Kernel Implementations IV
Recording Design Decisions

• The *commutative diagram* is a great device to help you *think about* why (whether?) a kernel class correctly implements the kernel interface.

• However, it is also important to *record (document)* the key design decisions illustrated in a commutative diagram, if they are not already recorded in the Java code itself.
Two Key Design Decisions

• Perhaps surprisingly, there are really only two key design decisions that need to be recorded in (Javadoc) comments:
  – The *representation invariant*: Which “configurations” of values of the instance variables can ever arise?
  – The *abstraction function*: How are the values of the instance variables to be interpreted to get an abstract value?
Commutative Diagram
The abstract state space is fully described in the kernel interface (the mathematical model of the type).
Example: Abstract State Space

• Consider \texttt{NaturalNumberKernel}, where we find this in the API:

\textbf{Mathematical Subtypes:}
\texttt{NATURAL is integer}

\texttt{exemplar \ n}

\texttt{constraint \ n >= 0}

\textbf{Mathematical Model (abstract value and abstract invariant of this):}
\texttt{type NaturalNumberKernel is modeled by}

\texttt{NATURAL}
Example: Abstract State Space

- Consider **NaturalNumberKernel**, where we find this in the API:

  Mathematical Subtypes:
  
  \[
  \text{NATURAL is integer}
  \]
  
  exemplar \( n \)
  
  constraint \( n \geq 0 \)

  Mathematical Model (abstract value and abstract invariant of this):
  
  \[
  \text{type NaturalNumberKernel is modeled by NATURAL}
  \]
Example: Abstract State Space

• Consider `NaturalNumberKernel`, where we find this in the Mathematical Subtypes:

  Mathematical Subtypes:

  `NATURAL is integer`

  exemplar `n`

  `constraint n >= 0`

  Mathematical Model (abstract value and abstract invariant of this):

  `type NaturalNumberKernel is modeled by NATURAL`
Example: Abstract State Space

- Consider `NaturalNumberKernel`, where we find this in the API:

  **Mathematical Subtypes:**
  
  `NATURAL` *is integer*

  exemplar \( n \)

  constraint \( n \geq 0 \)

  **Mathematical Model** (abstract value and abstract invariant of this):

  type `NaturalNumberKernel` *is modeled by*

  `NATURAL`

  ... that is *constrained* to be non-negative (i.e., greater than or equal to 0).
For this example, then, the abstract state space comprises the non-negative integers.
Commutative

The abstract transition is fully described in the kernel interface (the method contract).
Example: Abstract Transition

• Consider `multiplyBy10`, where we find this in the API:

  Updates:
  `this`

  Requires:
  \(0 \leq k < 10\)

  Ensures:
  \(this = 10 \times \#this + k\)
The method’s requires clause says where a transition arrow starts, and the ensures clause says where it ends.
Commutative

The *concrete transition* is fully described in the kernel class (the method body).
Example: Concrete Transition

- Consider `NaturalNumber2`, where we find this code in the `multiplyBy10` method body:

```java
    if (this.digits.length() > 0 || k > 0) {
        this.digits.push(k);
    }
```
The code in the method’s body tells us where a concrete transition arrow starts and ends.
(Technically, you sometimes also need this to tell where an arrow starts; patience...)
The **concrete state space** is only partially described in the kernel class (the instance variables).
Example: Concrete State Space

• Consider `NaturalNumber2`, where we find one instance variable in the code:

```
private Stack<Integer> digits;
```
Example: Concrete State Space

• Consider `NaturalNumber2`, where we find one instance variable in the code:

```java
private Stack<Integer> digits;
```

The type of this variable, `Stack<Integer>`, tells us its mathematical model: _string of integer_.

