Object Oriented Specification and Refinement in *Perfect Developer*

David Crocker

Escher Technologies Ltd.

PERFECT DEVELOPER
Making software bugs extinct!

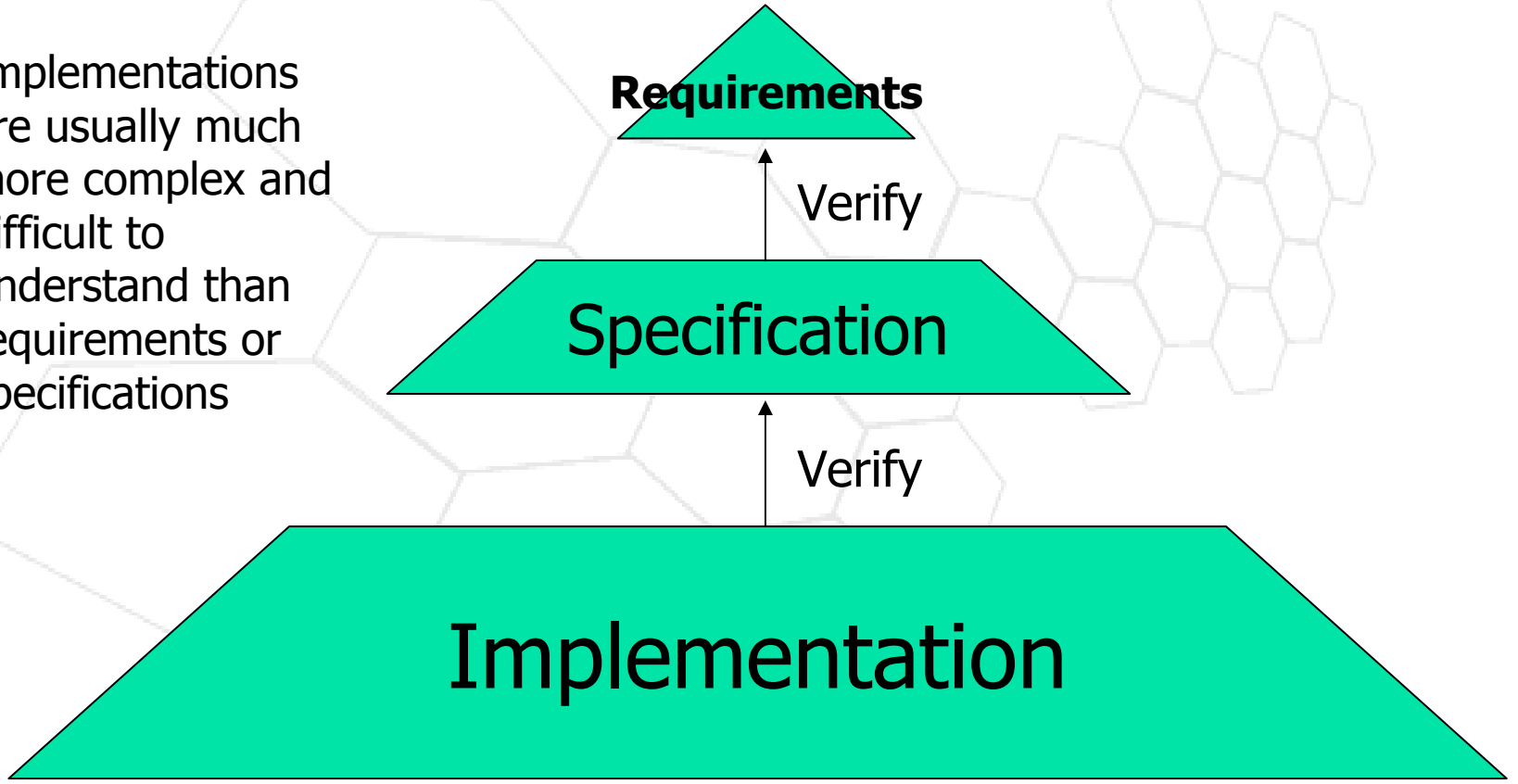
ESCHER TECHNOLOGIES

Structure of this presentation

- How to use *Perfect Developer*
- An example of specifying a component
- An example of refining a component

