CSC 307 Lecture Notes Week 4

Introduction to Requirements Modeling

Requirements Inspection Testing

I. Materials:

I. Materials:

A. Milestones 3-4 writeup

I. **Materials:**

A. Milestones 3-4 writeup

B. Milestone 4 example

I. **Materials:**

A. Milestones 3-4 writeup

B. Milestone 4 example

C. Java as an Abstract Modeling Language

I. **Materials:**

A. Milestones 3-4 writeup

B. Milestone 4 example

C. Java as an Abstract Modeling Language

D. SOP Volume 2: Requirements Testing

II. Lab quiz Friday, 16 October

II. Lab quiz Friday, 16 October

A. Covers SVN Basics.

II. **Lab quiz Friday, 16 October**

A. Covers SVN Basics.

B. Command-line interface.

II. **Lab quiz Friday, 16 October**

A. Covers SVN Basics.

B. Command-line interface.

C. No questions on SVN clients.

III. After "Analyze" comes "Specify".

III. After "Analyze" comes "Specify".

A. Formalize functional requirements, so that:

III. **After "Analyze" comes "Specify".**

A. Formalize functional requirements, so that:

1. Requirements are complete and consistent

III. **After "Analyze" comes "Specify".**

A. Formalize functional requirements, so that:

1. Requirements are complete and consistent

2. Requirements are clear and unambiguous

III. **After "Analyze" comes "Specify".**

A. Formalize functional requirements, so that:

1. Requirements are complete and consistent

2. Requirements are clear and unambiguous

B. This is the ***modeling*** step.

IV. When to model?

IV. When to model?

A. In more traditional process, done between ***Analyze*** and ***Implement*** steps.

IV. When to model?

A. In more traditional process, done between ***Analyze*** and ***Implement*** steps.

B. In agile process, done as needed in a ***Refactor*** step.

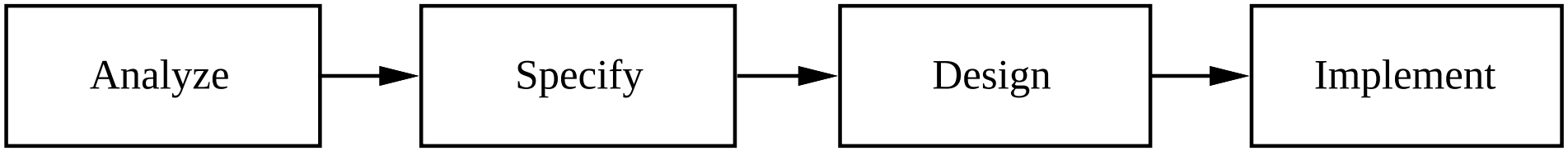
IV. When to model?

A. In more traditional process, done between ***Analyze*** and ***Implement*** steps.

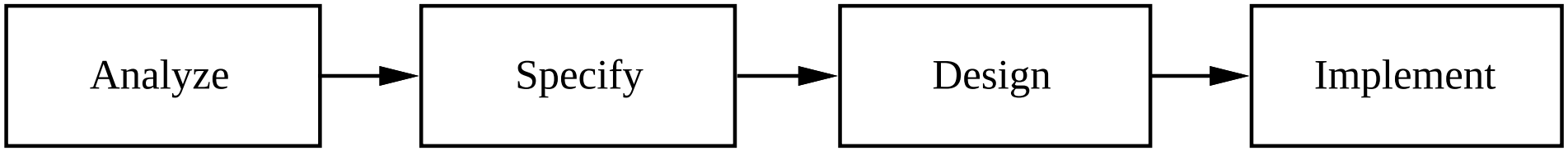
B. In agile process, done as needed in a ***Refactor*** step.

