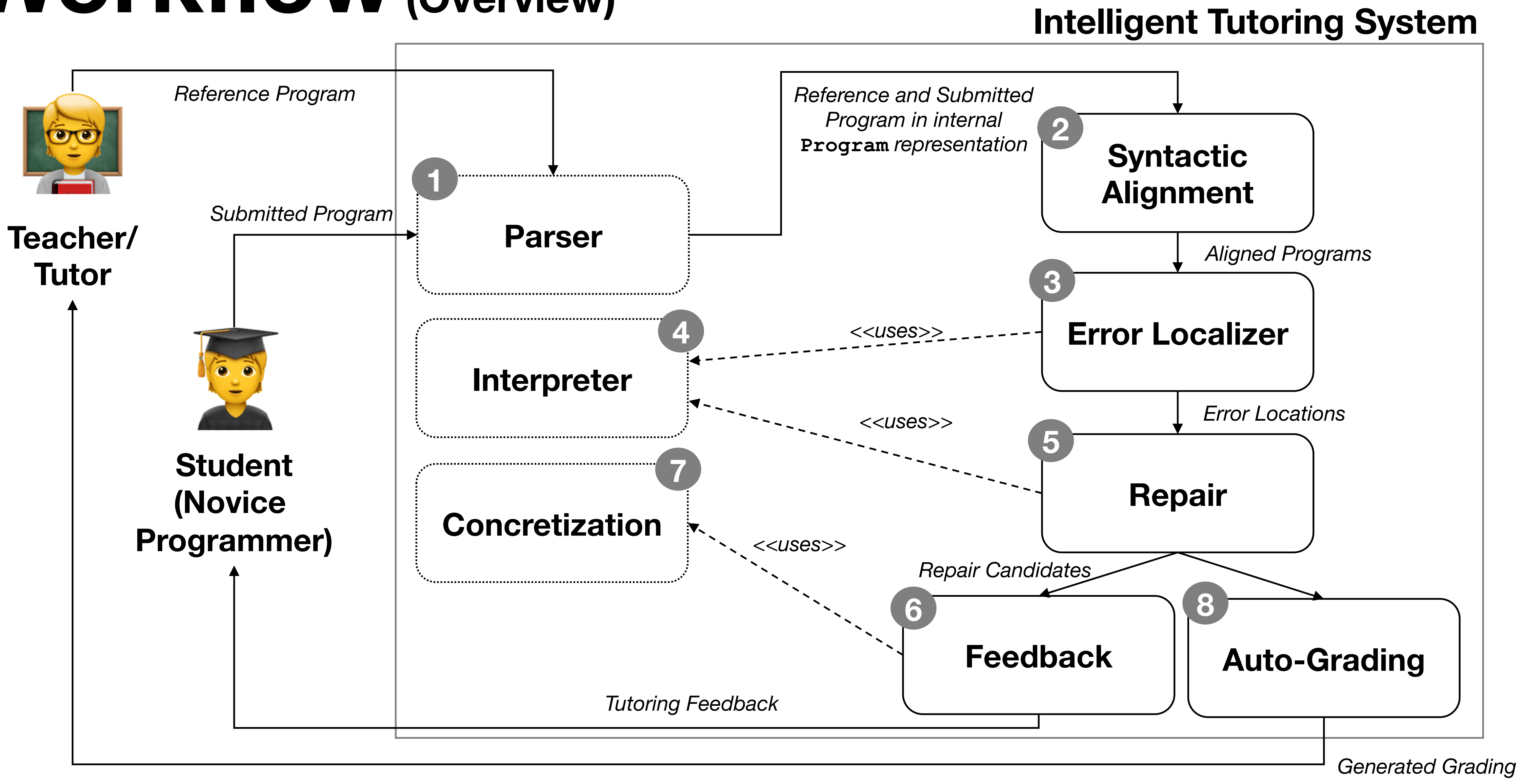
All components provide corresponding interfaces to an extent that most components can be implemented independently from each other.

This architecture also allows us to deploy certain components of the system as separate *services*. In this regard, the design borrows concepts from the ***service-oriented*** style. For example, we will later give you access to a *Parser* service that allows you to easily generate the internal CFG-based **`Program`** representation to create test cases for your projects.

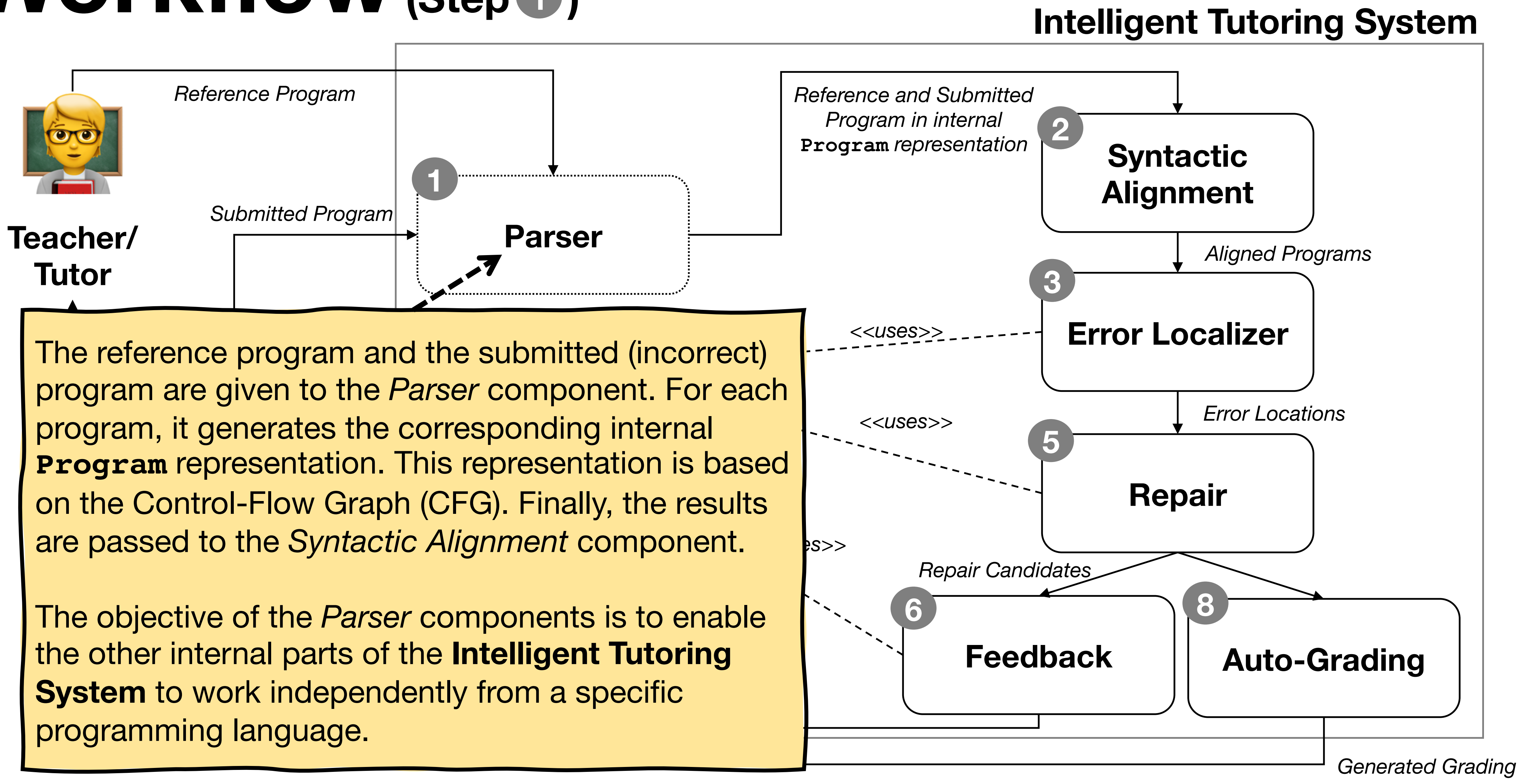
Tutoring Feedback

Generated Grading

Workflow (Overview)



Workflow (Step 1)