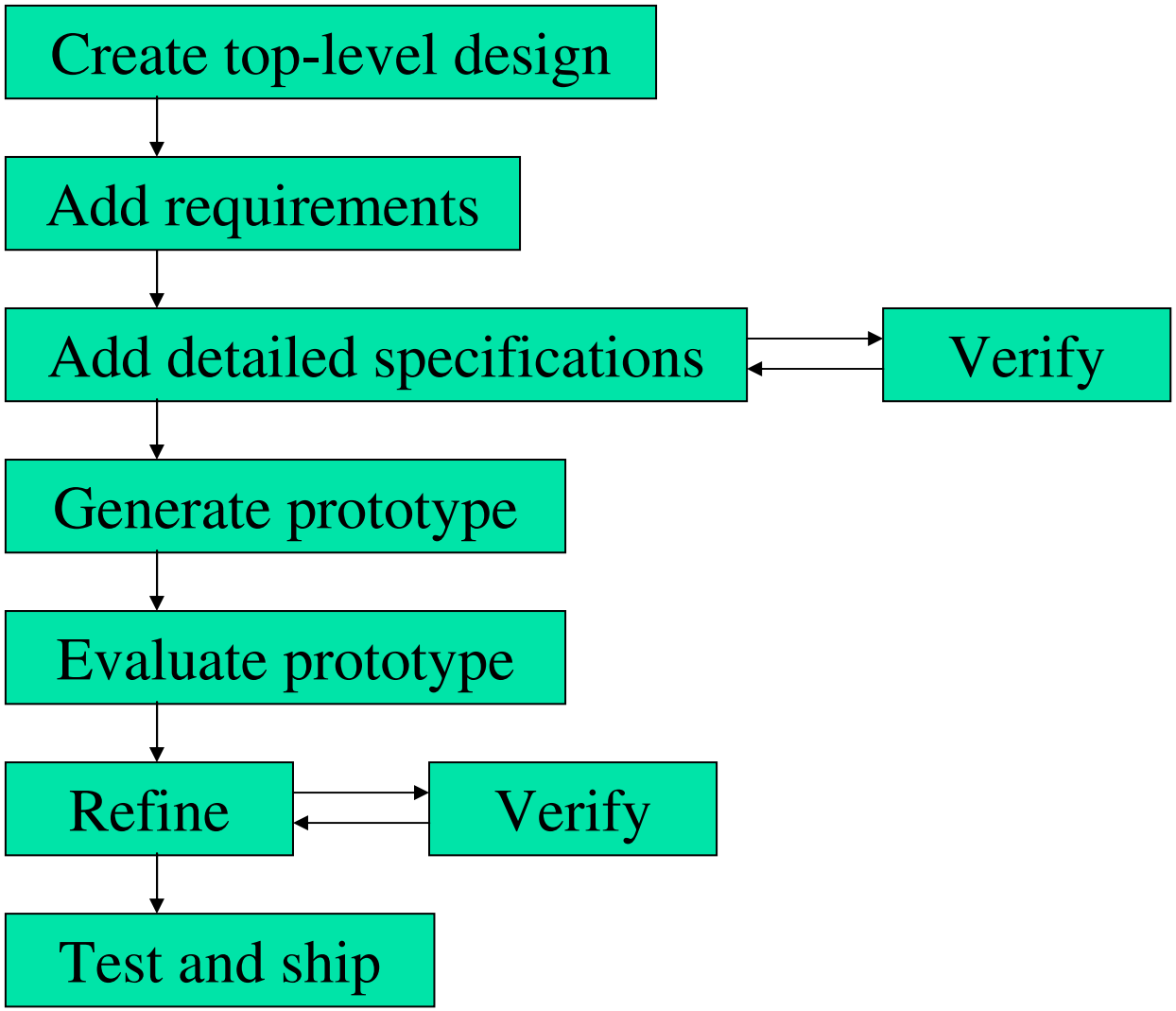
How *Perfect Developer* manages complexity

Implementations are usually much more complex and difficult to understand than requirements or specifications



ESCHER TECHNOLOGIES

How to use Perfect Developer



Using *Perfect Developer* (1)