So, in this example, we know everything in the concrete state space is a **string of integer** ...
... but we do not know whether all \textit{string of integer} values are in this space.
For instance, can these values of the instance variable \texttt{digits} ever arise?

\begin{itemize}
  \item <1>
  \item <-49, 17, 3>
  \item <0>
  \item <0, 5, 6>
  \item <6, 5, 0>
\end{itemize}
The interpretation of the instance variables as an abstract value is not described anywhere.
What’s Left to Write Down?
What’s Left to Write Down?

Item #1: Characterize the *concrete state space*.
The Representation Invariant

• The *representation invariant* characterizes the values that the data representation (instance variables) might have at the *end* of each kernel method body, including the constructor(s)

• The representation invariant is *made to hold* by the method bodies’ code, and it is *recorded* in the *convention* clause in a (Javadoc) comment for the kernel class
Variable Life-Cycle: Client
Variable Life-Cycle: Client

A variable is declared, e.g.,

NaturalNumber n ...
The variable is *initialized*, e.g.,

```java
... n = new NaturalNumber2();
```
Variable Life-Cycle: Client

A method is called, e.g.,

```java
n.multiplyBy10(7);
```
Variable Life-Cycle: Client

More methods are called, e.g.,

```java
n.multiplyBy10(4); ...
d = n.divideBy10(); ...
if (n.isZero()) {...}
```
Variable Life-Cycle: Client

The variable *goes out of scope*, i.e.,

\[ \ldots \]
Variable Life-Cycle: Client

The claim of the kernel class implementer is that the representation invariant holds at the end of the constructor call and each subsequent method call.
Now look *inside each call*. Note that the constructor body must *make* the representation invariant hold at the end of the constructor …
Variable Life-Cycle: Implementer

… so the representation invariant *must necessarily hold* at the *beginning* of the first method call …
Variable Life-Cycle: Implementer

... and the code in the body for that method must *make* the representation invariant hold at the *end* of the first method call ...
Variable Life-Cycle: Implementer

... and so on for each method call. The representation invariant therefore may be assumed to hold at the beginning of each method body, if the code makes it hold at the end of each method body!
Example: \texttt{NaturalNumber2}

- Can these values of the instance variable \texttt{digits} ever arise to represent the abstract \texttt{NaturalNumber} value seen by the client?

\texttt{<1>}
\texttt{<-49, 17, 3>}
\texttt{<0>}
\texttt{<0, 5, 6>}
\texttt{<6, 5, 0>
Example: \textit{NaturalNumber2}

- The implementer’s intent is that the value of \texttt{digits} has the following features:
  - It contains only the numbers $0, 1, \ldots, 9$
  - It never has a $0$ at the right end
Example: **NaturalNumber2**

- We might document this as follows (which is simpler than in the sample project code for **NaturalNumber2**):

  ```
  /**
   * @convention
   * for all k: integer
   *  where (<k> is substring of $this.digits)
   *  (0 <= k and k <= 9) and
   *  <0> is not suffix of $this.digits
   */
  ```
Example: \texttt{NaturalNumber2}

- We might document this as follows (which is simpler than in the sample code for \texttt{NaturalNumber2}):

```java
/**
 * @convention
 * for all \texttt{k: integer}
 * where (<k> is substring of $this.digits)
 * \texttt{0 \leq k and k \leq 9} and
 * <0> is not suffix of $this.digits
 */
```

This is the Javadoc tag for the representation invariant.
Example: **NaturalNumber2**

- We might document this as follows (which is simpler than in the sample code for **NaturalNumber2**):

```plaintext
/**
 * @convention
 * for all \( k: \text{integer} \)
 * where \(<k> \text{ is substring of } \$this.digits\)
 * \(0 \leq k \text{ and } k \leq 9\) \text{ and } \(<0> \text{ is not suffix of } \$this.digits\)
 */
```

\$this\ is special notation to name the data representation of \this\ in such comments.
Example: NaturalNumber2

- In fact, here is an even simpler way to say the same thing:

```plaintext
/**
 * @convention
 * entries($this.digits) is subset of {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} and
 * <0> is not suffix of $this.digits
 */
```
The Representation Invariant

• To summarize, for a kernel class:
  – The constructor(s) must \textit{make} the representation invariant true
  – The representation invariant may be \textit{assumed} to be true at the \textit{beginning} of each method body
  – Each method body must \textit{make} the representation invariant true (again) at the time it returns
What’s Left to Write Down?
What’s Left to Write Down?

