Testing

Abram Hindle
hindle1@ualberta.ca

Henry Tang
hkktang@ualberta.ca

Department of Computing Science
University of Alberta

CMPUT 301 – Introduction to Software Engineering
Slides adapted from Dr. Hazel Campbell, Dr. Ken Wong

Goal

- Does program P obey specification S?
  - What is P?
  - What is S?

Approaches

- Reasoning about the state model for P:
  - Typically, a huge number of states
  - Every practical technique must be inaccurate
  - Could *abstract* states
  - Could *sample* states
  - Or both

Approaches

- Abstraction:
  - Often used in static software analysis techniques
    - E.g., model checking P for some specific S
  - Techniques often pessimistically inaccurate
    - May report P is faulty when P is correct

Approaches

- Sampling:
  - Often used in dynamic analysis techniques
    - E.g., testing, profiling
  - Techniques often optimistically inaccurate
    - May report P is correct when P is faulty
    - Testing drives P through a sampling of states, but the samples may not generalize to actual situations

Software Defects

- Some terms:
  - Human *errors* can lead to *faults* in work products, which may cause *failures* when running the software
  - Can try to find faults through *testing*, reviews, proof, model checking, code analysis, etc.
  - Some avoid the term *bug*, since it implies something wandered into the code

Examples of Defects

- Actual behavior differing from expected:
  - Algorithmic
    - Code logic does not produce the proper output
  - Overload
    - Data structure unexpectedly filled completely
  - Performance
    - Violates service level agreement
  - Accuracy
    - Calculated result not to the desired level of accuracy
  - Timing
    - Race condition in coordinating concurrent processes

Failure

- AT&T failure (1990):
  - 114 switching nodes of their long-distance system crashed
  - The outage lasted for 9 h, 70 million calls went uncompleted
- Reason:
  - If a node crashes, it tells neighboring nodes to reroute traffic around it
  - A bug in handling this message caused the receiving node to also crash, etc.

Fault in Code

- Root cause:

```
do {
    switch (…) {
    case …:
        if (…) {
            …
            break;
        } else {
            …
        }
    …
    }
} while (…);
```

After expensive testing phase, a small change was made without again retesting

Why Test?

- Goals:
  - Verification
    - Check that requirements are satisfied
  - Not only to *confirm* normal behavior
    - Find problems to *refute* that the program is correct
  - Establish due diligence
    - Evidence in case of product liability litigation
  - Avoid regression
    - Prevent previous problems from reoccurring

Regression Testing

- Goal:
  - To avoid breaking things that should work
    - Collect, reuse, and re-run automated test cases
  - Do regression test after a change or fix
    - Re-run tests to check whether previously passing tests of the system now fail
    - E.g., old defect somehow became unfixed

Limits of Testing

- Issues:
  - A program cannot be tested completely
    - Too many inputs and path combinations to cover
  - Testing cannot find all defects
    - Cannot show their absence, just their presence
  - Challenging
    - Testing may be expensive and frustrating
    - Test code itself could add its own defects

Test-Driven Development

Automated Testing

- Purpose:
  - Write software to help test software
    - Automation essential to test-driven development and refactoring
- Limitations:
  - Manual testing still need to observe certain problems
    - E.g., strange noises from the speaker, flickering graphics

Automated Testing

- A good automated unit test:
  - Is simple to write and understand
    - Reduces the chance of defects in the test code
  - Runs quickly
    - Allows it to be re-run frequently while developing
  - Is isolated
    - Could run multiple unit tests in parallel
  - Shows exactly what went wrong if it fails
    - Reduce time spent in a debugger