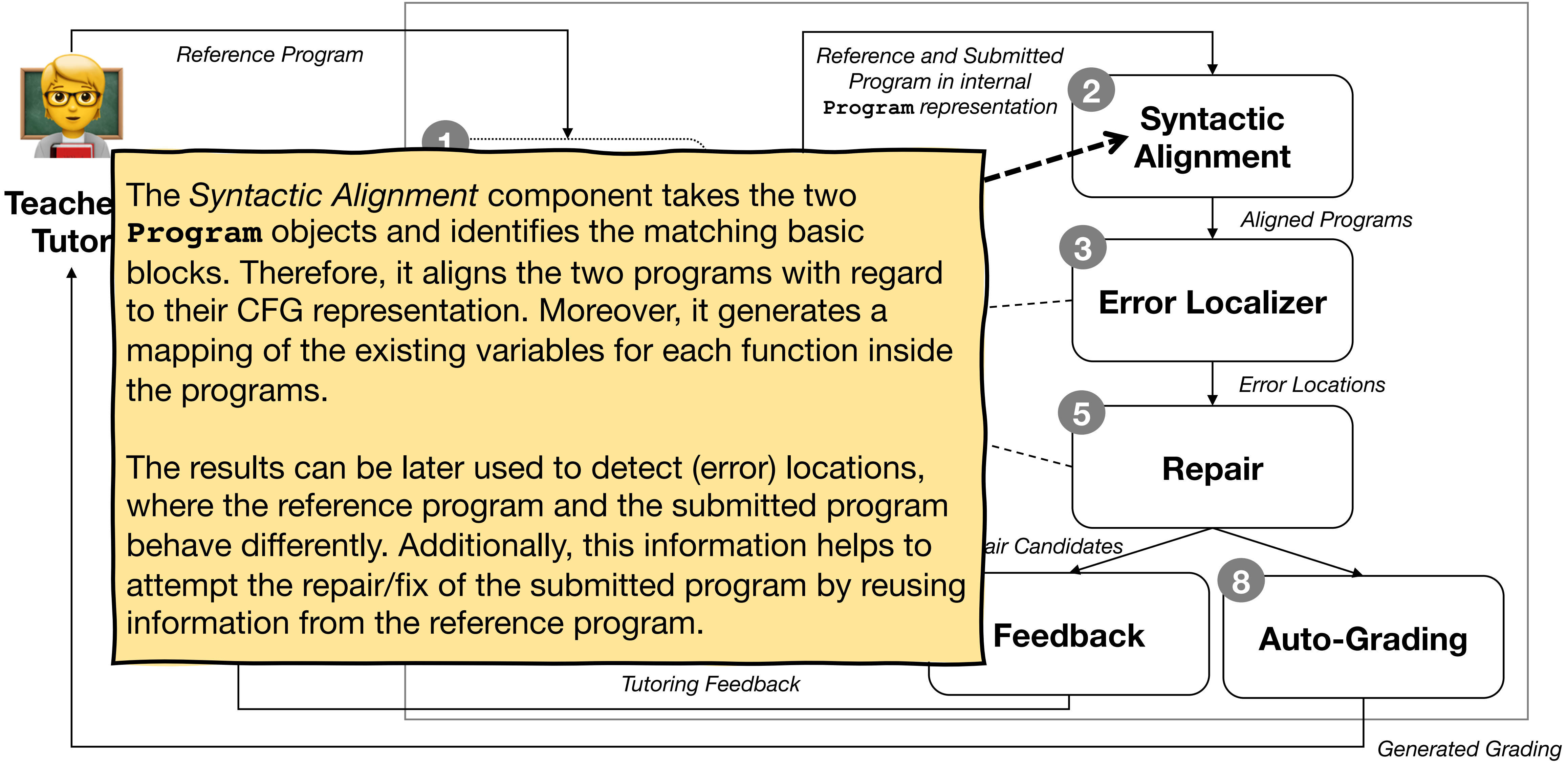
The reference program and the submitted (incorrect) program are given to the *Parser* component. For each program, it generates the corresponding internal **`Program`** representation. This representation is based on the Control-Flow Graph (CFG). Finally, the results are passed to the *Syntactic Alignment* component.

The objective of the *Parser* components is to enable the other internal parts of the **Intelligent Tutoring System** to work independently from a specific programming language.

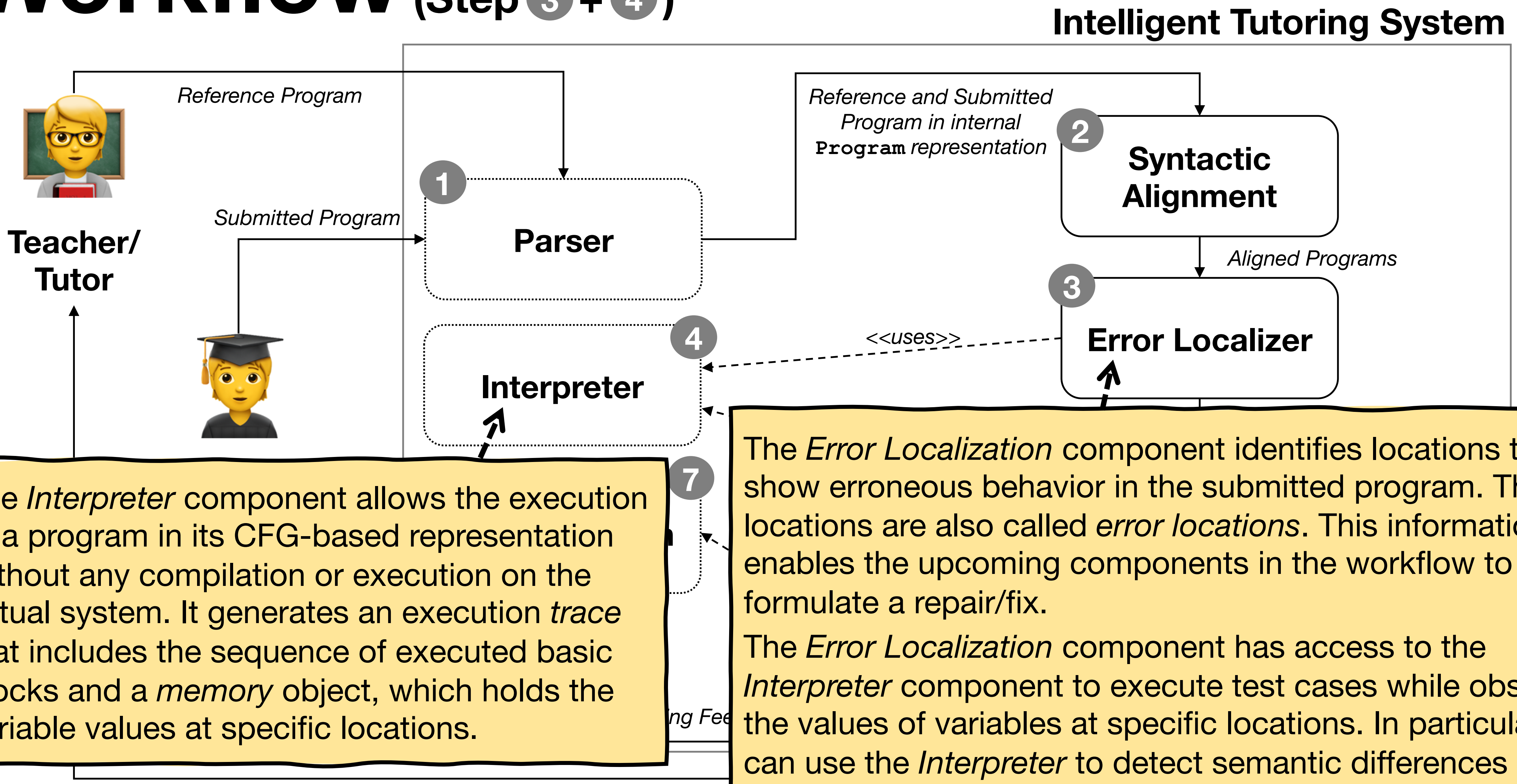
Workflow (Step 2)

The *Syntactic Alignment* component takes the two **`Program`** objects and identifies the matching basic blocks. Therefore, it aligns the two programs with regard to their CFG representation. Moreover, it generates a mapping of the existing variables for each function inside the programs.

The results can be later used to detect (error) locations, where the reference program and the submitted program behave differently. Additionally, this information helps to attempt the repair/fix of the submitted program by reusing information from the reference program.



Workflow (Step 3 + 4)

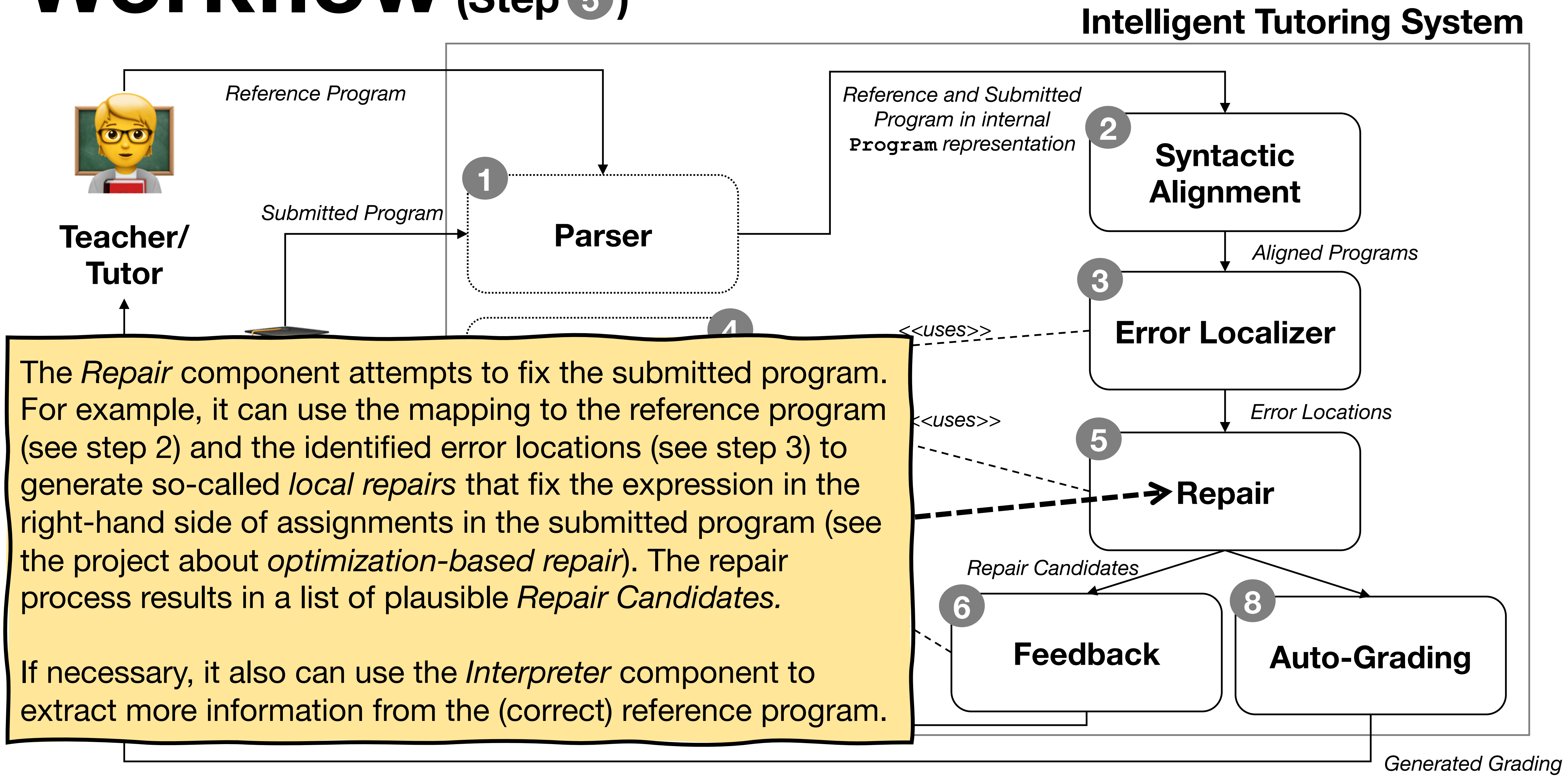


The *Interpreter* component allows the execution of a program in its CFG-based representation without any compilation or execution on the actual system. It generates an execution *trace* that includes the sequence of executed basic blocks and a *memory* object, which holds the variable values at specific locations.