C. See Figure 1.

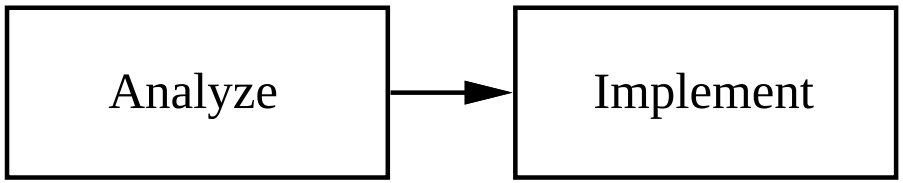
Traditional Approach:

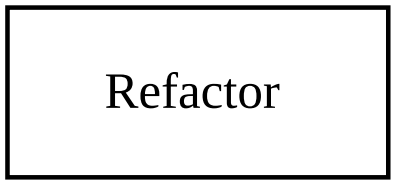


Traditional Approach:



Agile Approach:





Refactor when necessary, which entails what's done in Specciry and Design

V. Modeling Languages.

V. Modeling Languages.

A. There are a number of alternatives.

V. Modeling Languages.

A. There are a number of alternatives.

B. We'll use a subset of Java.

V. **Modeling Languages.**

A. There are a number of alternatives.

B. We'll use a subset of Java.

C. See the handout
"Java as an Abstract Modeling Language".

V. **Modeling Languages.**

A. There are a number of alternatives.

B. We'll use a subset of Java.

C. See the handout
"Java as an Abstract Modeling Language".

D. *See online notes for further details.*

VI. How formal do we get?

VI. How formal do we get?

A. We'll go all the way to formal math logic.

VI. How formal do we get?

A. We'll go all the way to formal math logic.

B. We'll do it step-by-step.

VI. **How formal do we get?**

A. We'll go all the way to formal math logic.

B. We'll do it step-by-step.

C. *See online notes for details.*

VII. Elements of the model.

VII. Elements of the model.

A. Objects

VII. Elements of the model.

A. Objects -- *classes in Java*

VII. Elements of the model.

A. Objects -- *classes in Java*

B. Operations

VII. **Elements of the model.**

A. **Objects** -- *classes in Java*

B. **Operations** -- *methods in Java*

VII. Elements of the model.

A. Objects -- *classes in Java*

B. Operations -- *methods in Java*

C. Modules

VII. **Elements of the model.**

A. **Objects** -- *classes in Java*

B. **Operations** -- *methods in Java*

C. **Modules** -- *packages in Java*

VIII. Heuristics for deriving model

VIII. Heuristics for deriving model

A. Derive from UI pictures and narrative.

VIII. **Heuristics for deriving model**

A. Derive from UI pictures and narrative.

B. Heuristics include:

VIII. **Heuristics for deriving model**

A. Derive from UI pictures and narrative.

B. Heuristics include:

1. Buttons, menu items = ***operations***.

VIII. **Heuristics for deriving model**

A. Derive from UI pictures and narrative.

B. Heuristics include:

1. Buttons, menu items = ***operations***.

2. Data-entry and output screens = ***objects***.

Heuristics, cont'd

3. Data-entry dialogs = ***input objects***.

Heuristics, cont'd

3. Data-entry dialogs = ***input objects***.

4. Output screens = ***output objects***.

Heuristics, cont'd

3. Data-entry dialogs = ***input objects***.

4. Output screens = ***output objects***.

5. Number, string, boolean, enum literals
   = ***primitive objects***.

Heuristics, cont'd

3. Data-entry dialogs = ***input objects***.

4. Output screens = ***output objects***.

5. Number, string, boolean, enum literals
   = ***primitive objects***.

6. Hierarchical structure in nested windows.

Heuristics, cont'd

C. Details of object and operation attributes derived from scenario narrative.

IX. Examples from Calendar Tool

IX. Examples from Calendar Tool

A. Apply the preceding heuristics.

IX. Examples from Calendar Tool

A. Apply the preceding heuristics.

B. Complete details in ***`specification`*** directory of Milestone 4 example.

X. Deriving scheduling operations

X. Deriving scheduling operations

A. `Schedule` command menu:

X. Deriving scheduling operations

A. `Schedule` command menu:

```
Appointment ...
Meeting ...
Task ...
Event ...
```

Deriving ops, cont'd

B. Applying first heuristic
(buttons, menus = operations):

Deriving ops, cont'd

B. Applying first heuristic
(buttons, menus = operations):

```
void scheduleAppointment();
void scheduleMeeting();
void scheduleTask();
void scheduleEvent();
```

