Thinking About Correctness

Nick Sumner
wsumner@sfu.ca
We prefer correct software

- Software bugs make life painful
We prefer correct software

- Software bugs make life painful
  - By now you have first hand experience
  - Tracking down causes can be challenging (RCA/Root Cause Analysis)
  - Even just agreeing on what a bug is can be challenging
We prefer correct software.

- Software bugs make life painful.
  - By now you have first hand experience.
  - Tracking down causes can be challenging (RCA/Root Cause Analysis).
  - Even just agreeing on what a bug is can be challenging.

- Think back to a familiar example. Where is the bug?

```java
Position
getNewPosition(Position old, double speedInMPH) {
    ...
    return newPosition;
}
...
... = getNewPosition(old, speedInMPS);
```
We prefer correct software.

- Software bugs make life painful.
  - By now you have first hand experience.
  - Tracking down causes can be challenging (RCA/Root Cause Analysis).
  - Even just agreeing on what a bug is can be challenging.

- Think back to a familiar example. Where is the bug?
  - Is it in `getNewPosition`?
  - Is it in the calling code?
  - Is it in the design requirements?!
We prefer correct software

- Software bugs make life painful
  - By now you have first hand experience
  - Tracking down causes can be challenging (RCA/Root Cause Analysis)
  - Even just agreeing on what a bug is can be challenging

- Think back to a familiar example. Where is the bug?
  - Is it in getNewPosition?
  - Is it in the calling code?
  - Is it in the design requirements?!

- In reality, even agreeing on where a bug resides can be fraught
  - Many bugs do not even have a root cause in code!
We prefer correct software

- Software bugs make life painful
  - By now you have first hand experience
  - Tracking down causes can be challenging (RCA/Root Cause Analysis)
  - Even just agreeing on what a bug is can be challenging

- Think back to a familiar example. Where is the bug?
  - Is it in getNewPosition?
  - It it in the calling code?
  - Is it in the design requirements?!

- In reality, even agreeing on where a bug resides can be fraught
  - Many bugs do not even have a root cause in code!

- We need extra leverage to make the problem manageable
Thinking about correctness can be guided by specifications
Specifications

- Thinking about correctness can be guided by specifications
- **Specifications** explain what a component is *intended* to do
  - What are the requirements necessary for successful completion?
  - What are the guarantees provided during execution & upon completion?
Specifications

- Thinking about correctness can be guided by *specifications*
- *Specifications* explain what a component is *intended* to do
  - What are the requirements necessary for successful completion?
  - What are the guarantees provided during execution & upon completion?
- For *clients*, these
  - separate the intentions/interface from implementation details
Thinking about correctness can be guided by specifications. Specifications explain what a component is intended to do, including:
- What are the requirements necessary for successful completion?
- What are the guarantees provided during execution & upon completion?

For clients, these:
- separate the intentions/interface from implementation details
- clarify the correct use
Specifications

- Thinking about correctness can be guided by *specifications*
- *Specifications* explain what a component is *intended* to do
  - What are the requirements necessary for successful completion?
  - What are the guarantees provided during execution & upon completion?
- For *clients*, these
  - separate the intentions/interface from implementation details
  - clarify the correct use
  - (maybe) provide safety guarantees to ensure correct use
Specifications

- Thinking about correctness can be guided by *specifications*
- **Specifications** explain what a component is *intended* to do
  - What are the requirements necessary for successful completion?
  - What are the guarantees provided during execution & upon completion?
- For *clients*, these
  - separate the intentions/interface from implementation details
  - clarify the correct use
  - *(maybe)* provide safety guarantees to ensure correct use
- For *implementers*, these
  - guide the design requirements & details *(smaller design space)*
Thinking about correctness can be guided by *specifications*.

*Specifications* explain what a component is *intended* to do:
- What are the requirements necessary for successful completion?
- What are the guarantees provided during execution & upon completion?

For *clients*, these:
- separate the intentions/interface from implementation details
- clarify the correct use
- (maybe) provide safety guarantees to ensure correct use

For *implementers*, these:
- guide the design requirements & details (smaller design space)
- Enable changing the implementation as long as the spec is met!
Specifications

- Thinking about correctness can be guided by *specifications*
- *Specifications* explain what a component is *intended* to do
  - What are the requirements necessary for successful completion?
  - What are the guarantees provided during execution & upon completion?
- For *clients*, these
  - separate the intentions/interface from implementation details
  - clarify the correct use
  - (maybe) provide safety guarantees to ensure correct use
- For *implementers*, these
  - guide the design requirements & details (smaller design space)
  - Enable changing the implementation as long as the spec is met!
- The specification is a *contract* for usage
Specifications

- Thinking about correctness can be guided by *specifications*

- **Specifications** explain what a component is *intended* to do
  - What are the requirements necessary for successful completion?
  - What are the guarantees provided during execution & upon completion?

- For *clients*, these
  - separate the intentions/interface from implementation details
  - clarify the correct use
  - (maybe) provide safety guarantees to ensure correct use

- For *implementers*, these
  - guide the design requirements & details (smaller design space)
  - Enable changing the implementation as long as the spec is met!