The *Error Localization* component identifies locations that show erroneous behavior in the submitted program. These locations are also called *error locations*. This information enables the upcoming components in the workflow to formulate a repair/fix.

The *Error Localization* component has access to the *Interpreter* component to execute test cases while observing the values of variables at specific locations. In particular, it can use the *Interpreter* to detect semantic differences between the reference program and the submitted program.

Workflow (Step 5)



The *Repair* component attempts to fix the submitted program. For example, it can use the mapping to the reference program (see step 2) and the identified error locations (see step 3) to generate so-called *local repairs* that fix the expression in the right-hand side of assignments in the submitted program (see the project about *optimization-based repair*). The repair process results in a list of plausible *Repair Candidates.*

If necessary, it also can use the *Interpreter* component to extract more information from the (correct) reference program.

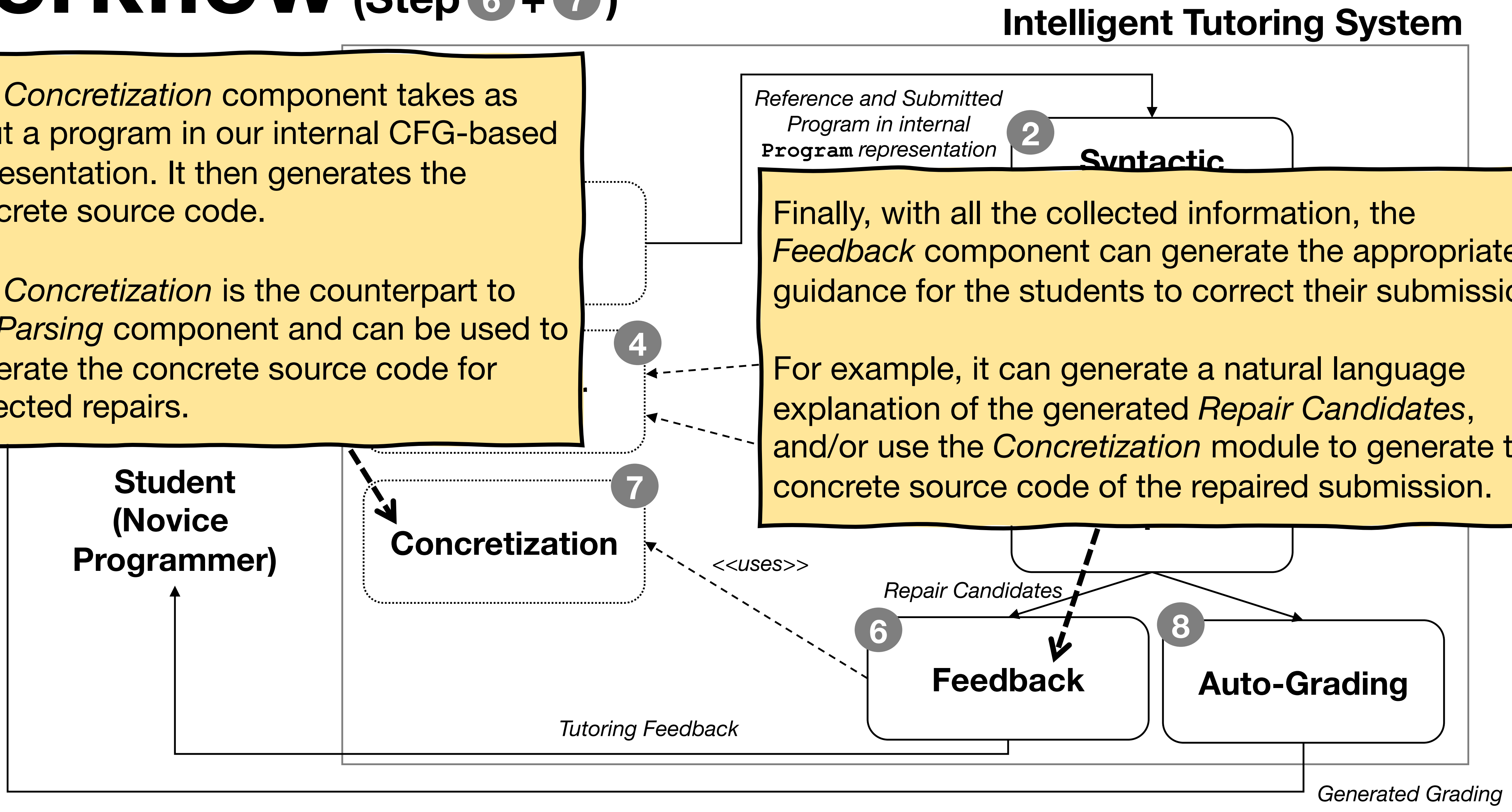
Workflow (Step 6 + 7)

The *Concretization* component takes as input a program in our internal CFG-based representation. It then generates the concrete source code.

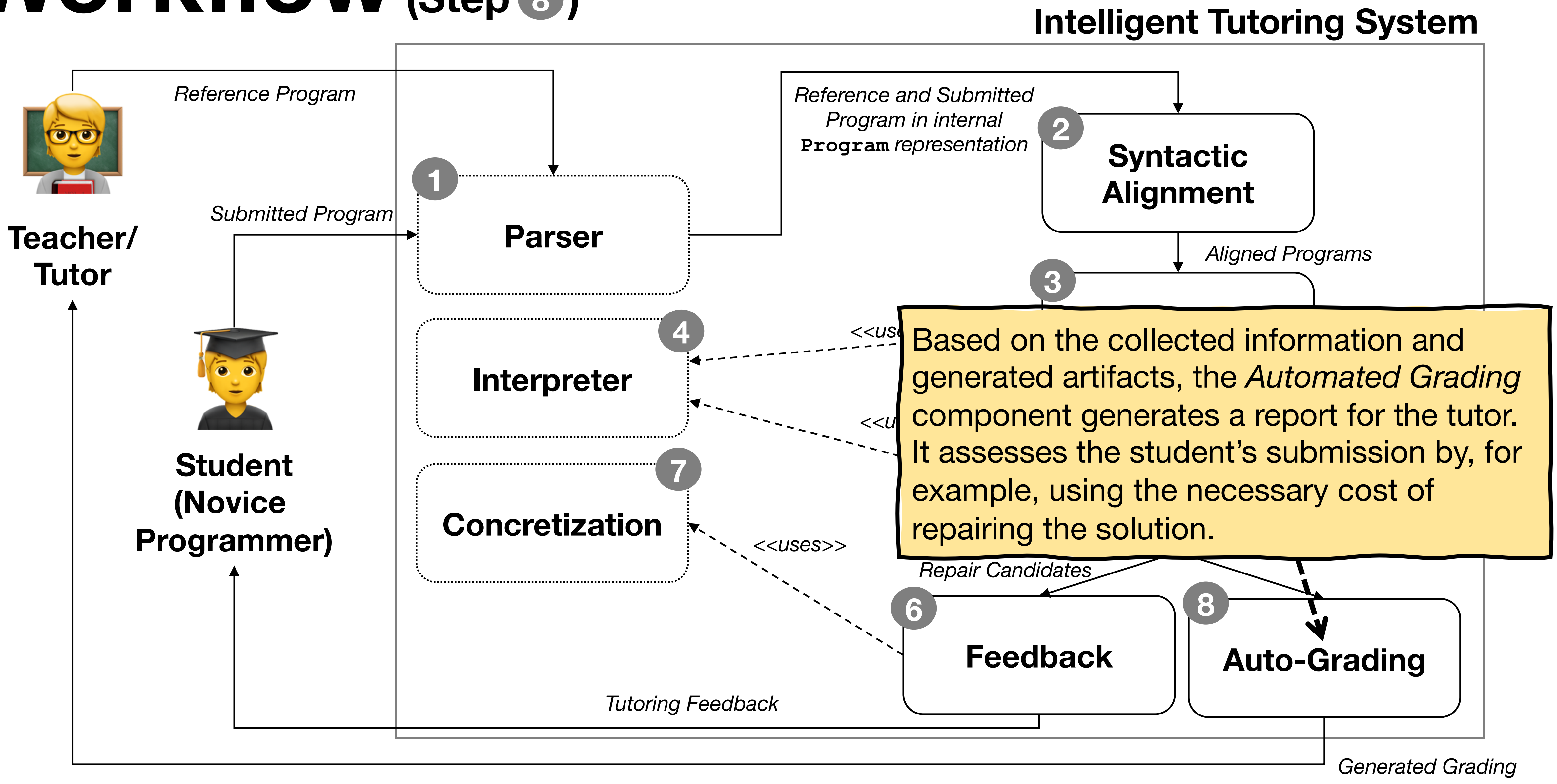
The *Concretization* is the counterpart to the *Parsing* component and can be used to generate the concrete source code for collected repairs.

Finally, with all the collected information, the *Feedback* component can generate the appropriate guidance for the students to correct their submission.

