CS212

Software Engineering Case Study:
SaM and AMS
Spring 2007

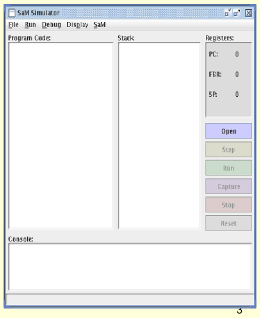
David Levitan

Basics of Good Design

- Keep the design as simple as possible yet complicated enough to easily expand
- Reuse as much as possible
  - Code modularization
  - Use OO to make everything as generic as possible
- Use the right programming language

Quick History of SaM

- Original version designed/developed by Prof. Keshav Pingali.
- Completely redesigned and reimplemented by Ivan Gyurdiev and myself several years ago.



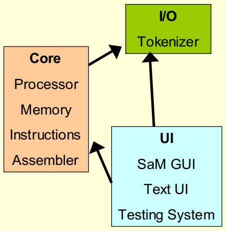
PC: 0
FBR: 0
SP: 0

Goals of SaM

- Provide a simplified JVM to students.
- Provide an easy to use GUI.
- Be easily expandable with new memory models, instructions, etc…
- Provide a simple tokenizer that can be used by both SaM assembler and Bali compilers.

Overall SaM Design

- We separated SaM into three distinct parts.
  - UI: Displays the system state.
  - Core: Manages system state and executes data.
  - I/O: Generic text tokenizer.



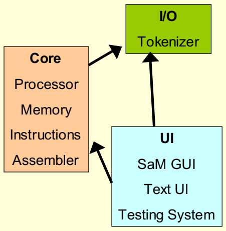
I/O
Tokenizer

Core
Processor
Memory
Instructions
Assembler

UI
SaM GUI
Text UI
Testing System

Overall SaM Design

- Each part has a specific purpose and interacts with other parts.
- Interfaces define all components. GUIs interact only with the interfaces
  - Components can be reimplemented without UI code modification.



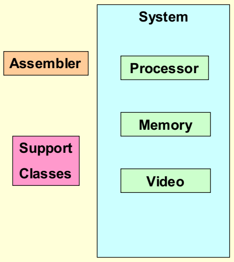
I/O
Tokenizer

Core
Processor
Memory
Instructions
Assembler

UI
SaM GUI
Text UI
Testing System

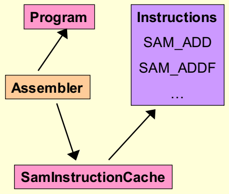
SaM Core Design

- Every component has both an interface and an implementation.
  - Allows easy switching of algorithms (malloc/free vs. garbage collection)
  - Enforces a specific interface to core access.



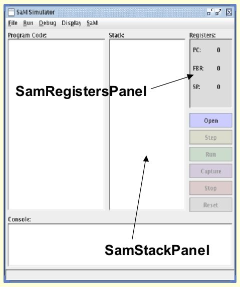
Instruction Loading

- Instructions are not hard coded.
- Reflection is used to load new instructions.
- Allows changes to instruction set without core code changes.
- Cache allows alternate sources.



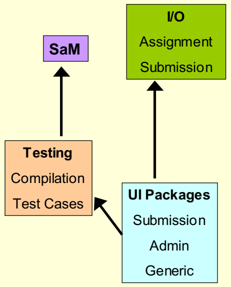
SaM GUI Design

- Everything is again modularized.
- However, no interfaces since outside access is not part of the design goals.
- Modularization does allow reuse of components in multiple GUIs.



AMS Design

- AMS is the testing system used to grade compilers.
- Code reuse
  - Testing package builds on SaM Testing.
  - Same GUI and testing harness is used by the tester in the student application and the TA application.

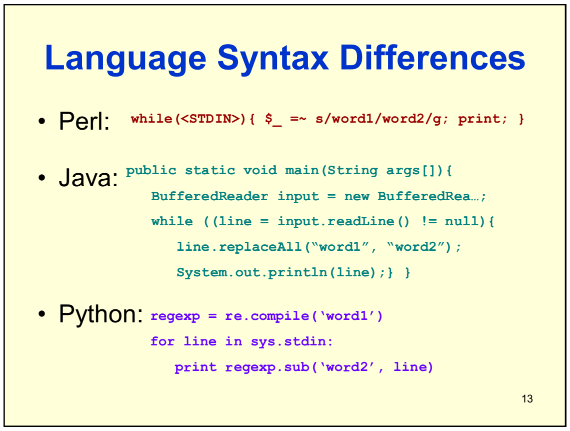


Design Summary

- Modularization is key.
  - Allows reuse of code.
    - Less code
    - Therefore, less code maintenance and fewer bugs
    - And allows you to build more faster.
- Providing standard interfaces is helpful for future code development and expansion.