Deriving ops, cont'd

C. Yet to identify these aspects:

Deriving ops, cont'd

C. Yet to identify these aspects:

1. What class they go in.

Deriving ops, cont'd

C. Yet to identify these aspects:

1. What class they go in.

2. What parameter(s) they take.

Deriving ops, cont'd

C. Yet to identify these aspects:

1. What class they go in.

2. What parameter(s) they take.

3. What return value they produce.

Deriving ops, cont'd

D. Operation names are verbs or verb phrases.

Deriving ops, cont'd

D. Operation names are verbs or verb phrases.

1. Use suitably modified UI elements.

Deriving ops, cont'd

D. Operation names are verbs or verb phrases.

1. Use suitably modified UI elements.

2. E.g., method name =

menu name + ***menu item name***

with Java syntax and case conventions.

XI. Deriving scheduling objects.

XI. **Deriving scheduling objects.**

A. Use second heuristic--

data-entry screens = objects

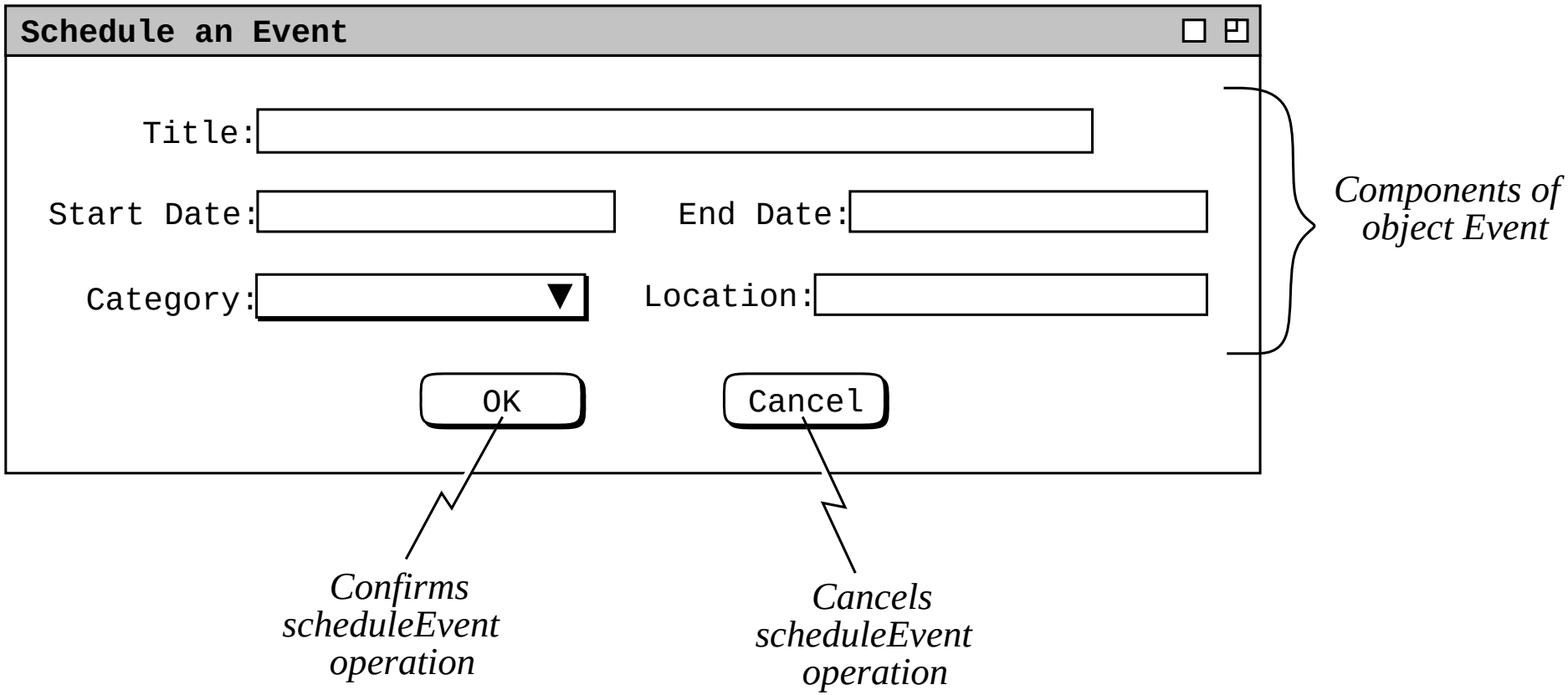
XI. **Deriving scheduling objects.**

A. Use second heuristic--

data-entry screens = objects

B. Applying to an `Event` object:

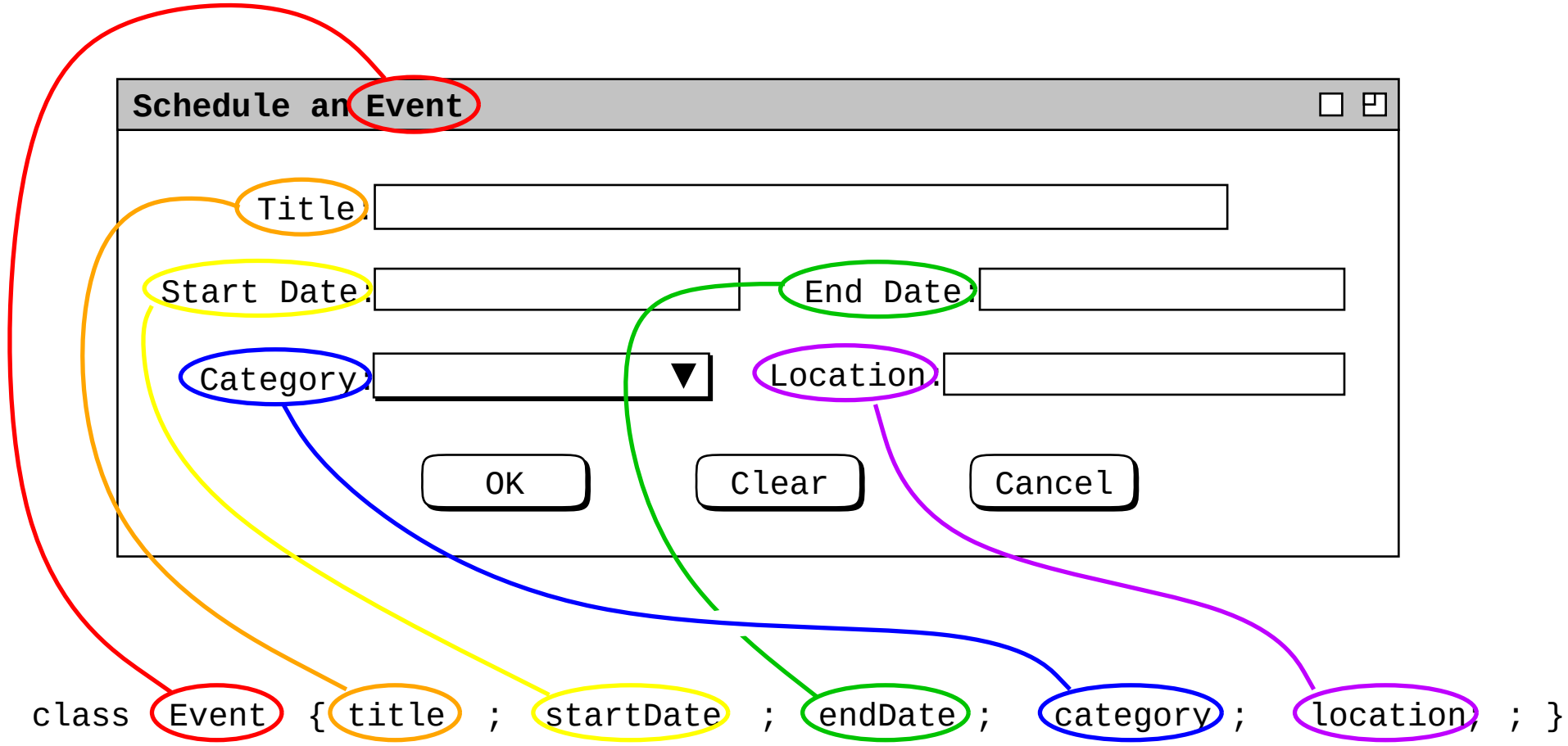
This picture



derives to this object

```
class Event {
    String title;
    Date startDate