For example, it can generate a natural language explanation of the generated *Repair Candidates*, and/or use the *Concretization* module to generate the concrete source code of the repaired submission.



Workflow (Step 8)

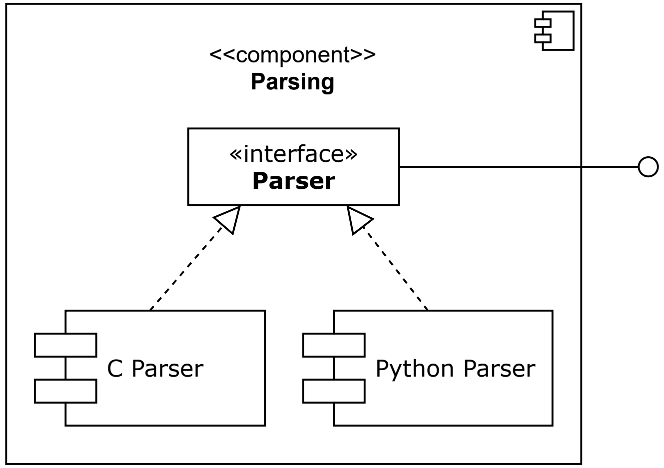


Based on the collected information and generated artifacts, the *Automated Grading* component generates a report for the tutor. It assesses the student's submission by, for example, using the necessary cost of repairing the solution.

Components in the context of the its-core baseline

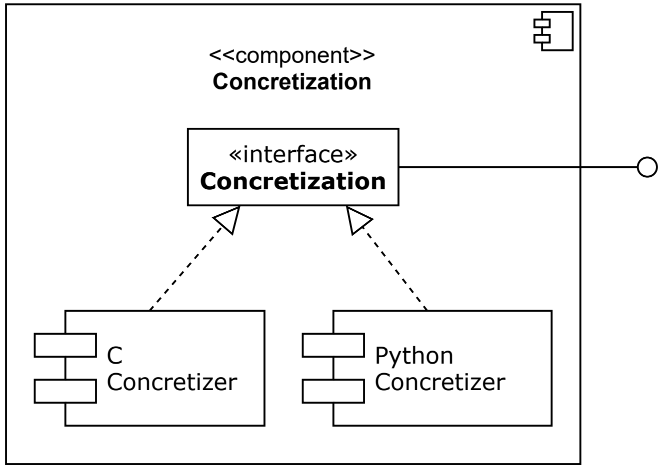
The next slides show more details about the components and their interfaces. For your projects, you will get access to the **`its-core`**, which includes all interfaces, common data structures, and some utility functions.

Components (1/4)



its-core ▸ src/main/java ▸ sg.edu.nus.se.its.parser ▸ Parser ▸

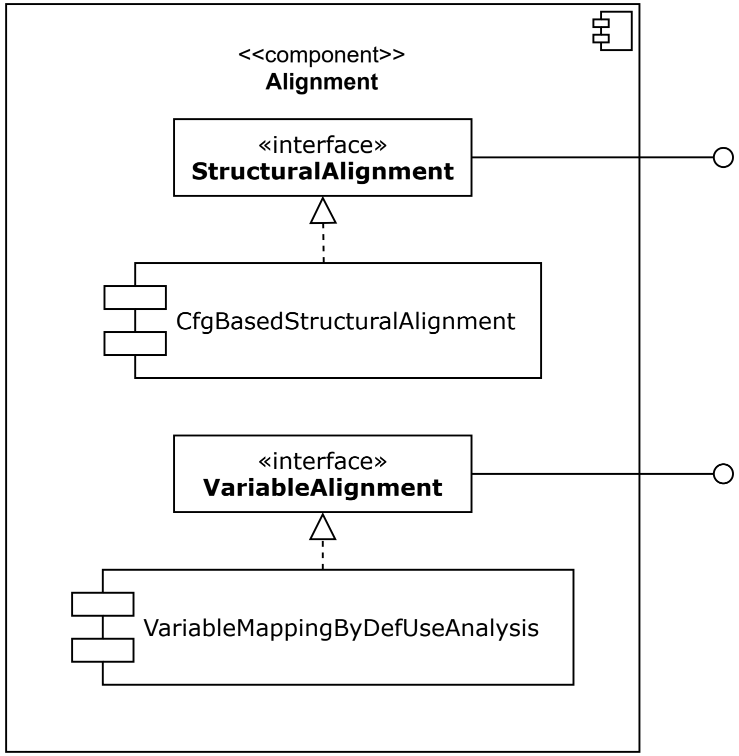
```java
package sg.edu.nus.se.its.parser;

import java.io.File;

/**
 * Interface for parsing a source code file into the internal data structure representation.
 */
public interface Parser {

  /**
   * Parses the program source code and returns the program in form of the internal object
   * representation.
   *
   * @param filePath - the path to the program text file.
   * @return the internal representation of the program source code.
   * @throws IOException if the file does not exist or could not be parsed.
   */
  public Program parse(File filePath) throws IOException;

}
```



its-core ▸ src/main/java ▸ sg.edu.nus.se.its.concretization ▸ Concretization ▸

```java
package sg.edu.nus.se.its.concretization;

import java.io.File;

/**
 * Interface for the concretization of Program objects.
 */
public interface Concretization {

  /**
   * Returns the file object of the repaired program in the internal representation.
   *
   * @param repairedProgram - the repaired program in its internal representation.
   * @return the source code representation of the given Program object
   */
  public File concretize(Program repairedProgram, String fileName) throws Exception;

}
```

Components (2/4)



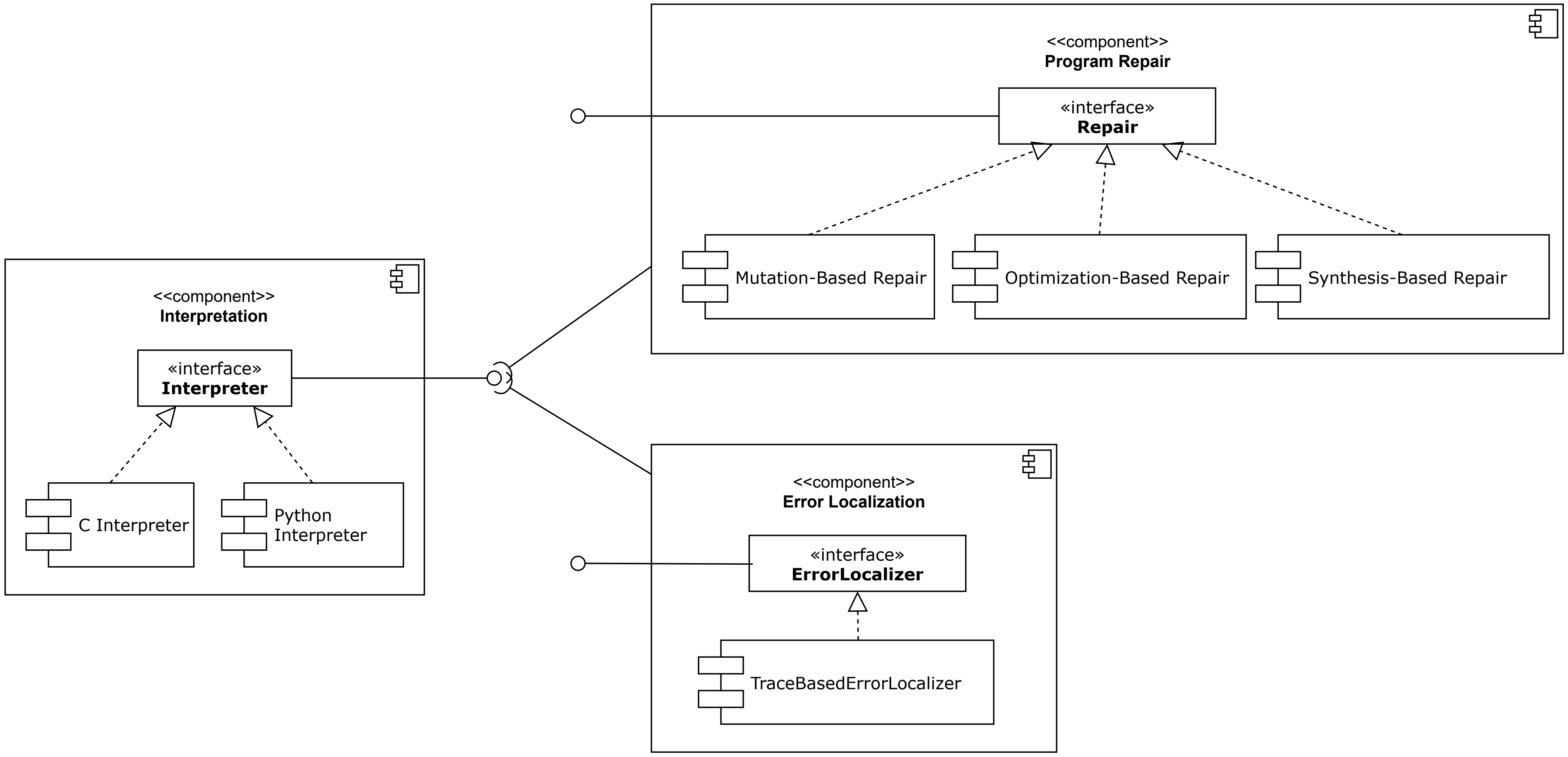
its-core ▸ src/main/java ▸ sg.edu.nus.se.its.alignment ▸ StructuralAlignment ▸

```java
package sg.edu.nus.se.its.alignment;

import sg.edu.nus.se.its.exception.AlignmentException;

/**
 * Interface for the syntactical alignment of the reference program and the submitted program.
 */
public interface StructuralAlignment {

  /**
   * Generates the structural alignment for the given two programs.
   *
   * @param reference - the reference program
   * @param submission - the submitted/incorrect program
   * @return the structural mapping
   * @throws AlignmentException - a custom alignment exception
   */
  StructuralMapping generateStructuralAlignment(Program reference, Program submission)
      throws AlignmentException;
}
```

its-core ▸ src/main/java ▸ sg.edu.nus.se.its.alignment ▸ VariableAlignment ▸

```java
package sg.edu.nus.se.its.alignment;

import sg.edu.nus.se.its.exception.AlignmentException;

/**
 * The interface for any variable alignment between two structural aligned program.
 */
public interface VariableAlignment {

  /**
   * Generates variable alignment.
   *
   * @param reference - the reference program
   * @param submission - the submitted/incorrect program
   * @param strucAlignment - the structural alignment
   * @return the resulting variable alignment
   * @throws AlignmentException - a custom alignment exception
   */
  VariableMapping generateVariableAlignment(Program reference, Program submission,
      StructuralMapping strucAlignment) throws AlignmentException;
}
```