- The specification is a *contract* for usage

Specifications also help establish root causes and guide fixing / maintenance.
Specifications (hopefully review)

- A specification usually includes
Specifications (hopefully review)

- A specification usually includes
  - *Preconditions*: guarantees a client must make upon usage
Specifications (hopefully review)

- A specification usually includes
  - *Preconditions:* guarantees a client must make upon usage
  - *Postconditions:* guarantees a provider must make if the client help up their end
Specifications (hopefully review)

- A specification usually includes
  - *Preconditions*: guarantees a client must make upon usage
  - *Postconditions*: guarantees a provider must make if the client help up their end
  - Additional nonfunctional requirements can be specified, as well
Specifications (hopefully review)

- A specification usually includes
  - *Preconditions:* guarantees a client must make upon usage
  - *Postconditions:* guarantees a provider must make if the client help up their end
  - Additional nonfunctional requirements can be specified, as well

- Note, if the preconditions *do not* hold, no guarantees are made
Specifications (hopefully review)

- A specification usually includes
  - *Preconditions:* guarantees a client must make upon usage
  - *Postconditions:* guarantees a provider must make if the client help up their end
  - Additional nonfunctional requirements can be specified, as well
- Note, if the preconditions *do not* hold, no guarantees are made
- For example:

```cpp
template<class Range, class Value>
size_t find(const Range& r, const Value& v);
```

PRECONDITION: r contains the value v
POSTCONDITION: returns an index of v in r
Specifications (hopefully review)

- A specification usually includes
  - **Preconditions:** guarantees a client must make upon usage
  - **Postconditions:** guarantees a provider must make if the client help up their end
  - Additional nonfunctional requirements can be specified, as well

- Note, if the preconditions *do not* hold, no guarantees are made

- For example:

```cpp
template<class Range, class Value>
size_t find(const Range& r, const Value& v);

PRECONDITION: r contains the value v
POSTCONDITION: returns an index of v in r
```
Specifications (hopefully review)

- A specification usually includes
  - *Preconditions*: guarantees a client must make upon usage
  - *Postconditions*: guarantees a provider must make if the client help up their end
  - Additional nonfunctional requirements can be specified, as well

- Note, if the preconditions *do not* hold, no guarantees are made

- For example:

```cpp
template<class Range, class Value>
size_t find(const Range& r, const Value& v);
```

**PRECONDITION:** r contains the value v  
**POSTCONDITION:** returns an index of v in r
Specifications (hopefully review)

- A specification usually includes
  - **Preconditions:**
    guarantees a client must make upon usage
  - **Postconditions:**
    guarantees a provider must make if the client help up their end
  - Additional nonfunctional requirements can be specified, as well

- Note, if the preconditions do not hold, no guarantees are made

- For example:

```cpp
template<class Range, class Value>
size_t find(const Range& r, const Value& v);
```

PRECONDITION: r contains the value v
POSTCONDITION: returns an index of v in r

How does this spec decouple the interface from implementation?
Specifications (hopefully review)

- A specification usually includes
  - **Preconditions**: guarantees a client must make upon usage
  - **Postconditions**: guarantees a provider must make if the client help up their end
  - Additional nonfunctional requirements can be specified, as well

- Note, if the preconditions *do not* hold, no guarantees are made

- For example:

```cpp
template<class Range, class Value>
size_t find(const Range& r, const Value& v);
```

**PRECONDITION**: r contains the value v
**POSTCONDITION**: returns the lowest index of v in r

Specifications (hopefully review)

- A specification usually includes
  - *Preconditions:* guarantees a client must make upon usage
  - *Postconditions:* guarantees a provider must make if the client help up their end
  - Additional nonfunctional requirements can be specified, as well

- Note, if the preconditions do not hold, no guarantees are made

- For example:

```cpp
template<class Collection, class Predicate>
Range partition(const Range& r, const Predicate& p);
```

PRECONDITION: None
POSTCONDITION:
Reorders r s.t. \( \forall x, y \in r, p(x) \land \neg p(y) \Rightarrow \text{index}(x) < \text{index}(y). \)
Returns the range s at the front of r s.t. \( \forall x \in r, p(x) \iff x \in s. \)
Specifications (hopefully review)

- A specification usually includes
  - *Preconditions*: guarantees a client must make upon usage
  - *Postconditions*: guarantees a provider must make if the client help up their end
  - Additional nonfunctional requirements can be specified, as well

- Note, if the preconditions do not hold, no guarantees are made

For example:

```cpp
template<class Collection, class Predicate>
Range partition(const Range& r, const Predicate& p);
```

**PRECONDITION:** None

**POSTCONDITION:**
- Reorders r s.t. \(\forall x,y \in r, p(x) \& \neg p(y) \rightarrow index(x) < index(y)\).
- Returns the range s at the front of r s.t. \(\forall x \in r, p(x) \leftrightarrow x \in s\).
Specifications (hopefully review)

- A specification usually includes
  - *Preconditions*: guarantees a client must make upon usage
  - *Postconditions*: guarantees a provider must make if the client help up their end
  - Additional nonfunctional requirements can be specified, as well