Programming Languages

- Each language has strengths and weaknesses.
- You don’t always have a choice, but when you do, pick wisely.
- Two main categories of strengths/ weaknesses
  - Syntax
  - Capabilities



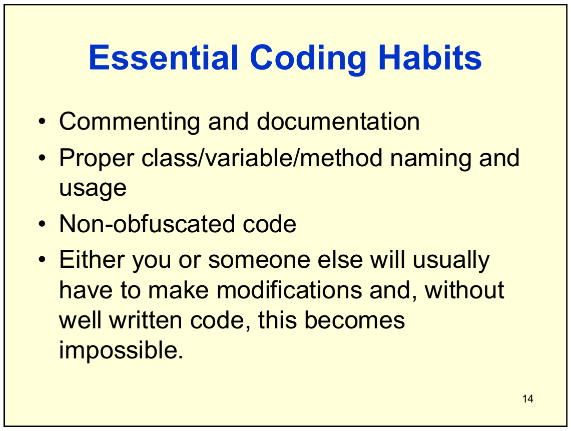
Language Syntax Differences

- Perl: `while(<STDIN>){ $_ =~ s/word1/word2/g; print; }`
- Java:

```
public static void main(String args[]){
   BufferedReader input = new BufferedRea…;
   while ((line = input.readLine() != null){
      line.replaceAll(“word1”, “word2”);
      System.out.println(line);} }
```

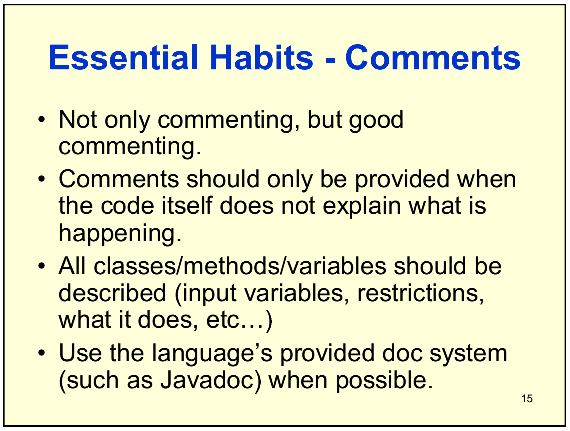
- Python:

```
regexp = re.compile(‘word1’)
for line in sys.stdin:
   print regexp.sub(‘word2’, line)
```



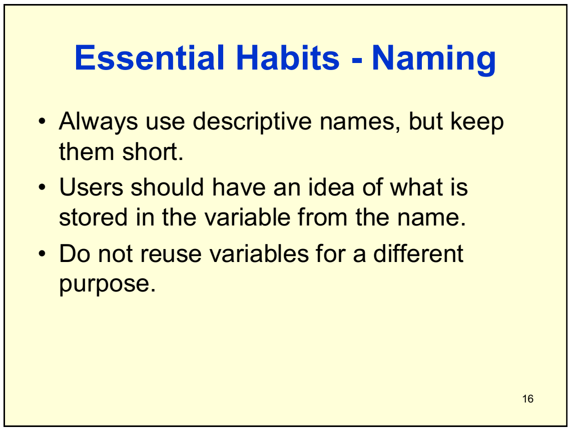
Essential Coding Habits

- Commenting and documentation
- Proper class/variable/method naming and usage
- Non-obfuscated code
- Either you or someone else will usually have to make modifications and, without well written code, this becomes impossible.



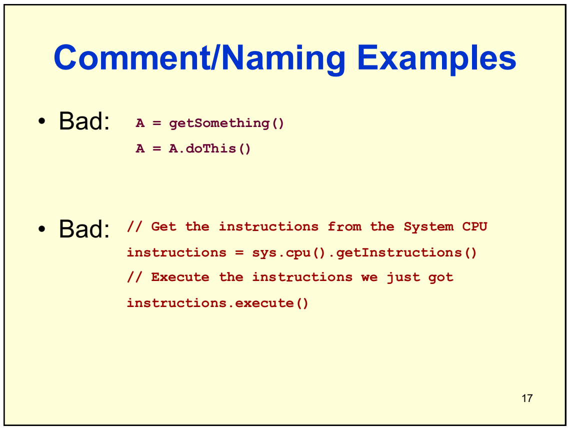
Essential Habits - Comments

- Not only commenting, but good commenting.
- Comments should only be provided when the code itself does not explain what is happening.
- All classes/methods/variables should be described (input variables, restrictions, what it does, etc…)
- Use the language’s provided doc system (such as Javadoc) when possible.



Essential Habits - Naming

- Always use descriptive names, but keep them short.
- Users should have an idea of what is stored in the variable from the name.
- Do not reuse variables for a different purpose.



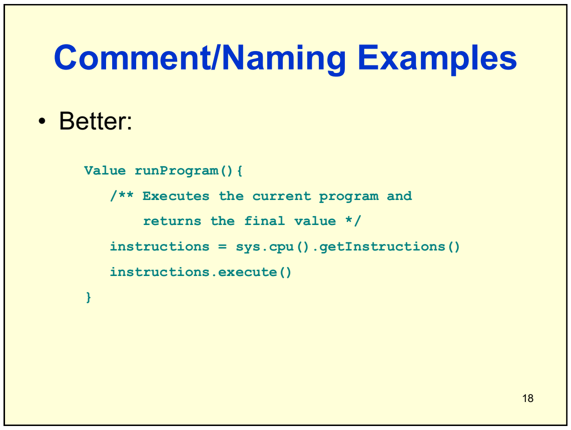
Comment/Naming Examples

- Bad:

```
A = getSomething()
A = A.doThis()
```

- Bad:

```
// Get the instructions from the System CPU
instructions = sys.cpu().getInstructions()
// Execute the instructions we just got
instructions.execute()
```



Comment/Naming Examples

- Better:

```
Value runProgram(){
   /** Executes the current program and
       returns the final value */
   instructions = sys.cpu().getInstructions()
   instructions.execute()
}
```