Components (3/4)



Components (3/4)

its-core ▸ src/main/java ▸ sg.edu.nus.se.its.interpreter ▸ Interpreter ▸

```java
package sg.edu.nus.se.its.interpreter;

import sg.edu.nus.se.its.model.Constant;

/**
 * Interfaces for the interpretation of the program execution.
 */
public interface Interpreter {

  public Trace executeProgram(Program program);

  public Trace executeProgram(Program program, Input input);

  public Object execute(Executable executable, Memory memory);

  public Object executeFunction(Function function, Memory memory);

  public Object executeConstant(Constant constant, Memory memory);

  public Object executeOperation(Operation operation, Memory memory);

  public Object executeVariable(Variable variable, Memory memory);

  public void setTimeout(int timeout);

}
```

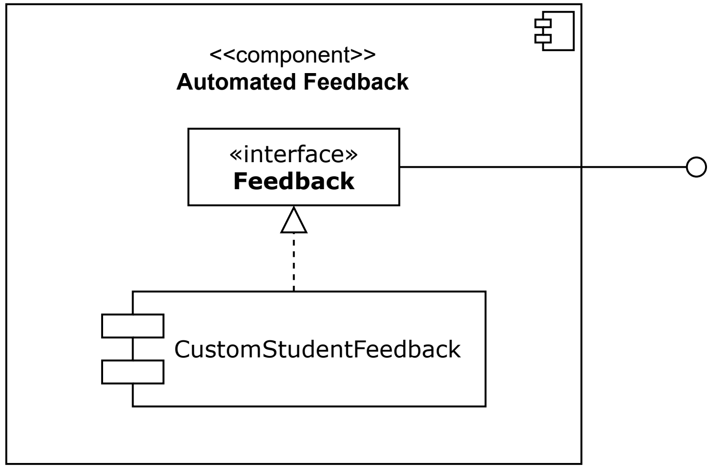
its-core ▸ src/main/java ▸ sg.edu.nus.se.its.repair ▸ Repair ▸

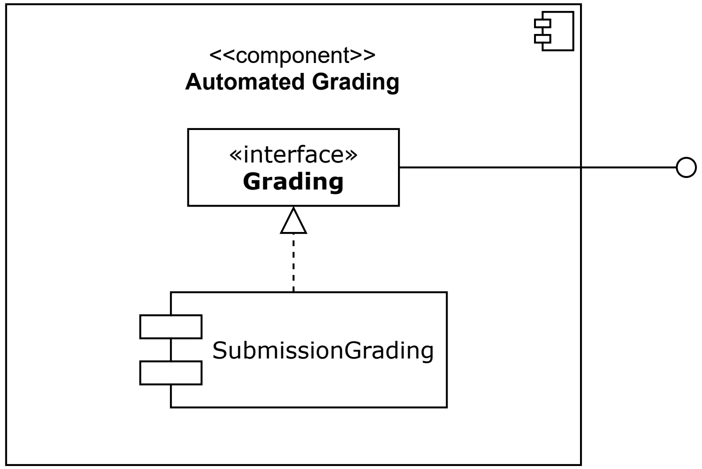
```java
package sg.edu.nus.se.its.repair;

import java.util.List;

/**
 * Interface for the repair module.
 */
public interface Repair {

  /**
   * Repairs the student's incorrect program, given the reference solution and a mapping of
   * erroneous blocks. The repaired program is in the internal representation format.
   *
   * @param referenceProgram -- the parsed internal representation of the reference program.
   * @param submittedProgram -- the parsed internal representation of the student's incorrect
   *        program.
   * @param errorLocations -- a list of bijective set of erroneous locations
   * @param variableMapping -- the variable mapping
   * @param inputs -- set of inputs used for the program
   * @return a list of possible repairs for the student's incorrect program
   * @see Program
   */
  public List<RepairCandidate> repair(Program referenceProgram, Program submittedProgram,
      ErrorLocalisation errorLocations, VariableMapping variableMapping, List<Input> inputs,
      Interpreter interpreter) throws Exception;
}
```

its-core ▸ src/main/java ▸ sg.edu.nus.se.its.errorlocalizer ▸ ErrorLocalizer ▸

```java
package sg.edu.nus.se.its.errorlocalizer;

import java.util.List;

/**
 * Interface for the error localization.
 */
public interface ErrorLocalizer {

  /**
   * Returns a list of erroneous blocks of the submitted program, given the reference program, the
   * student program and the mapping of aligned blocks.
   *
   * @param submittedProgram - the parsed internal representation of the student's program.
   * @param referenceProgram - the parsed internal representation of the reference program.
   * @param inputs - list of String value for the inputs to test for (can be null)
   * @param functionName - function to analyze
   * @param structuralMapping - the alignment of the reference program with the student's program.
   * @param variableMapping - the mapping of the variables in both programs
   * @param interpreter - interpreter object for the execution of the traces
   * @return list of error location pairs
   */
  public ErrorLocalisation localizeErrors(Program submittedProgram, Program referenceProgram,
      List<Input> inputs, String functionName, StructuralMapping structuralMapping,
      VariableMapping variableMapping, Interpreter interpreter);

}
```

Components (4/4)





its-core ▸ src/main/java ▸ sg.edu.nus.se.its.feedback ▸ Feedback ▸

```java
package sg.edu.nus.se.its.feedback;

import sg.edu.nus.se.its.model.Program;

/**
 * Interface for the feedback module.
 */
public interface Feedback {

  /**
   * Generates feedback for the student based on the identified list of repairs.
   *
   * @param repairCandidate -- one set of consistent local repairs
   * @param submittedProgram -- the submitted program
   * @return feedback in form of a String object
   */
  public String provideFeedback(RepairCandidate repairCandidate, Program submittedProgram);

}
```

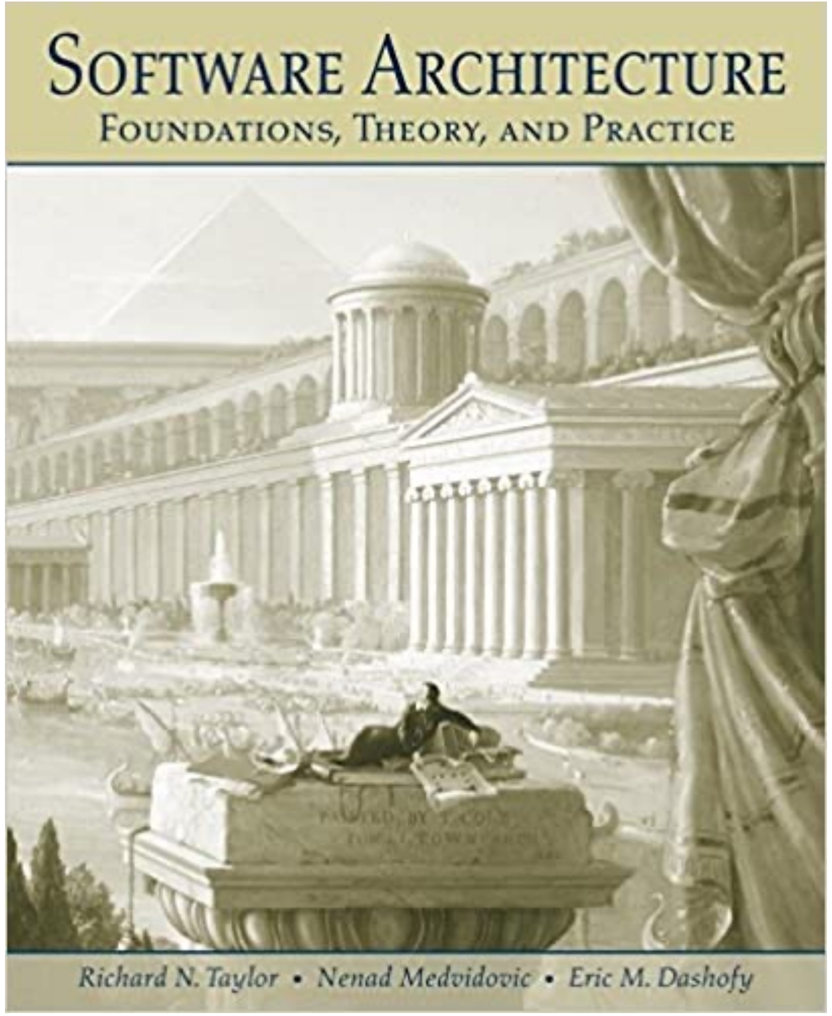
Note that for both, *Feedback* and *Grading*, the interfaces are not 100% fixed. Depending on ”what” information your *Feedback* and *Grading* strategy requires, we will adapt the interfaces accordingly. We will also discuss with you how feasible your plans are and how they can be adjusted to fit the system.