- Note, if the preconditions *do not* hold, no guarantees are made

- For example:

  ```cpp
template<class Collection, class Predicate>
Range partition(const Range& r, const Predicate& p);
```

  **PRECONDITION:** None

  **POSTCONDITION:**
  Reorders r s.t. $\forall x, y \in r, \ p(x) \& !p(y) \rightarrow \text{index}(x) < \text{index}(y)$. Returns the range s at the front of r s.t. $\forall x \in r, \ p(x) \iff x \in s$. 
Specifications (hopefully review)

- A specification usually includes
  - *Preconditions:* guarantees a client must make upon usage
  - *Postconditions:* guarantees a provider must make if the client help up their end
  - Additional nonfunctional requirements can be specified, as well

- Note, if the preconditions *do not* hold, no guarantees are made

- For example:

  ```cpp
  template<class Collection, class Predicate>
  Range partition(const Range& r, const Predicate& p);
  ```

  **PRECONDITION:** None
  **POSTCONDITION:**
  Reorders r s.t. \( \forall x,y \in r, p(x) \& \neg p(y) \rightarrow \text{index}(x) < \text{index}(y). \)
  Returns the range s at the front of r s.t. \( \forall x \in r, p(x) \leftrightarrow x \in s. \)
Specifications

- Specifications can be formal or informal
Specifications

- Specifications can be formal or informal
  - Informal: usually expressed in comments
  - Formal: expressed in a language that can automatically be analyzed
Specifications

• Specifications can be formal or informal
  – **Informal**: usually expressed in comments
  – **Formal**: expressed in a language that can automatically be analyzed

• What sorts of trade-offs do you see between these?
Specifications

- Specifications can be formal or informal
  - Informal: usually expressed in comments
  - Formal: expressed in a language that can automatically be analyzed

What sorts of trade-offs do you see between these?

---

Omar Rizwan
@rs nous

but why 5??

```c
prebuffer, cJSON_Bool imt);
/* Render a cJSON entity to text using a buffer already allocated in memory with given length. Returns 1 on success and 0 on failure. */
/* NOTE: cJSON is not always 100% accurate in estimating how much memory it will use, so to be safe allocate 5 bytes more than you actually need */
CJSON_PUBLIC(cJSON_Bool) cJSON_PrintPreallocated(cJSON *item, char *buffer, const int length, const cJSON_Bool format);
/* Delete a cJSON entity and all subentities. */
```
Specifications

• Specifications can be formal or informal
  – Informal: usually expressed in comments
  – Formal: expressed in a language that can automatically be analyzed

• What sorts of trade-offs do you see between these?
  – Informal specs allow loose reasoning & may even hide bugs. They can also drift from the implementation. BUT they are cheaper to write.
Specifications

• Specifications can be formal or informal
  – Informal: usually expressed in comments
  – Formal: expressed in a language that can automatically be analyzed

• What sorts of trade-offs do you see between these?
  – Informal specs allow loose reasoning & may even hide bugs. They can also drift from the implementation. BUT they are cheaper to write.
  – Formal specs can be challenging to write (imagine distributed systems). If code is poorly coupled, they increase maintenance costs. BUT they provide stronger guarantees.
Specifications

- Specifications can be formal or informal
  - Informal: usually expressed in comments
  - Formal: expressed in a language that can automatically be analyzed

- What sorts of trade-offs do you see between these?
  - Informal specs allow loose reasoning & may even hide bugs. They can also drift from the implementation. BUT they are cheaper to write.
  - Formal specs can be challenging to write (imagine distributed systems). If code is poorly coupled, they increase maintenance costs. BUT they provide stronger guarantees.

- In practice, a combination of the two is frequently used. Being able to reason formally helps with designing systems. Managing risk/benefit is important.
Specifications

- Each language will have its own tools and languages for writing formal specs, e.g.
  - Java – JML
  - C++ - Boost contracts, std contracts (maybe)
  - Eiffel – built in
Specifications

- Each language will have its own tools and languages for writing formal specs, e.g.

```java
//@ requires sortedArray != null
&& 0 < sortedArray.length < Integer.MAX_VALUE;
//@ requires \forall int i; 0 <= i < sortedArray.length;
    \forall int j; i < j < sortedArray.length;
    sortedArray[i] <= sortedArray[j];
//@ old boolean containsValue =
    (\exists int i; 0 <= i < sortedArray.length; sortedArray[i] == value);
//@ ensures containsValue ==> 0 <= \result < sortedArray.length;
//@ ensures !containsValue ==> \result == -1;
//@ pure
public static int search(int[] sortedArray, int value) {
    ...
}
```
Each language will have its own tools and languages for writing formal specs, e.g.

```java
//@ requires sortedArray != null
&& 0 < sortedArray.length < Integer.MAX_VALUE;
//@ requires \forall int i; 0 <= i < sortedArray.length;
\forall int j; i < j < sortedArray.length;
    sortedArray[i] <= sortedArray[j];
//@ old boolean containsValue =
    (\exists int i; 0 <= i < sortedArray.length; sortedArray[i] == value);
//@ ensures containsValue <=> 0 <= \result < sortedArray.length;
//@ ensures !containsValue <=> \result == -1;
//@ pure
public static int search(int[] sortedArray, int value) {
    ...
}
```
Specifications