    Date endDate
    Category category
    String location;
}
```

Deriving objs, cont'd



Deriving objs, cont'd

Remember, the heuristic

data-entry screens = objects

is a ***rule of thumb***, not an exact rule.

Deriving objs, cont'd

C. So far we've done some initial analysis:

Deriving objs, cont'd

C. So far we've done some initial analysis:

1. The title and location are primitive strings

Deriving objs, cont'd

C. So far we've done some initial analysis:

1. The title and location are primitive strings

2. Other types not yet fully defined

Deriving objs, cont'd

C. So far we've done some initial analysis:

1. The title and location are primitive strings

2. Other types not yet fully defined

```
class Date { /* ... */ }
class Category { /* ... */ }
```

XII. Object derivation details.

XII. Object derivation details.

A. Java type derived from UI elements.

XII. **Object derivation details.**

A. Java type derived from UI elements.

B. Table 1 summarizes.

Java Type Common Interface Form

Java Type	Common Interface Form
int	string editor, slider, dial

Java Type	Common Interface Form
int	string editor, slider, dial
double	same as integer

Java Type	Common Interface Form
int	string editor, slider, dial
double	same as integer
String	string editor, combo box

Java Type	Common Interface Form
int	string editor, slider, dial
double	same as integer
String	string editor, combo box
boolean	string editor, on/off button

Java Type	Common Interface Form
int	string editor, slider, dial
double	same as integer
String	string editor, combo box
boolean	string editor, on/off button
data field	box containing other types

Java Type	Common Interface Form
int	string editor, slider, dial
double	same as integer
String	string editor, combo box
boolean	string editor, on/off button
data field	box containing other types
enum	radio buttons; fixed-length list

Java Type	Common Interface Form
int	string editor, slider, dial
double	same as integer
String	string editor, combo box
boolean	string editor, on/off button
data field	box containing other types
enum	radio buttons; fixed-length list
Collection	variable-length list

Java Type	Common Interface Form
int	string editor, slider, dial
double	same as integer
String	string editor, combo box
boolean	string editor, on/off button
data field	box containing other types
enum	radio buttons; fixed-length list
Collection	variable-length list
Method	push button or menu item