Acknowledgment

Our slides are based on the resources for:

"Software Architecture: Foundations, Theory, and Practice"
by Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; 2008 John Wiley & Sons, Inc.

https://www.softwarearchitecturebook.com/resources/



Layered Style (1/3)

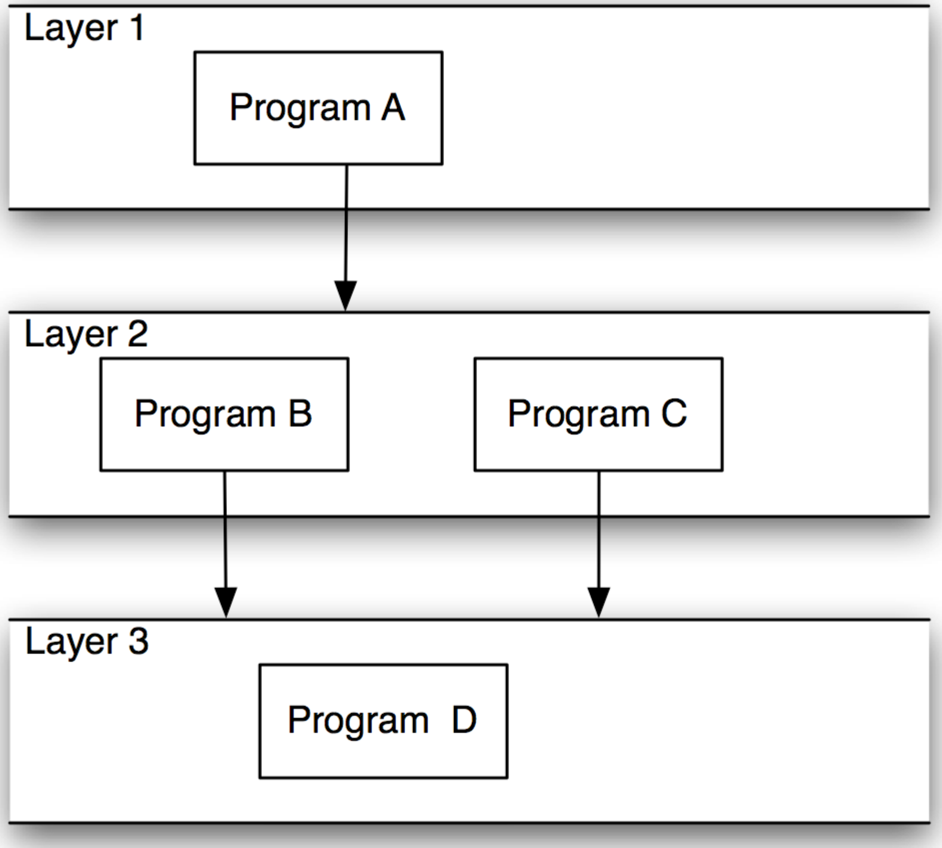
- Hierarchical system organization
  - "Multi-level client-server"
  - Each layer exposes an interface (API) to be used by above layers
- Each layer acts as a
  - Server: service provider to layers "above"
  - Client: service consumer of layer(s) "below"
- Connectors are protocols of layer interaction
- Example: operating systems
- Virtual machine style results from fully opaque layers

Layered Style (2/3)

- **Advantages**
  - Increasing abstraction levels
  - Evolvability
  - Changes in a layer affect at most the adjacent two layers
    - Reuse
  - Different implementations of layer are allowed as long as interface is preserved
  - Standardized layer interfaces for libraries and frameworks

Layered Style (3/3)

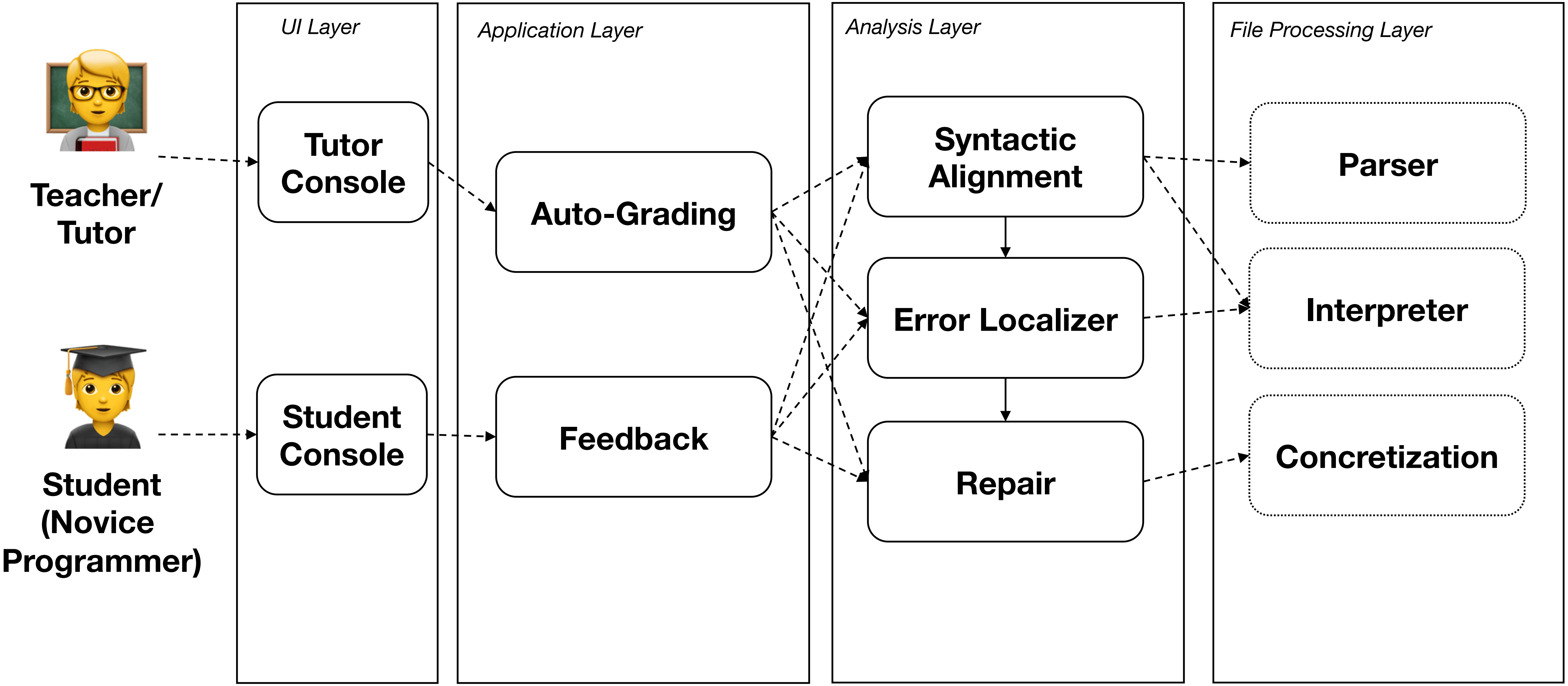
- **Disadvantages**
  - Not universally applicable
  - Performance
- **Layers** may have to be **skipped**
  - Determining the correct abstraction level



How can the ***Layered Style*** be applied for our ***Intelligent Tutoring System*** and its components?

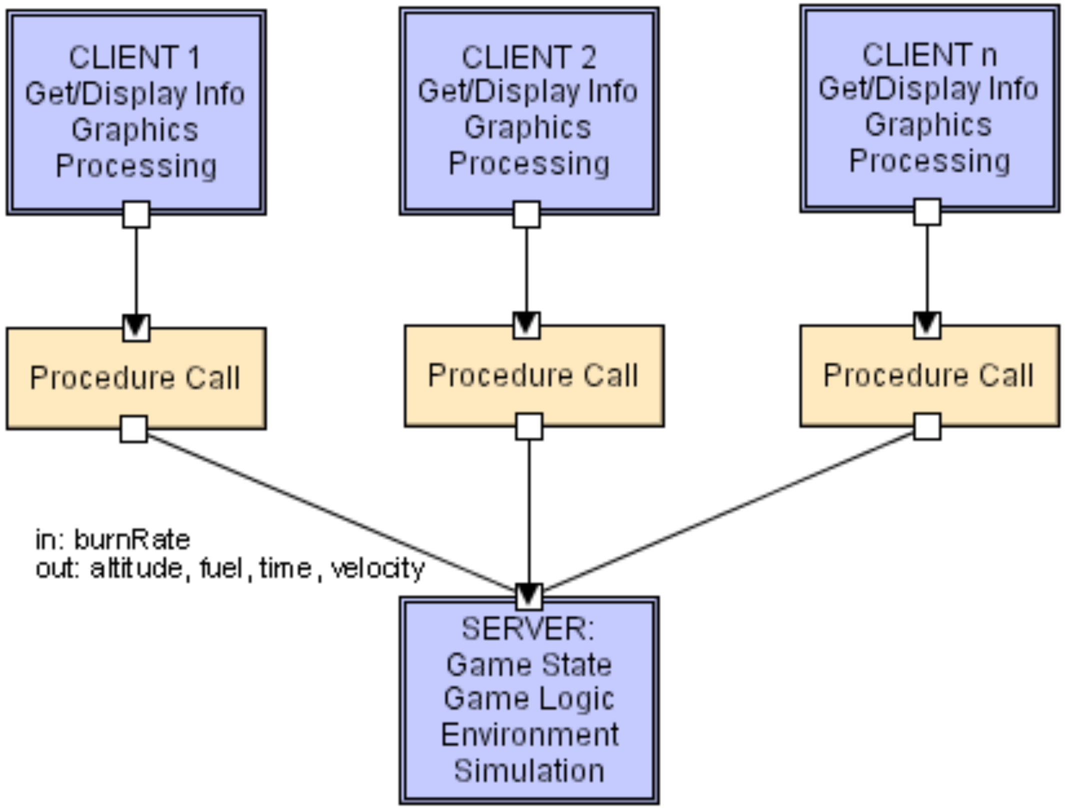
Note that the **following examples** for the Intelligent Tutoring System **might not** represent the best way of **organizing the components** and **designing their interactions**. They are meant for illustrating the dicussed architectural styles.