- Each language will have its own tools and languages for writing formal specs, e.g.

```java
//@ requires sortedArray != null
&& 0 < sortedArray.length < Integer.MAX_VALUE;
//@ requires \forall int i; 0 <= i < sortedArray.length;
\forall int j; i < j < sortedArray.length;
sortedArray[i] <= sortedArray[j];
//@ old boolean containsValue =
\exists int i; 0 <= i < sortedArray.length; sortedArray[i] == value;
//@ ensures containsValue <=> 0 <= \result < sortedArray.length;
//@ ensures !containsValue <=> \result == -1;
//@ pure
public static int search(int[] sortedArray, int value) {
    ...
}
```
Specifications

- Each language will have its own tools and languages for writing formal specs, e.g.

```java
//@ requires sortedArray != null
&& 0 < sortedArray.length < Integer.MAX_VALUE;
//@ requires \forall int i; 0 <= i < sortedArray.length;
\forall int j; i < j < sortedArray.length;
\forall int j; i < j < sortedArray.length;
\forall int j; i < j < sortedArray.length;
//@
//@ old boolean containsValue =
\exists int i; 0 <= i < sortedArray.length; sortedArray[i] == value);
//@ ensures containsValue <-> 0 <= \result < sortedArray.length;
//@ ensures !containsValue <-> \result == -1;
//@ pure
public static int search(int[] sortedArray, int value) {
    ...
}
```
Specifications

- Each language will have its own tools and languages for writing formal specs, e.g.

```java
//@ requires sortedArray != null
&& 0 < sortedArray.length < Integer.MAX_VALUE;
//@ requires \forall int i; 0 <= i < sortedArray.length;
   \forall int j; i < j < sortedArray.length;
   sortedArray[i] <= sortedArray[j];
//@ old boolean containsValue =
   (\exists int i; 0 <= i < sortedArray.length; sortedArray[i] == value);
//@ ensures containsValue <=> 0 <= \result < sortedArray.length;
//@ ensures !containsValue <=> \result == -1;
//@ pure
public static int search(int[] sortedArray, int value) {
    ...
}
```

[OpenJML]
Specifications

- Each language will have its own tools and languages for writing formal specs, e.g.

```java
//@ requires sortedArray != null
&& 0 < sortedArray.length < Integer.MAX_VALUE;
//@ requires \forall int i; 0 <= i < sortedArray.length;
    \forall int j; i < j < sortedArray.length;
        sortedArray[i] <= sortedArray[j];
//@ old boolean containsValue =
    (\exists int i; 0 <= i < sortedArray.length; sortedArray[i] == value);
//@ ensures containsValue <=> 0 <= \result < sortedArray.length;
//@ ensures !containsValue <=> \result == -1;
//@ pure
public static int search(int[] sortedArray, int value) {
    ...
}
```
Specifications

- Each language will have its own tools and languages for writing formal specs, e.g.

```java
//@ requires sortedArray != null
&& 0 < sortedArray.length < Integer.MAX_VALUE;
//@ requires \forall int i; 0 <= i < sortedArray.length;
    \forall int j; i < j < sortedArray.length;
        sortedArray[i] <= sortedArray[j];
//@ old boolean containsValue =
    (\exists int i; 0 <= i < sortedArray.length; sortedArray[i] == value);
//@ ensures containsValue ==> 0 <= \result < sortedArray.length;
//@ ensures !containsValue ==> \result == -1;
//@ pure
public static int search(int[] sortedArray, int value) {
    ...}
```
Specifications

- Each language will have its own tools and languages for writing formal specs, e.g.
  - Java – JML
  - C++ - Boost contracts, std contracts (maybe)
  - Eiffel – built in

```java
public static int search(int[] sortedArray, int value) {
    assert sortedArray != null && 0 < sortedArray.length;
    assert isSorted(sortedArray) : "Array not sorted";
    ... 
    assert -1 <= result && result < array.length;
}
```

Trade offs?
Specifications

• Each language will have its own tools and languages for writing formal specs, e.g.
  – Java – JML
  – C++ - Boost contracts, std contracts (maybe)
  – Eiffel – built in

• Using these formal specs enables contracts to be checked at compile time in high assurance code!
Specifications

- Each language will have its own tools and languages for writing formal specs, e.g.
  - Java – JML
  - C++ - Boost contracts, std contracts (maybe)
  - Eiffel – built in
- Using these formal specs enables contracts to be checked at compile time in high assurance code!
- These are generally built on foundations of program logics
Specifications

• Each language will have its own tools and languages for writing formal specs, e.g.
  – Java – JML
  – C++ - Boost contracts, std contracts (maybe)
  – Eiffel – built in

• Using these formal specs enables contracts to be checked at compile time in high assurance code!