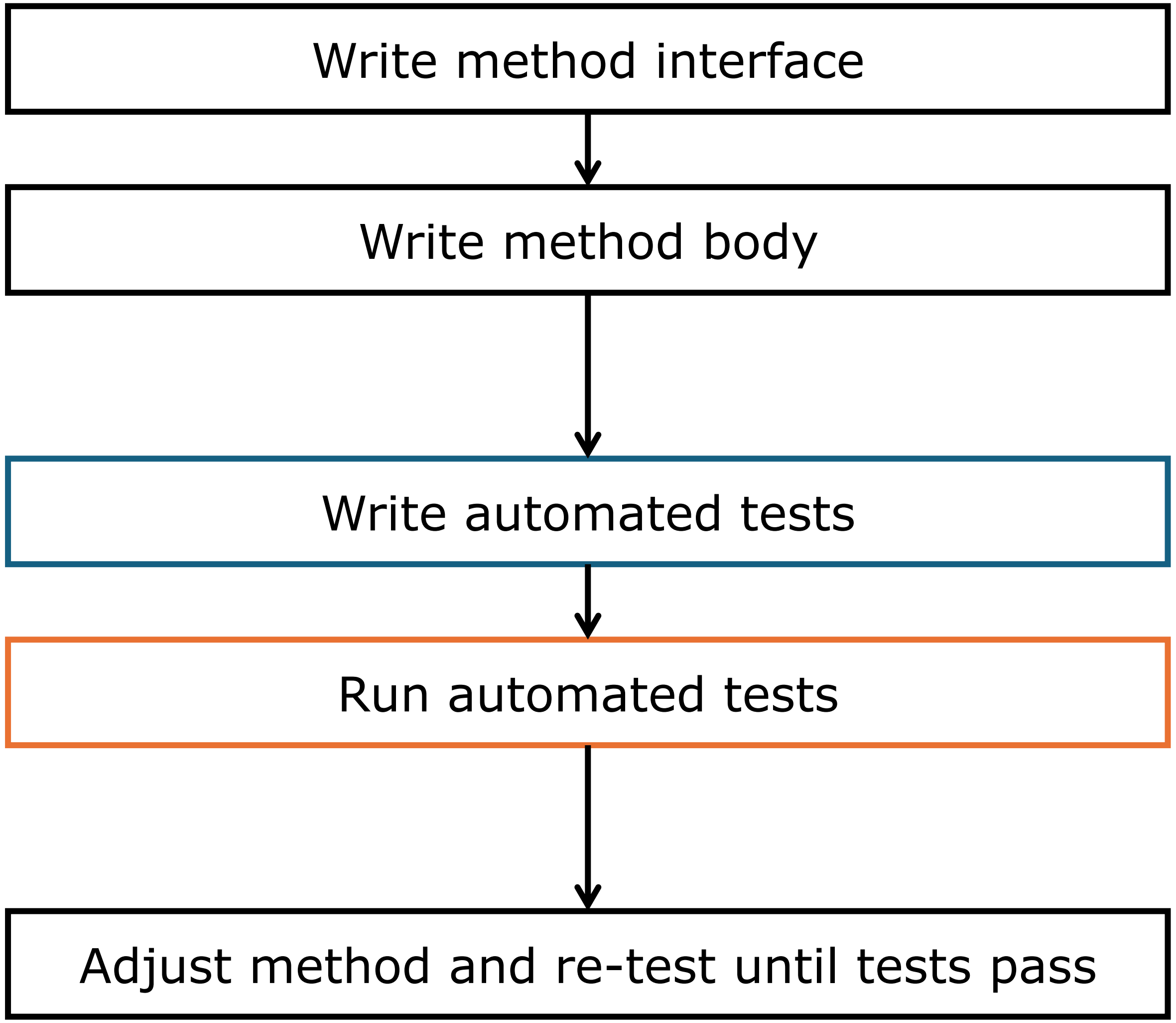
Automated Testing

- “Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.”
  — Martin Fowler

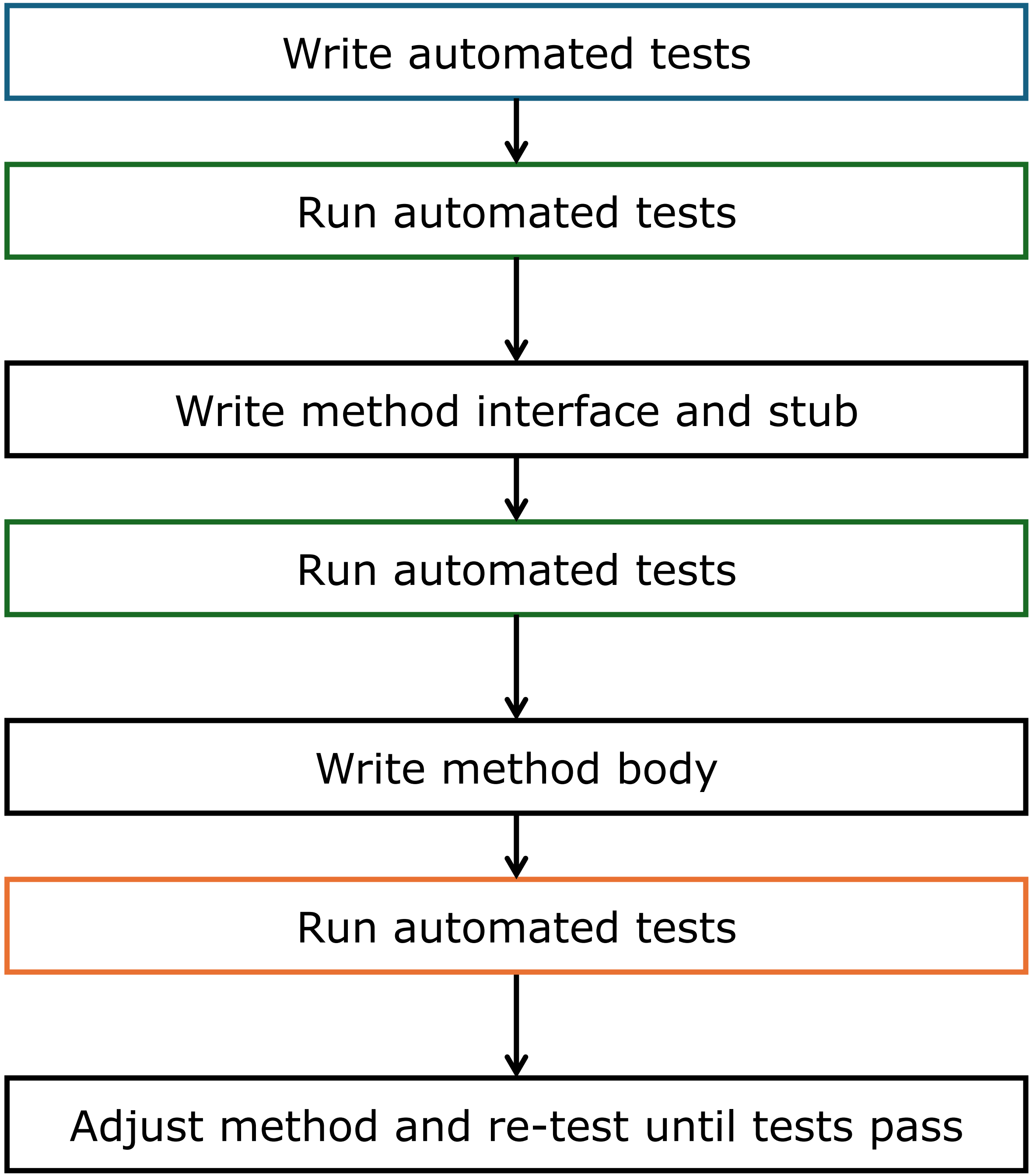
Test-Driven Development

- Idea:
  - If testing is so useful, let's write the tests first
  - These automated tests capture *code-level requirements* to be satisfied
  - Once code is written so that these tests pass, then these requirements are met

Traditional development



Test-first or test-driven development



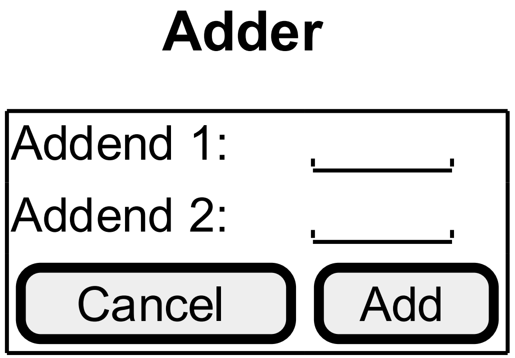
State-Based Testing

State-Based Testing

- Steps:
  - Set up software into a known state
    - E.g., initialize variables
  - Trigger transitions to cause state changes
    - E.g., call methods to change variables
    - E.g., interact with the user interface
  - Verify the actual arrived state is expected
    - E.g., see if actual values in variables meet expectations