Layered Style (applied)



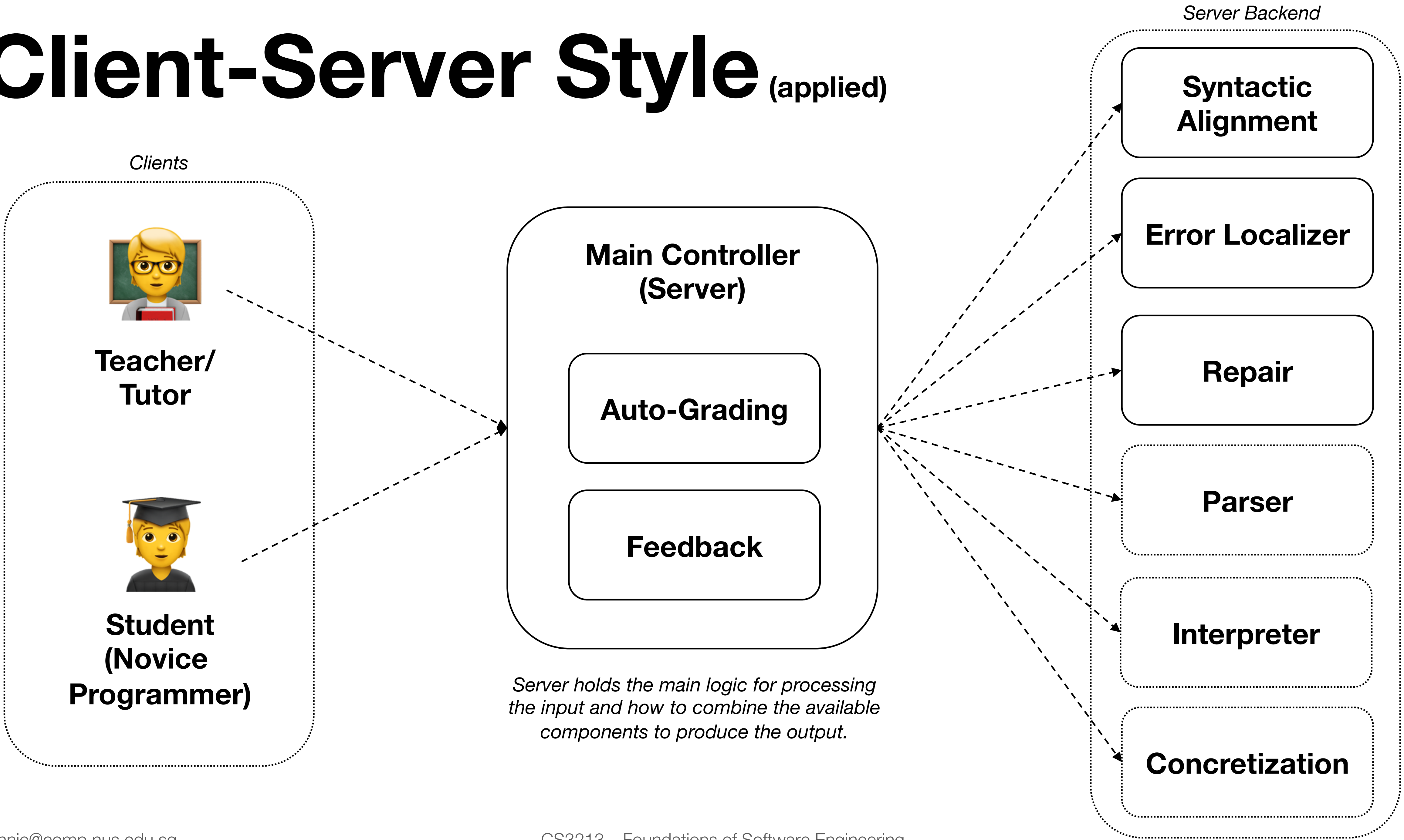
Client-Server Style

- **Components** are clients and servers
- **Servers** do not know number or identities of clients
- **Clients** know server's identity
- **Connectors** are RPC-based network interaction protocols



How can the ***Client-Server Style*** be applied for our ***Intelligent Tutoring System*** and its components?

Client-Server Style (applied)



Pipe and Filter Style (1/3)

- **Components are filters**
  - Transform input data streams into output data streams
  - Possibly incremental production of output
- **Connectors are pipes**
  - Conduits for data streams
- **Style invariants**
  - Filters are independent (no shared state)
  - Filter has no knowledge of up- or down-stream filters
- **Examples**
  - UNIX shell — signal processing
  - Distributed systems — parallel programming
- Example: `ls invoices | grep -e August | sort`

Pipe and Filter Style (2/3)

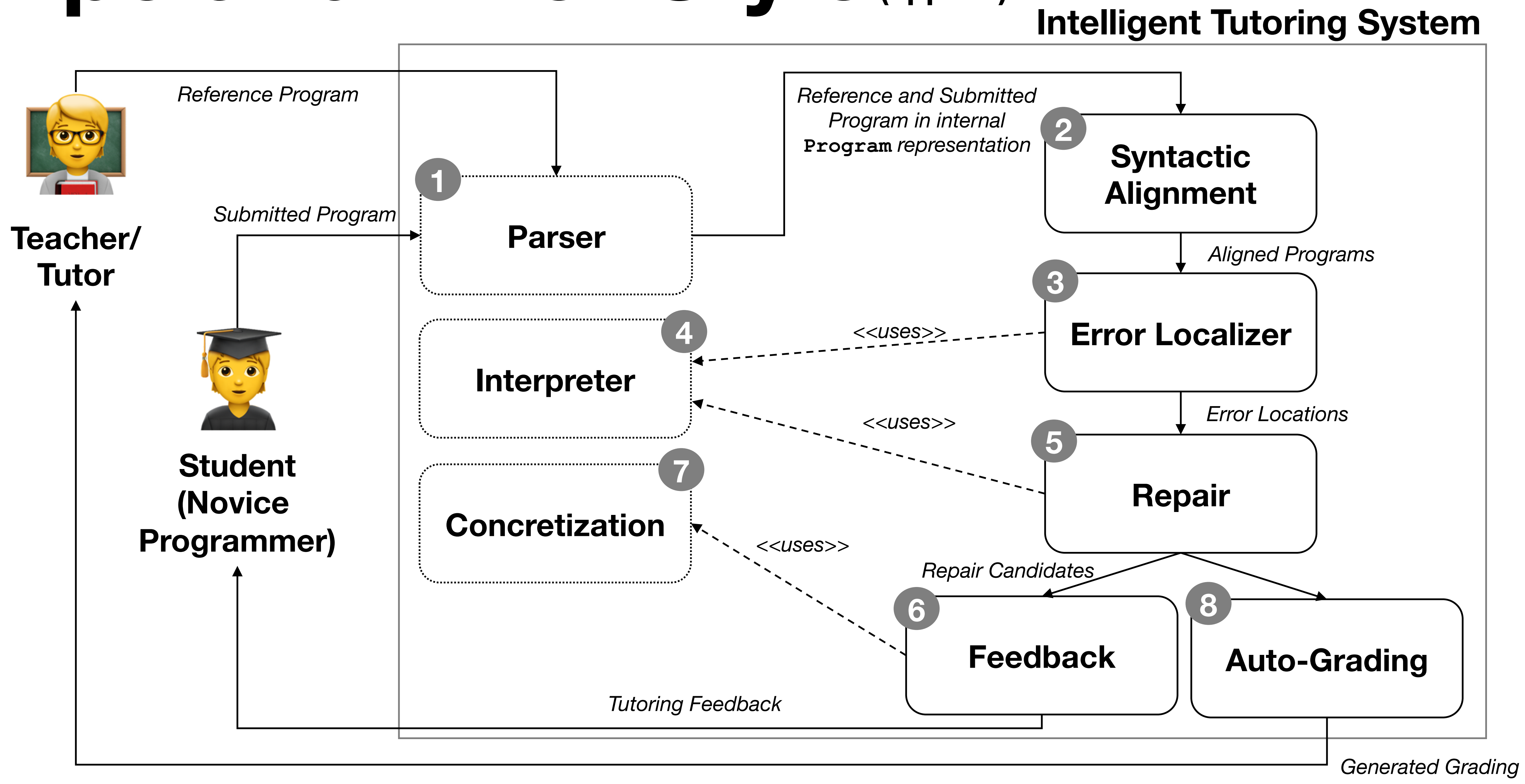
- **Variations**
  - Pipelines — linear sequences of filters
  - Bounded pipes — limited amount of data on a pipe
  - Typed pipes — data strongly typed
- **Advantages**
  - System behavior is a succession of component behaviors
  - Filter addition, replacement, and reuse
    - Possible to hook any two filters together
  - Certain analyses
    - Throughput, latency, deadlock
  - Concurrent execution

Pipe and Filter Style (3/3)

- **Disadvantages**
  - Batch organization of processing
  - Interactive applications
  - Lowest common denominator on data transmission

How can the ***Pipe and Filter Style*** be applied for our ***Intelligent Tutoring System*** and its components?