• These are generally built on foundations of program logics

\[
\{P\} \mathcal{C} \{Q\}
\]

  – When P holds before a component c, Q will hold after
Specifications – design concerns

Design concerns

- How clear & informative is the specification to a reader?
Specifications – design concerns

Design concerns

- How clear & informative is the specification to a reader?
- Is the specification strong enough to prevent defects?
Specifications – design concerns

Design concerns

- How clear & informative is the specification to a reader?
- Is the specification strong enough to prevent defects?
- Is the specification weak enough to allow flexibility?
Specifications – design concerns

Design concerns

- How clear & informative is the specification to a reader?
- Is the specification strong enough to prevent defects?
- Is the specification weak enough to allow flexibility?
- How early will defects be found? (Early in execution? Early in design?)
Specifications – design concerns

Design concerns

- How clear & informative is the specification to a reader?
- Is the specification strong enough to prevent defects?
- Is the specification weak enough to allow flexibility?
- How early will defects be found? (Early in execution? Early in design?)
- Do you want to place more burden on the client or the provider?
Specifications – design concerns

Design concerns

- How clear & informative is the specification to a reader?
- Is the specification strong enough to prevent defects?
- Is the specification weak enough to allow flexibility?
- How early will defects be found? (Early in execution? Early in design?)
- Do you want to place more burden on the client or the provider?
  - Originally, Postel’s law was regarded highly
    Be conservative in what you do. Be liberal in what you accept.
Specifications – design concerns

Design concerns

- How clear & informative is the specification to a reader?
- Is the specification strong enough to prevent defects?
- Is the specification weak enough to allow flexibility?
- How early will defects be found? (Early in execution? Early in design?)
- Do you want to place more burden on the client or the provider?
  - Originally, Postel’s law was regarded highly
    Be conservative in what you do. Be liberal in what you accept.
  - This is now regarded as problematic, poorly maintainable, & prone to security problems
In some cases, design can be simplified by saying that something always holds for a component.
In some cases, design can be simplified by saying that something always holds for a component
- These pointers are never null
- This collection is never empty
- The value {'a', 'b', 'c', ...} will always be present in a collection
Invariants

- In some cases, design can be simplified by saying that something always holds for a component
  - These pointers are never null
  - This collection is never empty
  - The value \{'a', 'b', 'c', ...\} will always be present in a collection

- An invariant is a condition that is always true
Invariants

- In some cases, design can be simplified by saying that something always holds for a component
  - These pointers are never null
  - This collection is never empty
  - The value {'a', 'b', 'c', ...} will always be present in a collection

- An invariant is a condition that is always true
  - Invariants may apply at different granularities & abstractions
    *class* invariants, *loop* invariants, *representation* invariants, ...
Invariants

- In some cases, design can be simplified by saying that something always holds for a component
  - These pointers are never null
  - This collection is never empty
  - The value {'a', 'b', 'c', ...} will always be present in a collection

- An invariant is a condition that is always true
  - Invariants may apply at different granularities & abstractions
    class invariants, loop invariants, representation invariants, ...

How are constructors related?
Invariants

• In some cases, design can be simplified by saying that something always holds for a component
  – These pointers are never null
  – This collection is never empty
  – The value {'a', 'b', 'c', ...} will always be present in a collection

• An invariant is a condition that is always true
  – Invariants may apply at different granularities & abstractions class invariants, loop invariants, representation invariants, ...

• Invariants can help you leverage inductive reasoning to simplify design
  – They can also give a bit of rigour to otherwise ad hoc code
Invariants

- In some cases, design can be simplified by saying that something always holds for a component
  - These pointers are never null
  - This collection is never empty
  - The value {‘a’, ‘b’, ‘c’, …} will always be present in a collection

- An invariant is a condition that is always true
  - Invariants may apply at different granularities & abstractions
    - class invariants, loop invariants, representation invariants, ...

- Invariants can help you leverage inductive reasoning to simplify design
  - They can also give a bit of rigour to otherwise ad hoc code

In fact, I’ve used invariants to help design some of the demos we’ve seen in class!
In some cases, design can be simplified by saying that something always holds for a component:
- These pointers are never null
- This collection is never empty
- The value {'a', 'b', 'c', ...} will always be present in a collection

An invariant is a condition that is always true:
- Invariants may apply at different granularities & abstractions

Invariants can help you leverage inductive reasoning to simplify design:
- They can also give a bit of rigour to otherwise ad hoc code

```java
//@ ghost boolean containsValue = 
(exists int i; 0 <= i < sortedArray.length; sortedArray[i] == value);
if (value < sortedArray[0]) return -1;
if (value > sortedArray[sortedArray.length-1]) return -1;
int lo = 0;
int hi = sortedArray.length-1;
//@ loop_invariant 0 <= lo < sortedArray.length
&& 0 <= hi < sortedArray.length;
//@ loop_invariant containsValue ==> 
    sortedArray[lo] <= value <= sortedArray[hi];
//@ loop_invariant \forall int i; 0 <= i < lo; sortedArray[i] < value;
//@ loop_invariant \forall int i; hi < i < sortedArray.length;
    value < sortedArray[i];
//@ loop_decreases hi - lo;
while (lo <= hi) {
    int mid = lo + (hi-lo)/2;
    if (sortedArray[mid] == value) {
        return mid;
    } else if (sortedArray[mid] < value) {
        lo = mid+1;
    } else {
        hi = mid-1;
    }
}
return -1;
```
Invariants

● In some cases, design can be simplified by saying that something always holds for a component – These pointers are never null
  – This collection is never empty
  – The value \{'a', 'b', 'c', ...\} will always be present in a collection

● An invariant is a condition that is always true – Invariants may apply at different granularities & abstractions class, loop invariants, representation invariants, ...