- Establish the requirements
- Plan the system and its interaction with users (using UML tools or by hand)
- Define the classes, method signatures and requirements in *Perfect*
- Model the data manipulated by each class

Using *Perfect Developer* (2)

- Write the method specifications
- Verify that the specifications meet the requirements
- Generate a prototype
- Test the prototype for usability

Using *Perfect Developer* (3)

- Identify those aspects of the prototype where better performance is needed
- Refine the abstract model to an implementation model, where needed
- Refine the specifications to efficient implementations, where needed
- Verify the refinements

Example and demonstration

- A suite of applications needs a spell-checker module
- We have been asked to develop the dictionary component for this module

Dictionary requirements (1)

- The dictionary shall store words only (without any associated semantics)
- It must provide means to:
  - Create a new, empty dictionary
  - Test whether a word is in the dictionary
  - Add a word to the dictionary (the word must not already be in it)
  - Remove a word from the dictionary (the word must already be in it)

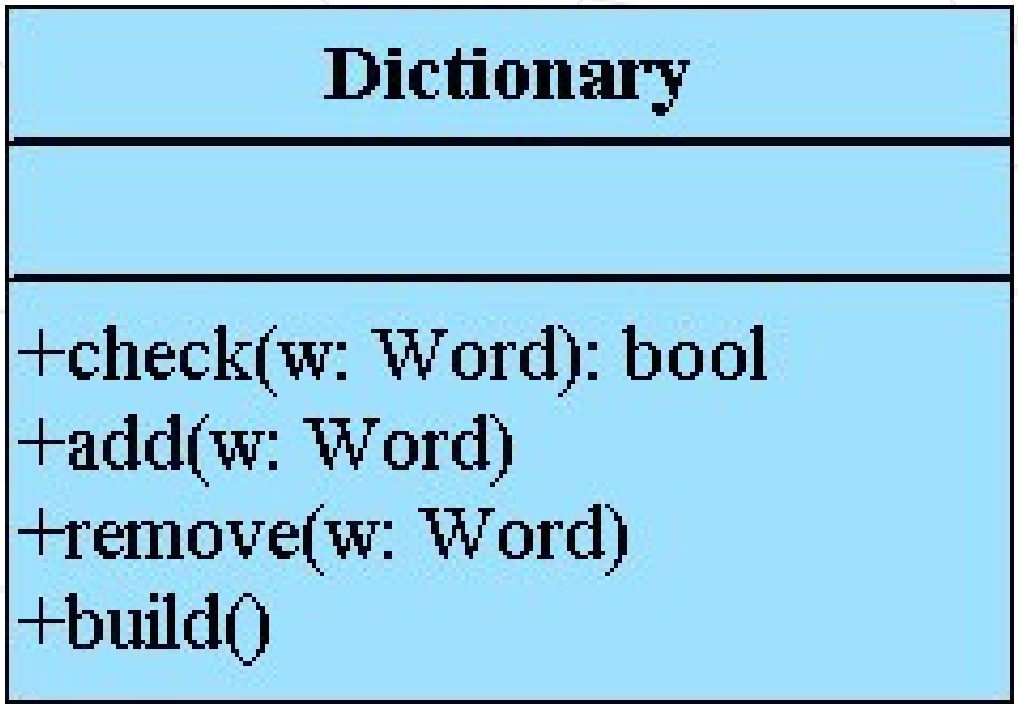
Dictionary requirements (2)

- By “add” we mean that after adding a word, it is present in the dictionary
- By “remove” we mean that after removing a word it is no longer in the dictionary
- An “add” can be undone by a “remove” of the same word
- Adding or removing a word does not affect whether a different word is in the dictionary
- No word is in an “empty” dictionary

Creating the top-level design

- Create class outlines and method signatures
- These can be imported from a UML model

UML class diagram for Dictionary



ESCHER TECHNOLOGIES

```
// Define the form of word that we wish to spell-check
class Word ^= those x:seq of char :- #x ~= 0;


// The dictionary class
class Dictionary ^=
abstract
  ?
interface
  function check(w: Word): bool     // check whether a word is present
    ^= ?;
  schema !add(w: Word)              // add a word
    pre ~check(w)                   // the word is not already there
    post ?;
  schema !remove(w: Word)           // remove a word
    pre check(w)                    // the word is already there
    post ?;
  build{}                           // build an empty dictionary
    post ?;
end;
```