Pipe and Filter Style (applied)



Publish-Subscribe (1/3)

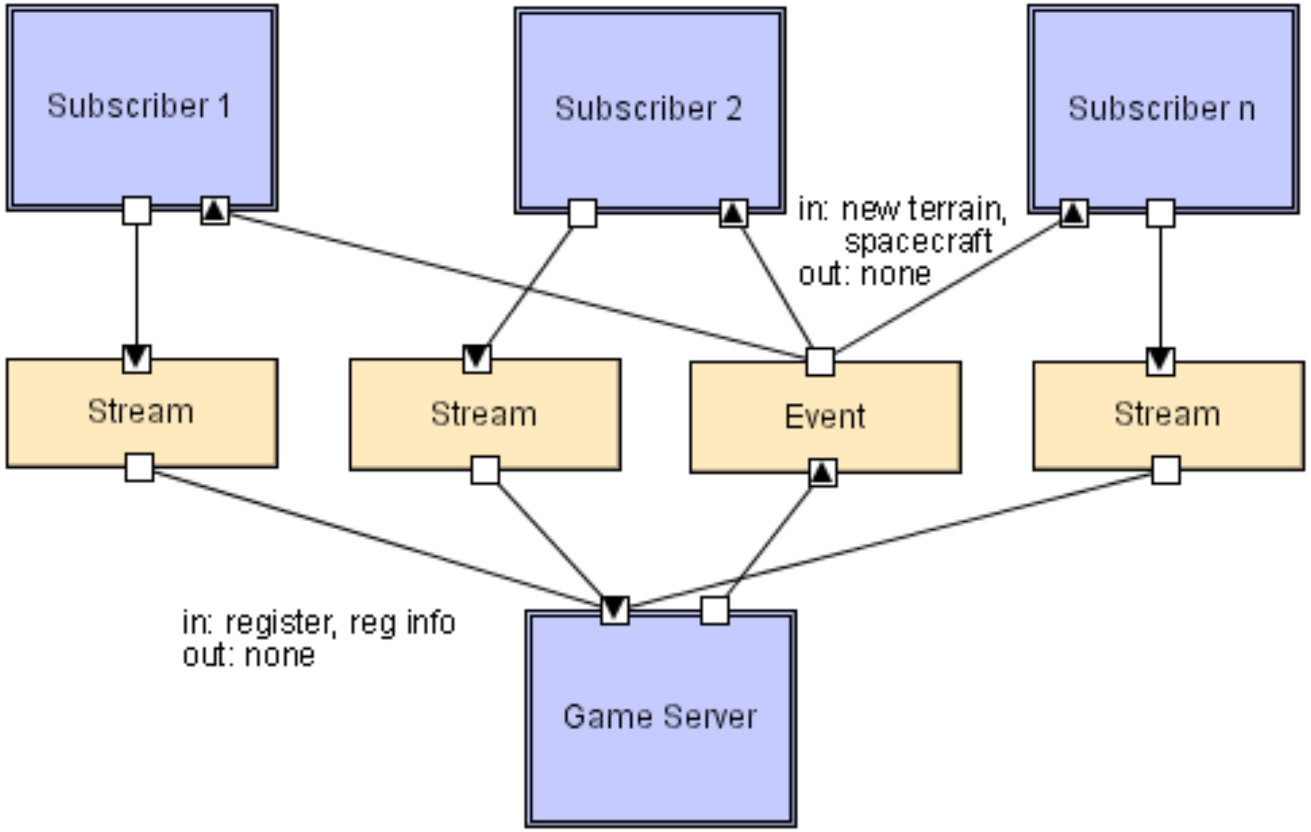
Subscribers register/deregister to receive specific messages or specific content.

Publishers broadcast messages to subscribers either synchronously or asynchronously.

Publish-Subscribe (2/3)

- **Components:** Publishers, subscribers, proxies for managing distribution
- **Connectors:** Typically a network protocol is required. Content-based subscription requires sophisticated connectors.
- **Data Elements:** Subscriptions, notifications, published information
- **Topology:** Subscribers connect to publishers either directly or may receive notifications via a network protocol from intermediaries
- **Qualities** yielded: Highly efficient one-way dissemination of information with very low-coupling of components

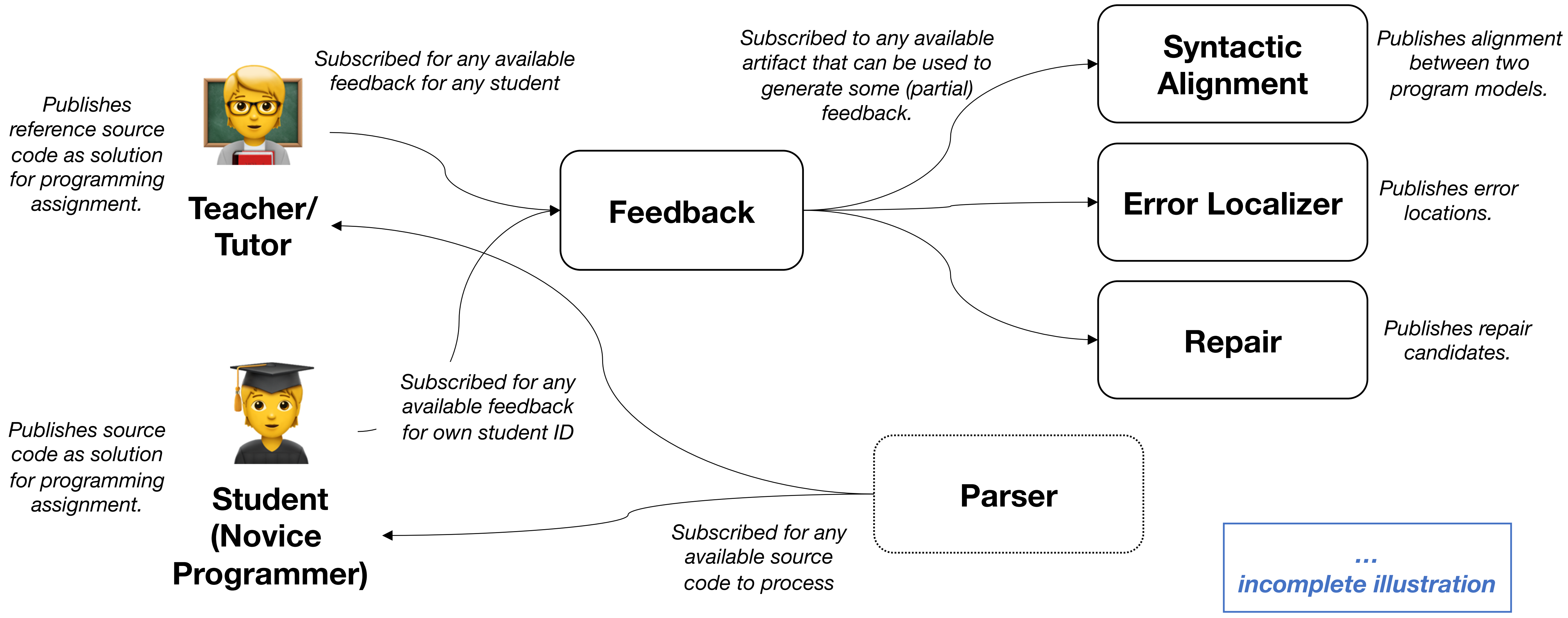
Publish-Subscribe (3/3)



How can the ***Publish-Subscribe Style*** be applied for our ***Intelligent Tutoring System*** and its components?

Publish-Subscribe (applied)

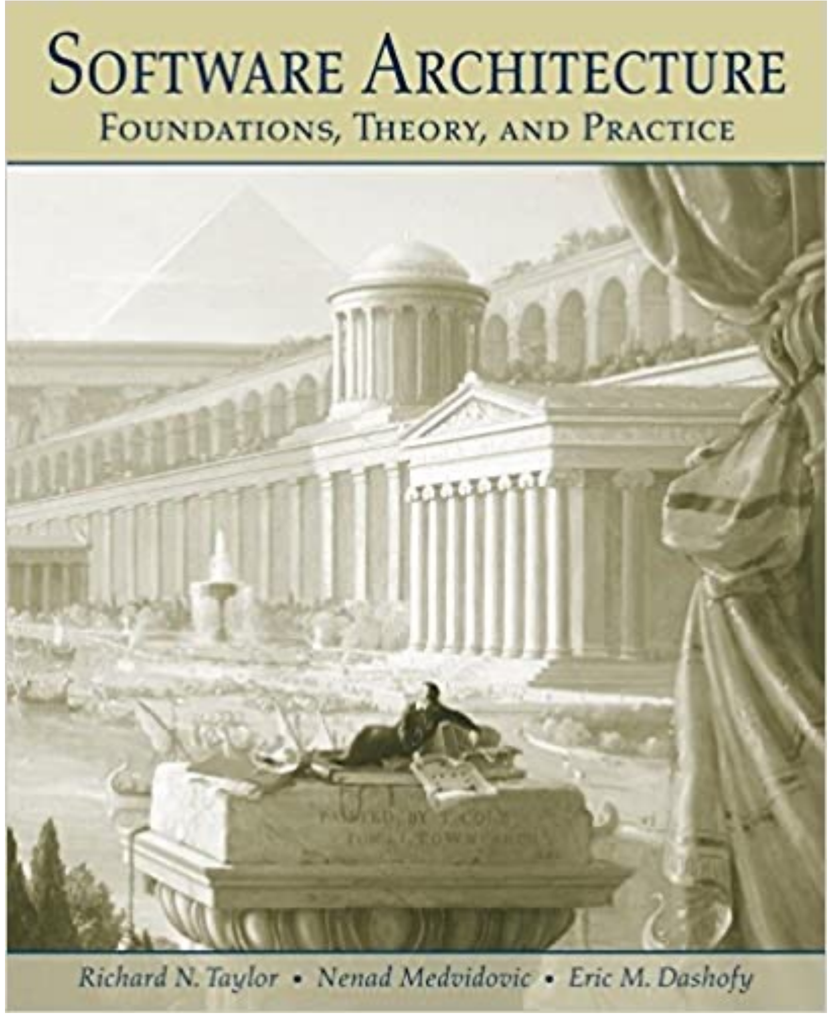
Arrows show subscriptions.



There is more!

- **Pipe-and-Filter**
- Shared-Data
- **Publish-Subscribe**
- **Client Server Style**
- Peer-to-Peer Style
- Communicating-Processes Style

"Software Architecture: Foundations, Theory, and Practice"
by Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; 2008 John Wiley & Sons, Inc.



Any remaining question about software architecture?