● Invariants can help you leverage inductive reasoning to simplify design – They can also give a bit of rigour to otherwise ad hoc code

```java
//@ ghost boolean containsValue =
(exists int i; 0 <= i < sortedArray.length; sortedArray[i] == value);
if (value < sortedArray[0]) return -1;
if (value > sortedArray[sortedArray.length-1]) return -1;
int lo = 0;
int hi = sortedArray.length-1;
//@ loop_invariant 0 <= lo < sortedArray.length
&& 0 <= hi < sortedArray.length;
//@ loop_invariant containsValue ==>
    sortedArray[lo] <= value <= sortedArray[hi];
//@ loop_invariant \forall int i; 0 <= i < lo; sortedArray[i] < value;
//@ loop_invariant \forall int i; hi < i < sortedArray.length;
    value < sortedArray[i];
//@ loop_decreases hi - lo;
while (lo <= hi) {
    int mid = lo + (hi-lo)/2;
    if (sortedArray[mid] == value) {
        return mid;
    } else if (sortedArray[mid] < value) {
        lo = mid+1;
    } else {
        hi = mid-1;
    }
}
return -1;
```
Invariants

- In some cases, design can be simplified by saying that something always holds for a component:
  - These pointers are never null
  - This collection is never empty
  - The value {'a', 'b', 'c', ...} will always be present in a collection

- An invariant is a condition that is always true:
  - Invariants may apply at different granularities & abstractions
  - For example, class invariants, loop invariants, representation invariants, ...

- Invariants can help you leverage inductive reasoning to simplify design.
  - They can also give a bit of rigour to otherwise ad hoc code.

```java
//@ ghost boolean containsValue =
    (\exists int i; 0 <= i < sortedArray.length; sortedArray[i] == value);
if (value < sortedArray[0]) return -1;
if (value > sortedArray[sortedArray.length-1]) return -1;
int lo = 0;
int hi = sortedArray.length-1;
//@ loop_invariant 0 <= lo < sortedArray.length
    && 0 <= hi < sortedArray.length;
//@ loop_invariant containsValue ==> sortedArray[lo] <= value <= sortedArray[hi];
//@ loop_invariant \forall int i; 0 <= i < lo; sortedArray[i] < value;
//@ loop_invariant \forall int i; hi < i < sortedArray.length;
    value < sortedArray[i];
//@ loop_decreases hi - lo;
while (lo <= hi) {
    int mid = lo + (hi-lo)/2;
    if (sortedArray[mid] == value) {
        return mid;
    } else if (sortedArray[mid] < value) {
        lo = mid+1;
    } else { // value > sortedArray[mid]
        hi = mid-1;
    }
} return -1;
```
In some cases, design can be simplified by saying that something always holds for a component:

- These pointers are never null
- This collection is never empty
- The value {'a', 'b', 'c', ...} will always be present in a collection

An invariant is a condition that is always true. Invariants may apply at different granularities & abstractions:

- class invariants,
- loop invariants,
- representation invariants, ...

Invariants can help you leverage inductive reasoning to simplify design. They can also give a bit of rigour to otherwise ad hoc code.

```java
//@ ghost boolean containsValue =
    (∃ int i; 0 ≤ i < sortedArray.length; sortedArray[i] == value);
if (value < sortedArray[0]) return -1;
if (value > sortedArray[sortedArray.length-1]) return -1;
int lo = 0;
int hi = sortedArray.length-1;

//@ loop_invariant 0 ≤ lo < sortedArray.length
    && 0 ≤ hi < sortedArray.length;
//@ loop_invariant containsValue ==> sort
    edArray[lo] ≤ value ≤ sortedArray[hi];
//@ loop_invariant \forall int i; 0 ≤ i < lo; sortedArray[i] < value;
//@ loop_invariant \forall int i; hi < i < sortedArray.length;
    value < sortedArray[i];
//@ loop_decreases hi - lo;
while (lo <= hi) {
    int mid = lo + (hi-lo)/2;
    if (sortedArray[mid] == value) {
        return mid;
    } else if (sortedArray[mid] < value) {
        lo = mid+1;
    } else {
        hi = mid-1;
    }
}
return -1;
```
Once you have agreement on a contract, you must decide how to manage it.
Once you have agreement on a contract, you must decide how to manage it.

No matter which philosophy you choose, your still want to find & report errors as soon as possible.
Enforcement OR Dealing with errors

- Once you have agreement on a contract, you must decide how to manage it.
- No matter which philosophy you choose, you still want to find & report errors as soon as possible.
- Major philosophies at extremes:
Once you have agreement on a contract, you must decide how to manage it.