XIII. Refining object definitions.

XIII. **Refining object definitions.**

A. From narrative for event dialog, `Title` and `Location` are free-form strings.

XIII. **Refining object definitions.**

A. From narrative for event dialog, `Title` and `Location` are free-form strings.

B. `String` type models free-form strings

Refining objs, cont'd

C. Details of date formats not yet worked out.

Refining objs, cont'd

C. Details of date formats not yet worked out.

1. Given this, leave def of `Date` to later.

Refining objs, cont'd

C. Details of date formats not yet worked out.

1. Given this, leave def of `Date` to later.

2. I.e.,

```
class Date { /* ... */ }
```

Refining objs, cont'd

D. UI displays `Category` as list of selections.

Refining objs, cont'd

D. UI displays `Category` as list of selections.

1. This might lead to model `Category` as just a string, represented the selected category name.

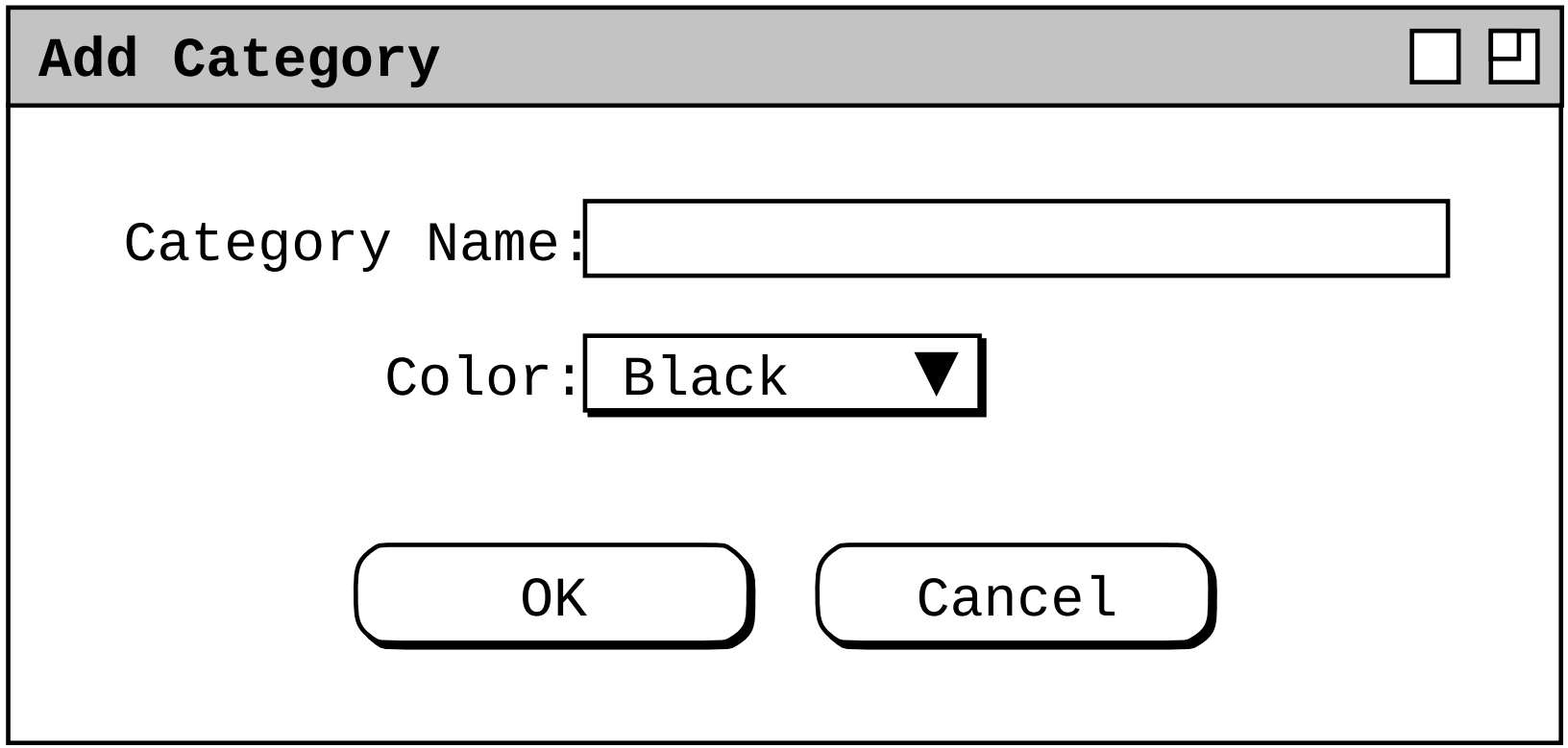
Refining objs, cont'd

D. UI displays `Category` as list of selections.

1. This might lead to model `Category` as just a string, represented the selected category name.

2. More careful analysis from this picture:

Refining objs, cont'd



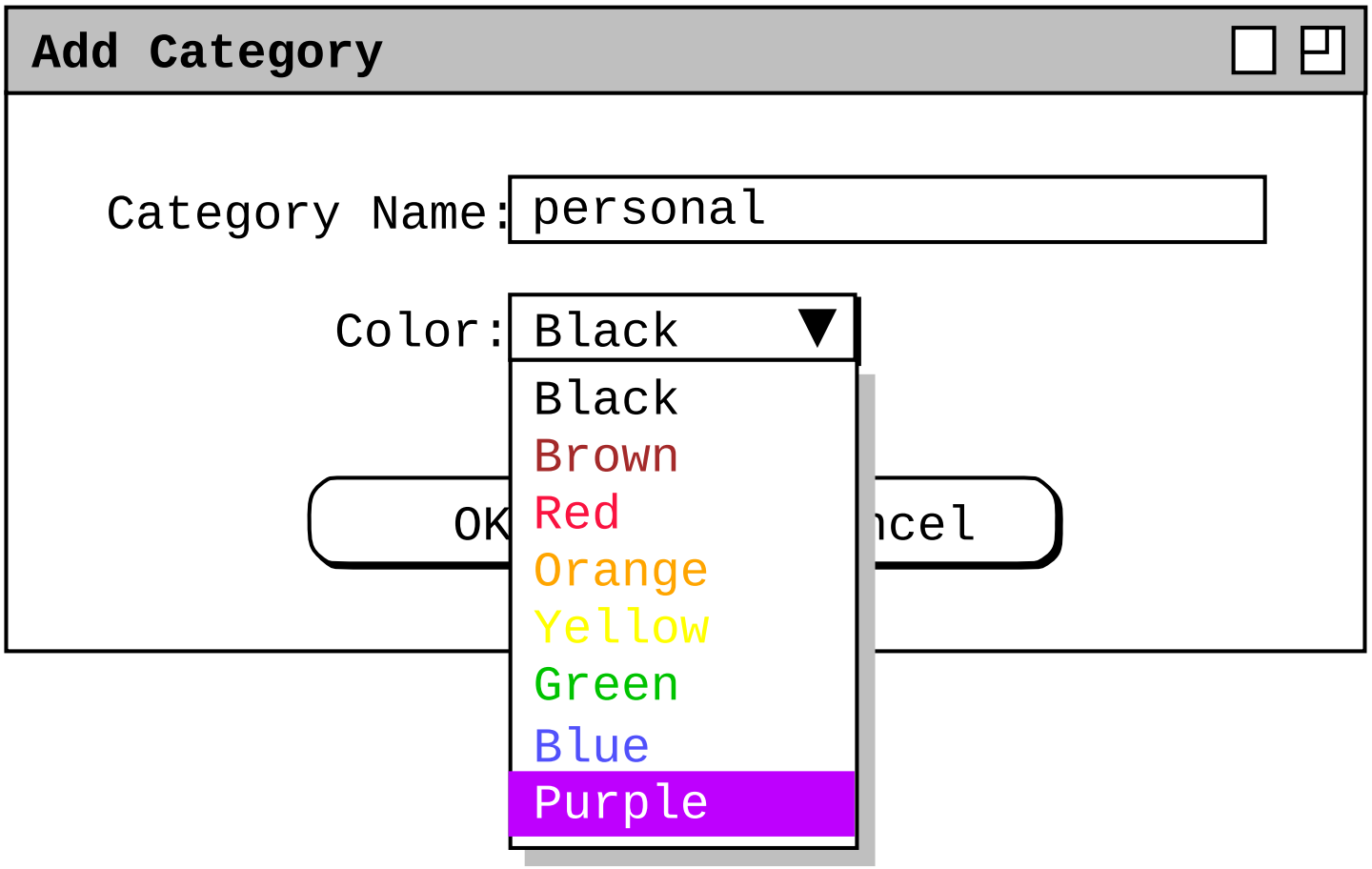
Refining objs, cont'd

3. Hence, more accurate def is:

```
class Category {
    String name;
    Color color;
}
```

Refining objs, cont'd

4. Subsequent screen shows `Color` as



Refining objs, cont'd

5. Hence, model as follows:

```
enum Color {
    Black, Brown, Red, Orange,
    Yellow or Green, Blue, Purple;
}
```

Refining objs, cont'd

E. Preceding analysis is typical.

Refining objs, cont'd

E. Preceding analysis is typical.

1. Derive initial obj defs from UI pictures.