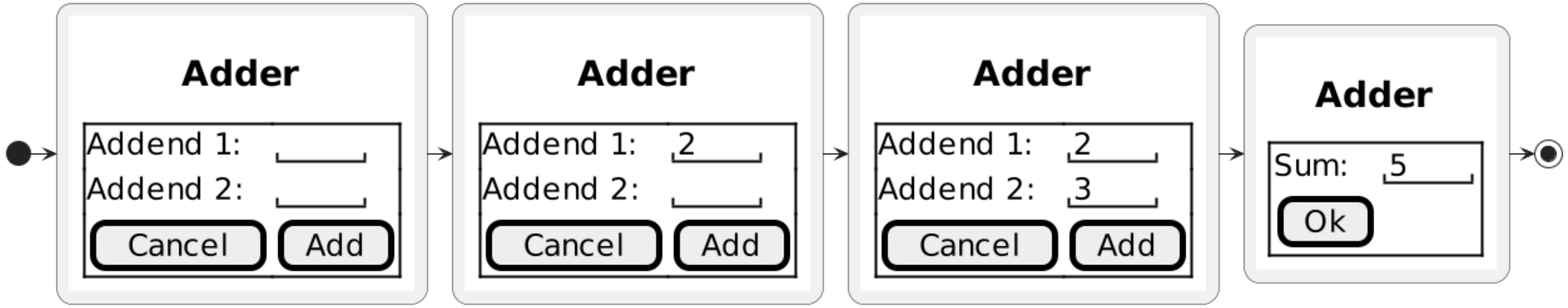
Black Box Testing

• Testing, without seeing the code:



Black Box Testing

- Expected behavior:



  - Deviations from the expected interaction?

Black Box Testing

- Tips:
  - Want be systematic about what to test
    - E.g., focus on the adder functionality for now
  - Avoid redundant tests
    - Too easy to keep adding meaningless extra tests
  - Determine *equivalence classes* of tests

Black Box Testing

- Equivalence classes:
  - Each test inside an equivalence class checks the “same thing”
  - If a test inside the class will catch a defect, the other tests probably also will
  - If a test inside the class will not catch a defect, the other tests probably also will not
  - Keep only a few tests in each class, as representatives

Black Box Testing

- Example test cases:
  - Be systematic about what to test, not knowing the internal code

	Addends	Sum	Description (also check commutative)
2	3	5	Something simple
99	99	198	Large positive pair
99	-14	85	Large positive plus negative
99	16	115	Large positive plus positive
-99	-99	-198	Large negative pair
-99	-14	-113	Large negative plus negative
-99	16	-83	Large negative plus positive
-99	99	0	Large positive plus large negative
9	9	18	Largest single digit positive pair

Black Box Testing

- Example test cases:
  - Guessing at internal algorithm or representation

	Addends	Sum	Description (also check commutative)
0	0	0	All zero special case
0	23	23	Zero plus positive
-78	0	-78	Negative plus zero
127	127	254	Max signed bytes
-128	127	-1	Min and max signed bytes
-128	-128	-256	Min signed bytes
2147483647	2147483647		Max signed integers
-2147483648	2147483647	-1	Min and max signed integers
-2147483648	-2147483648		Min signed integers

Black Box Testing

- Example test cases:
  - Data input from fields in user interface

	Addends	Sum	Category (also check commutative)
4/3	2		Expression
$2	$2		Currency symbols
+5	3		Addition sign
(9)	9		Parentheses around negatives
l	1		Lower case letter l
O	0		Upper case letter O
<tab>	<tab>		No input
1.2	5		Decimal
A	b		Invalid characters

Black Box Testing

- Example test cases:
  - And even more user interface explorations
    - Editing with delete, backspace, cursor keys, etc.
    - Using F1, escape, and control characters
    - Vary timing of data entry

Testing Strategies

- Big-bang strategy:
  - Test thoroughly only after the whole system is put together
  - Pro(?)
    - "Project almost finished, only testing left"
  - Cons
    - Hard to pinpoint the cause of a failure

Testing Strategies

- Top-down incremental strategy:
  - Implement/test the highest-level modules first
    - Provide stubs for lower-level functionality not yet implemented
    - Higher-level modules are the test drivers
- Bottom-up incremental strategy:
  - Implement/test the lowest-level modules first
    - Need to write test drivers

Testing Techniques

- Creating good tests:
  - Test every error message
    - Error-handling code tends to be weaker
  - Test under other configurations
    - Programmers are biased to their own setup

Design for Testing

Good Software Design

- Software should be flexible:
  - Easy to change to respond to new needs
  - Easy to understand
  - Easy to extend, without exploding complexity
- Software should be testable:
  - Easy to construct the units
  - Easy to set up units into desired state
  - Easy to drive code and witness effects