No matter which philosophy you choose, you still want to find & report errors as soon as possible.

Major philosophies at extremes:
  - Provider must ensure consistency of the component
Enforcement OR Dealing with errors

- Once you have agreement on a contract, you must decide how to manage it.
- No matter which philosophy you choose, your still want to find & report errors as soon as possible.
- Major philosophies at extremes:
  - Provider must ensure consistency of the component
  - The client must fulfill its obligations in order to use the component
Design by contract (obligation of the client)

- You document & formalize a the contract
- A component may assume that its preconditions hold
Design by contract (obligation of the client)

- You document & formalize a the contract
- A component may assume that its preconditions hold
- The client may use the strong contract to guard program behavior early & enforce consistency
- If a violation occurs, the contracts may be used to guide debugging
Defensive programming (obligation of provider)

- The component author includes all checks necessary for correctness.
- If a contract is violated at runtime, then the author notifies the client via some error mechanism.
Trade offs & Implementations

- Design by contract usually has fewer checks in practice
Trade offs & Implementations

- Design by contract usually has fewer checks in practice
  - They can be easier to maintain
  - There are lower performance overheads
  - Assumptions of one component may be hoisted through many
  - There can be greater risks without static enforcement
Trade offs & Implementations

- Design by contract usually has fewer checks in practice
  - They can be easier to maintain
  - There are lower performance overheads
  - Assumptions of one component may be hoisted through many
  - There can be greater risks without static enforcement

- Defensive programming usually has more checks
Trade offs & Implementations

- Design by contract usually has fewer checks in practice
  - They can be easier to maintain
  - There are lower performance overheads
  - Assumptions of one component may be hoisted through many
  - There can be greater risks without static enforcement

- Defensive programming usually has more checks
  - Can occlude the meaning of the business logic
  - Errors are typically only at runtime
  - It is easier to locally guarantee, e.g. safety & security.
Trade offs & Implementations

- Design by contract usually has fewer checks in practice
  - They can be easier to maintain
  - There are lower performance overheads
  - Assumptions of one component may be hoisted through many
  - There can be greater risks without static enforcement

- Defensive programming usually has more checks
  - Can occlude the meaning of the business logic
  - Errors are typically only at runtime
  - It is easier to locally guarantee, e.g. safety & security.

- Frequently in practice:
  - Assertions
  - Exceptions
Failing fast

- Using either philosophy, you prefer to fail as early as possible.
  - Prevent the corruption of state
  - Observation of a defect will be closer to the cause
Failing fast

- Using either philosophy, you prefer to fail as early as possible.
  - Prevent the corruption of state
  - Observation of a defect will be closer to the cause

- This leads to common patterns...
  - Validate user input before starting to process it
  - Check where API invocations may violate invariants & throw
Failing fast

- Using either philosophy, you prefer to fail as early as possible.
  - Prevent the corruption of state
  - Observation of a defect will be closer to the cause

- This leads to common patterns...
  - Validate user input before starting to process it
  - Check where API invocations may violate invariants & throw

```java
List<Integer> integers = new ArrayList<>(1, 2, 3);
for (Integer integer : integers) {
    integers.remove(1);
}
```

[Baeldung 2019]
Failing fast

- Using either philosophy, you prefer to fail as early as possible.
  - Prevent the corruption of state
  - Observation of a defect will be closer to the cause
- This leads to common patterns...
  - Validate user input before starting to process it
  - Check where API invocations may violate invariants & throw

```java
List<Integer> integers = new ArrayList(1, 2, 3);
for (Integer integer : integers) {
    integers.remove(1);
}
```

How may these patterns relate to software architecture?
Assertions

- Assertions follow a design by contract idiom
Assertions

- Assertions follow a design by contract idiom
  - Not checked during a normal build
  - Check whether a condition is true and terminate the program
  - Used for documentation, debugging, & testing
Assertions

- Assertions follow a design by contract idiom
  - Not checked during a normal build
  - Check whether a condition is true and terminate the program
  - Used for documentation, debugging, & testing

- The exact relationship between asserts & defects is nuanced but there is some evidence that they decrease defect rates
Assertions

- Assertions follow a design by contract idiom
  - Not checked during a normal build
  - Check whether a condition is true and terminate the program
  - Used for documentation, debugging, & testing
- The exact relationship between asserts & defects is nuanced but there is some evidence that they decrease defect rates

```cpp
#include <cassert>
constexpr Image ascii[256] = ...

Image& getCharGlyph(int asciiCode) {
    assert(0 < asciiCode && asciiCode < 256 && "ASCII code out of range.");
    return ascii[asciiCode];
}
```
Exceptions

- Exceptions typically follow a defensive programming strategy
  - A component will check that the spec is satisfied at its boundaries
  - An exception is thrown when the spec is violated
Exceptions

- Exceptions typically follow a defensive programming strategy
  - A component will check that the spec is satisfied at its boundaries
  - An exception is thrown when the spec is violated