Refining objs, cont'd

E. Preceding analysis is typical.

1. Derive initial obj defs from UI pictures.

2. Refine based on narrative.

Refining objs, cont'd

E. Preceding analysis is typical.

1. Derive initial obj defs from UI pictures.

2. Refine based on narrative.

3. Continue until all objects defined in terms of primitives.

XIV. Refining operation definitions.

XIV. **Refining operation definitions.**

A. The key step is determining class.

XIV. **Refining operation definitions.**

A. The key step is determining class.

B. Clarifies what object is operated on.

XIV. **Refining operation definitions.**

A. The key step is determining class.

B. Clarifies what object is operated on.

C. Analysis determines there's a *Calendar* object.

Refining ops, cont'd

D. Hence,

```
class Calendar {
    void scheduleAppointment();
    void scheduleMeeting();
    void scheduleTask();
    void scheduleEvent();
}
```

Refining ops, cont'd

E. Using heuristic 3 (data-entries are inputs):

```
class Calendar {
   void scheduleAppointment(Appointment);
   void scheduleMeeting(Meeting);
   void scheduleTask(Task);
   void scheduleEvent(Event);
}
```

Refining ops, cont'd

F. We want all of abstract models to compile.

Refining ops, cont'd

F. We want all of abstract models to compile.

1. Abstract means leaving out all code.

Refining ops, cont'd

F. We want all of abstract models to compile.

1. Abstract means leaving out all code.

2. Declare all of the methods `abstract`

Refining ops, cont'd

G. Here's a compilable def:

```
abstract class Calendar {
   abstract void
      scheduleAppointment(Appointment);
   abstract void
      scheduleMeeting(Meeting);
   abstract void
      scheduleTask(Task);