Example Bad Design 1

- ```
  /**
   * Process photo album requests,
   * parse user preferences,
   * apply image transformations,
   * assemble images into albums,
   * deliver results to users
   */

  public class PhotoAlbumServer {

       … // lots of code

  }
  ```
  ```

Example Bad Design 1

- Poor flexibility:
  - Difficult to extract and reuse parts
  - Complex to add new features
  - Instance variables are “global”
- Poor testability:
  - Only end-to-end testing possible
  - Need golden results files for every combination of preference settings and image transformations

Improved Design 1

- Use separation of concerns:
  - RequestHandler class
  - UserPreferencesReader class
  - UserPreferencesParser class
  - ImageEffect class
  - ImageTransformer class
  - ...

Improved Design 1

- Better flexibility:
  - Uses object-oriented design
  - Easier to understand smaller, separate units
- Better testability:
  - More focused tests of each unit
  - Test fixtures easier to provide for each unit
  - Easier to check results

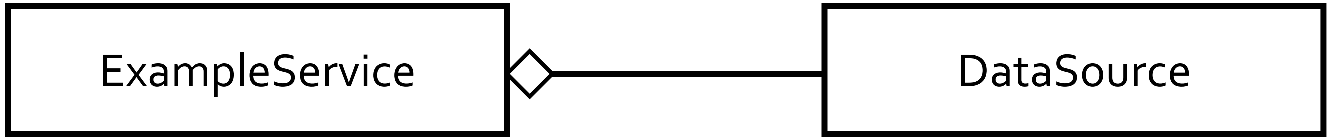
Forming Dependencies

```
• public class ExampleService {
      private DataSource theDataSource;
      …

      public ExampleService( … ) {
          theDataSource = new DataSource( … );
          …
      }

      public void doService() {
          …
          … = theDataSource.getInfo();
          …
      }
      …
  }
```

One approach is that the class makes what it depends on



“Dependency Injection”

- ```
  public class ExampleService {
      private DataSource theDataSource;
      …

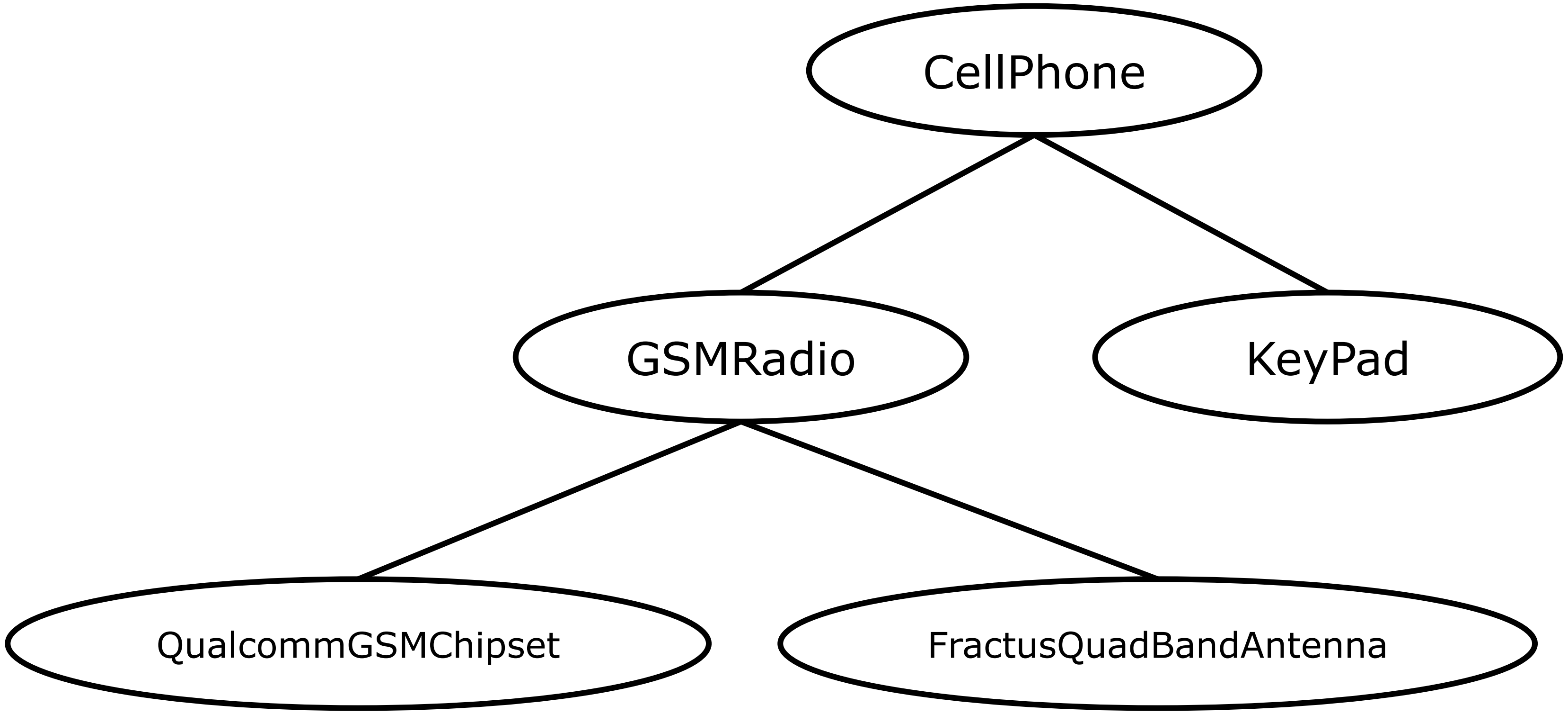
      public ExampleService(
          DataSource aDataSource ) {