Item #2: Document the *interpretation* of concrete values as abstract values.
The Abstraction Function

• The *abstraction function* describes how to *interpret* any concrete value (that satisfies the representation invariant) as an abstract value

• The abstraction function is not computed by any code, but is merely *recorded* in the *correspondence* clause in a (Javadoc) comment for the kernel class
Example: \texttt{NaturalNumber2}

- How does the kernel class implementer intend to interpret each of these values of the instance variable \texttt{digits}?

  \begin{itemize}
  
  \item \texttt{<1>}
  \item \texttt{< >}
  \item \texttt{<0, 5, 6>}
  \item \texttt{<1, 2, 3, 4, 5>}
  \item \texttt{<0>}
  \end{itemize}
Example: \texttt{NaturalNumber2}

- How does the kernel intend to interpret each of these values of the instance variable \texttt{digits}?

\begin{itemize}
  \item \texttt{<1>}
  \item \texttt{< >}
  \item \texttt{<0, 5, >}
  \item \texttt{<1, 3, 4, 5>}
  \item \texttt{<0>}
\end{itemize}

Trick question! This value of \texttt{digits} can never arise from the \texttt{NaturalNumber2} code. It does not satisfy the representation invariant; so, we don’t need to worry about how to interpret it!
Example: **NaturalNumber2**

- The interpretation of `digits` is described as follows (where `NUMERICAL_VALUE` is an appropriately-defined mathematical function from a `string of integer` to an `integer`):

```c
/**
 * @correspondence
 * this = NUMERICAL_VALUE(rev($this.digits))
 */
```
Example: **Natural**

- The interpretation of digits as follows (where `NUMERICAL_VALUE` is an appropriately-defined mathematical function from a string of integer to an `integer`):

```java
/**
 * @correspondence
 * this = NUMERICAL_VALUE(rev($this.digits))
 */
```

This is the Javadoc tag for the abstraction function.
Example: \texttt{NaturalNumber2}

- The interpretation of \texttt{digits} is described as follows (where \texttt{NUMERICAL\_VALUE} is an appropriately-defined mathematical function from a \textit{string of integer} to an \textit{integer}):

\begin{verbatim}
/**
 * @correspondence
 * \texttt{this} = \texttt{NUMERICAL\_VALUE} (\texttt{rev ($this$.digits)})
 */
\end{verbatim}

\texttt{this} is the (usual) notation to name the \textit{abstract value} in such comments.
Example: *NaturalNumber*

- The interpretation of digits is described as follows (where `NUMERICAL_VALUE` is an appropriately-defined mathematical function from a *string of integer* to an *integer*):

```cpp
/**
 * @correspondence
 * this = NUMERICAL_VALUE(rev($this.digits))
 */
```

$\textit{this}$ is special notation to name the data representation of $\textit{this}$ in such comments.
Consequences

• If the representation invariant and abstraction function are documented as suggested, then the work of implementing each constructor and each method in a kernel class can be done independently, and all the code will still “work together”
  – The code for each constructor and each method can be written by a different person!
Kernel Purity Rule

• *Kernel Purity Rule* — No method body in the kernel class should call any public method from the same component family
  – Every public method in the component family relies (for its correctness) on the representation invariant being satisfied when it is called, and this might not be true when a call is made from inside a method of the kernel class
Implications

• Implications of the kernel purity rule:
  – No public kernel method should call any other public kernel method from the same class
  – No public kernel method should call itself recursively
  – No method (public or private) in the kernel class should call any layered/secondary method from the same component family