Adding the requirements

- Determine functional requirements, expected behaviour, safety properties
- Express them in *Perfect*
  - Express a requirement concerning a single method as a *post-assertion*
  - Express a requirement concerning multiple methods as a *property* or *ghost schema*

```
class Dictionary ^=
abstract
  ?

interface

  function check(w: Word): bool     // check whether a word is present
    ^= ?;

  schema !add(w: Word)              // add a word
    pre ~check(w)                   // the word is not already there
    post ?
    assert self'.check(w);          // afterwards, the word is there

  schema !remove(w: Word)           // remove a word
    pre check(w)                    // the word is already there
    post ?
    assert ~self'.check(w);         // afterwards, it is not there

  build{}                           // build an empty dictionary
    post ?
    assert forall x: Word :- ~self'.check(x);   // it really is empty

  // If we add and then remove a new word, we get back to the start
  property(w: Word)
    pre ~check(w)
    assert self after it!add(w) then it!remove(w) = self;

  // Adding one word does not affect whether other words are included
  property(w1, w2: Word)
    pre ~check(w1), w1 ~= w2
    assert (self after it!add(w1)).check(w2) = self.check(w2);

end;
```

Adding detailed specifications

- Define an abstract data model of each class
- Add the method specifications
- Verify that the specifications meet the requirements

Predefined collection types

- **set of** X
- **bag of** X
- **seq of** X
- **map of** (X -> Y)

```
class Dictionary ^=
abstract
  var words: set of Word;

interface

  function check(w: Word): bool     // check whether a word is present
    ^= ?;

  schema !add(w: Word)              // add a word
    pre ~check(w)                   // the word is not already there
    post ?
    assert self'.check(w);          // afterwards, the word is there

  schema !remove(w: Word)           // remove a word
    pre check(w)                    // the word is already there
    post ?
    assert ~self'.check(w);         // afterwards, it is not there

  build{}                           // build an empty dictionary
    post ?
    assert forall x: Word :- ~self'.check(x);

  // If we add and then remove a new word, we get back to the start
  property(w: Word)
    pre ~check(w)
    assert self after it!add(w) then it!remove(w) = self;

  // Adding one word does not affect whether other words are included
  property(w1, w2: Word)
    pre ~check(w1), w1 ~= w2
    assert (self after it!add(w1)).check(w2) = self.check(w2);

end;
```

Adding detailed specifications

- Define an abstract data model of each class
- Add the method specifications
- Verify that the specifications meet the requirements

```
class Dictionary ^=
abstract
  var words: set of Word;

interface

  function check(w: Word): bool     // check whether a word is present
    ^= w in words;

  schema !add(w: Word)              // add a word
    pre ~check(w)                   // the word is not already there
    post words! = words.append(w)
    assert self'.check(w);          // afterwards, the word is there

  schema !remove(w: Word)           // remove a word
    pre check(w)                    // the word is already there
    post words! = words.remove(w)
    assert ~self'.check(w);         // afterwards, it is not there

  build{}                           // build an empty dictionary
    post words! = set of Word{}
    assert forall x: Word :- ~self'.check(x);

  // If we add and then remove a new word, we get back to the start
  property(w: Word)
    pre ~check(w)
    assert self after it!add(w) then it!remove(w) = self;

  // Adding one word does not affect whether other words are included
  property(w1, w2: Word)
    pre ~check(w1), w1 ~= w2
    assert (self after it!add(w1)).check(w2) = self.check(w2);

end;
```

Postconditions in *Perfect*

- A *Perfect* postcondition defines:
  - The condition(s) that the final state must satisfy
  - The frame (which components may be modified)
- The most general form of postcondition is:

  change *variables* **satisfy** *predicate-list*
- The postcondition v! = e is a shorthand for:

  change v **satisfy** v'= e

Adding detailed specifications

- Define an abstract data model of each class
- Add the method specifications
- Verify that the specifications meet the requirements

Live demonstration of verification (1)

Generating a prototype

- Use *Perfect Developer* to automatically refine the specifications to code
- Compile and link the code
- Verify that it meets the user requirements
- Determine where better performance is needed

Where is better performance needed?