          theDataSource = aDataSource;
          …
      }

      public void doService() {
          …
          … = theDataSource.getInfo();
          …
      }
      …
  }
  ```

Alternatively, construct what this class depends on outside the class
  ```

System Assembly



System Assembly without DI

- ```
  public class CellPhone {
      …
      public CellPhone() {
          radio = new GSMRadio();
          inputDevice = new KeyPad();
          …
      }
  }
  ```
- ```
  public class GSMRadio {
      …
      public GSMRadio() {
          chipset = new QualcommGSMChipset();
          antenna = new FractusQuadBandAntenna();
      }
  }
  ```
- ```
  CellPhone phone = new CellPhone();
  // fully assembled
  ```
  ```

System Assembly without DI

- Poor flexibility:
  - Difficult to change and plug in parts
    - For different radio, different input device, etc.
- Poor testability:
  - Can’t supply test versions of parts
    - Stuck with given parts
  - Entire aggregate is constructed
    - Could be expensive

System Assembly with DI

- ```
  public class CellPhone {
      …
      public CellPhone( Radio radio,
          InputDevice inputDevice ) {

          this.radio = radio;
          this.inputDevice = inputDevice;
      }
      …
  }
  ```
- ```
  public class GSMRadio extends Radio {
      …
      public GSMRadio( Chipset chipset,
          Antenna antenna ) {

          this.chipset = chipset;
          this.antenna = antenna;
      }
  }
  ```

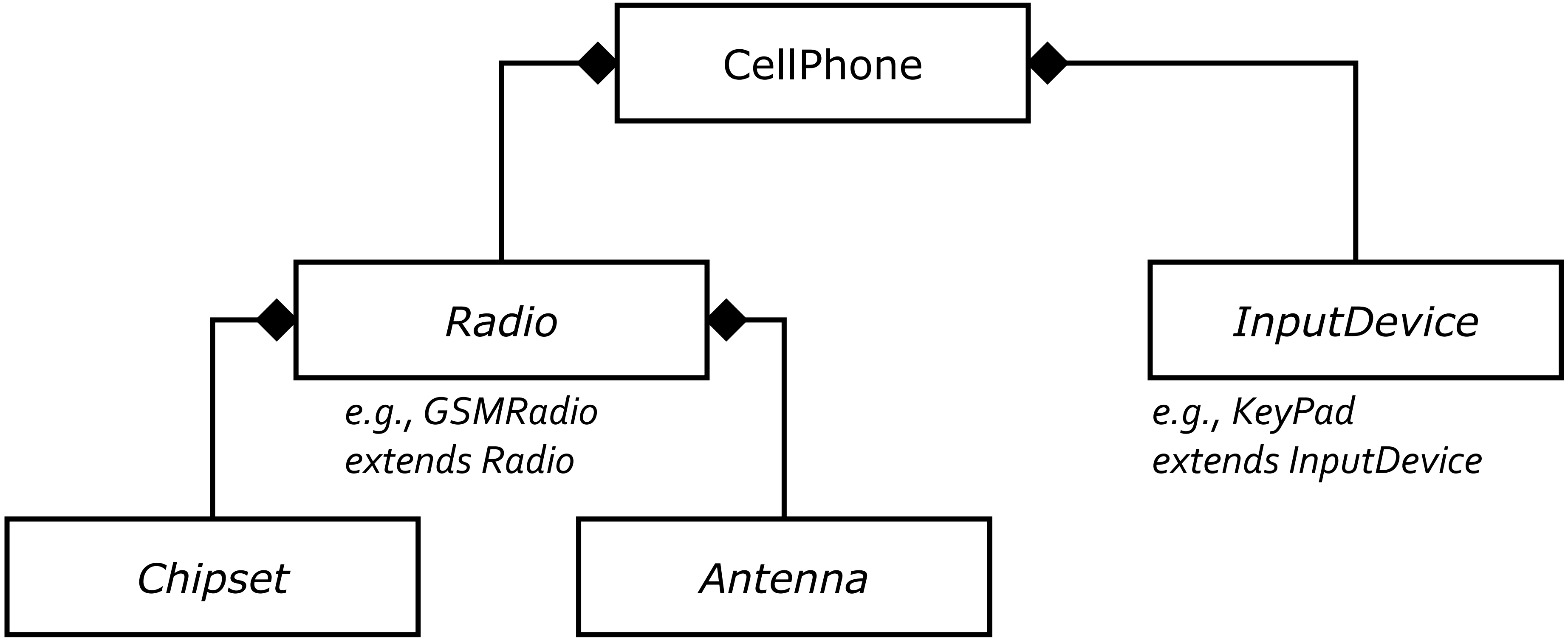
System Assembly with DI