- **NOTE**: One trend is to use exceptions for normal control flow. Prefer to avoid this.
Exceptions

- Exceptions typically follow a defensive programming strategy
  - A component will check that the spec is satisfied at its boundaries
  - An exception is thrown when the spec is violated
- **NOTE:** One trend is to use exceptions for normal control flow. Prefer to avoid this.
  - Exceptions are for exceptional circumstances
  - Both assertions & exceptions should be used with input validation at the boundaries of an interface!
Exceptions

- Exceptions typically follow a defensive programming strategy
  - A component will check that the spec is satisfied at its boundaries
  - An exception is thrown when the spec is violated
- NOTE: One trend is to use exceptions for normal control flow. Prefer to avoid this.
  - Exceptions are for exceptional circumstances
  - Both assertions & exceptions should be used with input validation at the boundaries of an interface!
- Exact exception semantics differ across languages, but prefer to
  1) catch & manage specific exception types
  2) consider exceptions hard failures
Logging

- In practice, there is often not much you can do to recover from spec violations
  - Termination is often the right thing
  - But termination itself can be an error in some circumstance
  - Abruptly terminating may also make debugging challenging
Logging

- In practice, there is often not much you can do to recover from spec violations
  - Termination is often the right thing
  - But termination itself can be an error in some circumstance
  - Abruptly terminating may also make debugging challenging

- In practice, companies prefer to use logging
  - Maybe the absence of behavior was erroneous
  - Maybe a trend is erroneous
  - Maybe an error only happens when deployed
Logging

- In practice, there is often not much you can do to recover from spec violations
  - Termination is often the right thing
  - But termination itself can be an error in some circumstance
  - Abruptly terminating may also make debugging challenging
- In practice, companies prefer to use logging
  - Maybe the absence of behavior was erroneous
  - Maybe a trend is erroneous
  - Maybe an error only happens when deployed
- A logging system records program state & events over time.
Logging

LOG(INFO) << "Creating new account. "
<< "name:" << username;
LOG(INFO) << "Creating new account. "
<< "name:" << username;
LOG(INFO) 

<< "Creating new account. " 
<< "name:" << username;
Logging

LOG(INFO) << "Creating new account. "
    << "name:" << username;

LOG_IF(INFO, numUsers > 10)
    << "Many users logged in. "
    << "numusers:" << numUsers;
LOG(INFO) << "Creating new account. "
    << "name:" << username;

LOG_IF(INFO, numUsers > 10)
    << "Many users logged in. "
    << "numusers:" << numUsers;

CHECK_LT(index, size) << "Index out of bounds."
CHECK_NOTNULL(ptr);
Logging

- A logging system records program state & events over time.
- **Common to log:** [Fu et al., ICSE 2014]
Logging

- A logging system records program state & events over time.

- **Common to log:** [Fu et al., ICSE 2014]
  - Assertion failures
  - Critical return values
  - Exceptions

\[ \text{Unexpected Situations} \]
Logging

- A logging system records program state & events over time.
- **Common to log**: [Fu et al., ICSE 2014]
  - Assertion failures
  - Critical return values
  - Exceptions
  - Key branch points
  - Observation points

{ Unexpected Situations }
{ Key Execution Points }
• A logging system records program state & events over time.

• Common to log: [Fu et al., ICSE 2014]
  - Assertion failures
  - Critical return values
  - Exceptions
  - Key branch points
  - Observation points

  \{ Unexpected Situations \\
  \{ Key Execution Points \}

• Logging \textbf{too little} or \textbf{too much} can be a problem
  - Might miss what you want
  - Might create a haystack for your needle
  - Might spend too many resources!
Logging Guidelines

- Log all assertion failures
Logging Guidelines

● Log all assertion failures

● Log exceptions *at most once*
  – Might defer logging if exception is rethrown
  – Might skip logging exceptions that do no harm (e.g. if deleting a file failed because it was not there)
Logging Guidelines

- Log all assertion failures
- Log exceptions *at most once*
  - Might defer logging if exception is rethrown
  - Might skip logging exceptions that do no harm (e.g. if deleting a file failed because it was not there)
- Log all events needed for auditing
Logging Guidelines

- Log all assertion failures
- Log exceptions \textit{at most once}
  - Might defer logging if exception is rethrown
  - Might skip logging exceptions that do no harm
    (e.g. if deleting a file failed because it was not there)
- Log all events needed for auditing
- Log logic that provides \textit{context} for possible errors
Logging Guidelines

- Log all assertion failures
- Log exceptions *at most once*
  - Might defer logging if exception is rethrown
  - Might skip logging exceptions that do no harm
    (e.g. if deleting a file failed because it was not there)
- Log all events needed for auditing
- Log logic that provides context for possible errors
- **Make your log easy to use**
  - Machine parsable if possible
  - *What / When / Why / Where* should be clearly captured
Summary

- Specification can be a powerful tool for reasoning about program correctness
- You can apply a specification using
  - Design by contract (client managed)
  - Defensive programming (provider managed)
- Logging provides a key mechanism for getting more value out of specifications in practice