- Use profiling interface to identify bottlenecks in generated C++ code
- For this example, we will assume that the memory usage needs to be reduced

Refining the prototype

- Refine the abstract model to an implementation model, where needed
- Refine the specifications to efficient implementations, where needed
- Verify the refinements

```
// Refined version of spell checker dictionary.

class Dictionary ^=
abstract
  var words: set of Word;

internal

  // Because most plurals in English end in -s, we choose to save
  // space by storing only the singular forms and indicating whether
  // the plural form can be used.

  var plainWords, specialWords: set of Word;

  // Define how the internal data maps to the abstract data "words"

  function words ^= plainWords ++ specialWords
                    ++ (for x:: specialWords yield x ++ "s");

  invariant

    // To keep the data small we avoid redundancy
    // Each word should only be stored in one place
    plainWords ** specialWords = set of Word{},
    forall x:: specialWords :- (x ++ "s") ~in plainWords,
    forall x:: specialWords :- (x ++ "s") ~in specialWords,

    // We should not store a word and its plural separately
    forall x:: plainWords :- (x ++ "s") ~in plainWords;

interface
```

Refining the prototype

- Refine the abstract model to an implementation model, where needed
- Refine the specifications to efficient implementations, where needed
- Verify the refinements

```
internal

  var plainWords, specialWords: set of Word;

  invariant

    // To keep the data small we avoid redundancy
    // Each word should only be stored in one place
    plainWords ** specialWords = set of Word{},
    forall x::specialWords :- (x ++ "s") ~in plainWords,
    forall x::specialWords :- (x ++ "s") ~in specialWords,

    // We should not store a word and its plural separately
    forall x::plainWords :- (x ++ "s") ~in plainWords;

  // Define how the internal data maps to the abstract data "words"

  function words ^= plainWords ++ specialWords
                  ++ (for x::specialWords yield x ++ "s");

interface

  function check(w: Word): bool
    ^= w in words
    via  // re-implement function result in terms of internal data
        value w in plainWords
            | w in specialWords
            | #w >= 2 & w.last = `s` & w.front in specialWords
    end;
```

```
schema !add(w: Word)
  pre ~check(w)
  post words! = words.append(w)
  via    // re-implement postcondition in terms of internal data
    if
    [#w >= 2 & w.last = `s` & w.front in plainWords]:
      // word ends in "s" and root is in dictionary
      plainWords! = plainWords.remove(w.front),
      specialWords! = specialWords.append(w.front);
    [w ++ "s" in plainWords]:
      // the plural form of the word is already in the dictionary
      plainWords! = plainWords.remove(w ++ "s"),
      specialWords! = specialWords.append(w);
    []:
      // Plural is not in dictionary, or it is and can itself be
      // pluralised,
      // and word does not end in 's' or root is not in dictionary
      plainWords! = plainWords.append(w);
    fi
  end
  assert self'.check(w);
```

```
schema !remove(w: Word)
  pre check(w)
  post words! = words.remove(w)
  via
    if
    [w in plainWords]:
      // the simple case
      plainWords! = plainWords.remove(w);
    [w in specialWords]:
      // plural form is also in dictionary so we need to add it back
      specialWords! = specialWords.remove(w);
      !add(w ++ "s");
    [#w >= 2 & w.last = `s` & w.front in specialWords]:
      // word is a plural form so remove it and add back the singular
      specialWords! = specialWords.remove(w.front);
      !add(w.front);
    fi
  end
  assert ~self'.check(w);
```

Refining the prototype

- Refine the abstract model to an implementation model, where needed
- Refine the specifications to efficient implementations, where needed
- Verify the refinements

Live demonstration of verification (2)

What have we achieved?

- We started with some requirements
- We developed a simple specification using an abstract data model
- A simple refinement led to a much more complex implementation
- *Perfect Developer* verified each stage

Further information

- ***www.eschertech.com*** has more information including the Language Reference Manual and papers
- Ask for an evaluation copy

ESCHER TECHNOLOGIES