- ```
  // in some high-level class

  CellPhone phone = new CellPhone(
      new GSMRadio(
          new QualcommGSMChipset(),
          new FractusQuadBandAntenna()
      ),
      new KeyPad()
  );
  ```

Separates out "dependency resolution" from the constituent classes
  ```

System Assembly with DI



e.g., GSMRadio extends Radio

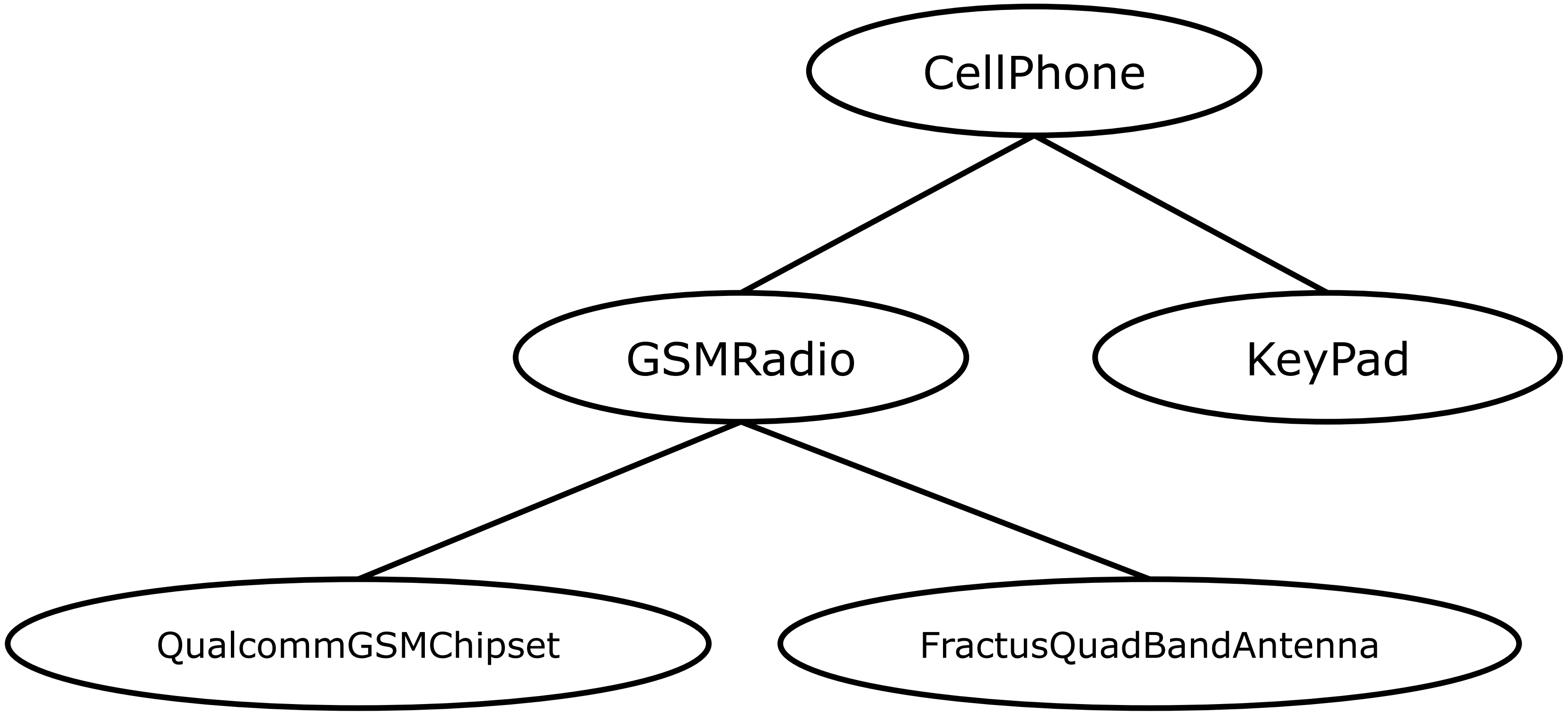
e.g., KeyPad extends InputDevice

e.g., QualcommGSMChipset extends Chipset

e.g., FractusQuadBandAntenna extends Antenna

Could have other subclasses beyond these examples

System Assembly with DI



The bottom-up assembly process instantiates the children and inserts them into the parents

Example Bad Design 2

- ```
  public class User {
      private Preferences prefs;

      public User( File prefFile ) {
          prefs = parseFile( prefFile );
          …
      }
      public void doSomething() {
          … // use prefs
      }
      …
      private Preferences parseFile( File prefFile ) {
          …
          aPrefs = new Preferences( … );
          … // setup prefs
          return aPrefs;
      }
  }
  ```
  ```

Example Bad Design 2

- Poor flexibility:
  - Changing preferences requires changing User
    - File format changes
  - Difficult to reuse User
    - Embedded preference file reading and parsing
- Poor testability:
  - Tests that deal with files are slow
  - Need test file for each preference combination

Improved Design 2

- ```
  class User {
      private Preferences prefs;

      public User( Preferences prefs ) {
          this.prefs = prefs;
          …
      }
      public void doSomething() {
          … // use prefs
      }
      …
  }
  ```

Dependency injection
  ```

Improved Design 2

- Better flexibility:
  - No change to User if file format changes
  - Preferences not limited to be made from files
- Better testability:
  - Can run fast
    - Pass in mock or fake Preferences object

“Mock Object”

-
```
public class UserTest {
    …
    public void testdoSomething() {

        // MockPreferences extends Preferences,
        // but is overridden with canned settings
        // (no test preference file needed)

        Preferences mockPrefs =
            new MockPreferences();

        User aUser = new User( mockPrefs );

        aUser.doSomething();
        …

        mockPrefs.assertNoChange();
    }
}
```

Example Bad Design 3

- Situation:
  - Many pieces of information are needed by classes throughout the system
  - But each class needs just one or a few items
  - How to provide this information to the consumers?

Example Bad Design 3

- Typical approaches:
  - Consumers get the data they need,
  - Make the data global,
  - Pass around a context object, or
  - Put the data in widely known and used classes

Example Bad Design 3

- ```
  public class Account {
      …
      public Account( User user ) {
          this.country = 
  user.getPreferences().getLocation().getCountry();
          …
      }
      …
  }
  ```
  ```

Example Bad Design 3

- Poor flexibility:
  - Method parameters do not show what the method really needs
  - Code “locks in” the structure it walks
- Poor testability:
  - Test needs to recreate this structure

Example Bad Design 3

- ```
  public void testSomethingForAccount() {
      // set up for test

      Country country = new Country( "Canada" );

      Location location = new Location();
      location.setCountry( country );

      Preferences prefs = new Preferences();
      prefs.setLocation( location );

      User user = new User( prefs );

      Account account = new Account( user );

      … // test Canadian account
  }
  ```

Test code should be simple (less likely to have defects)
  ```

Improved Design 3

- ```
  public void testSomethingForAccount() {

      Country country = new Country( "Canada" );

      // redesigned constructor
      // (requires only what is needed)
      Account account = new Account( country );

      … // test Canadian account
  }
  ```
  ```

More Information

- Books:
  - Test-Driven Development
    - K. Beck
    - Addison-Wesley, 2003
  - Testing Computer Software
    - C. Kaner, J. Falk, H. Q. Nguyen
    - Wiley, 1999
  - Lessons Learned in Software Testing
    - C. Kaner, J. Bach, B. Pettichord
    - Wiley, 2002
  - Flexible Design? Testable Design? You Don't Have to Choose!
    - R. Rufer and T. Bialik

More Information

- Links:
  - Cause of AT&T Network Failure
    - http://catless.ncl.ac.uk/Risks/9.62.html#subj2
  - The Way of Testivus
    - http://www.agitar.com/downloads/TheWayOfTestivus.pdf
  - JUnit Resources for Test-Driven Development
    - https://junit.org/junit5/