   abstract void
      scheduleEvent(Event);
}
```

XV. Identifying collection objects.

XV. Identifying collection objects.

A. Key aspect of data modeling.

XV. Identifying collection objects.

A. Key aspect of data modeling.

B. Collections contain zero or more objects.

XV. Identifying collection objects.

A. Key aspect of data modeling.

B. Collections contain zero or more objects.

C. Identified by descriptive language, known pattern of operations.

Identifying collections, cont'd

D. E.g., end of Section 2.2:

"After scheduling and confirming an appoint-ment, the appointment data are entered in an online working copy of the user's calendar."

Identifying collections, cont'd

E. Use Java `Collection` to model:

```
abstract class Calendar {
  abstract void scheduleAppointment(Appointment);
  abstract void scheduleMeeting(Meeting);
  abstract void scheduleTask(Task);
  abstract void scheduleEvent(Event);

  Collection<Appointment> data;

}
```

Identifying collections, cont'd

F. Over-simplification, since calendars can contain meetings, tasks and events, as well.

Identifying collections, cont'd

F. Over-simplification, since calendars can contain meetings, tasks and events, as well.

G. We'll refine soon, like this

```
Collection<ScheduledItem> data;
```

Identifying collections, cont'd

H. Also identify collections by four ops:

Identifying collections, cont'd

H. Also identify collections by four ops:

1. *Additive*, *destructive*, *modifying*, *selective*.

Identifying collections, cont'd

H. Also identify collections by four ops:

1. *Additive*, *destructive*, *modifying*, *selective*.

2. I.e., ops to add, delete, edit, and find.

Identifying collections, cont'd

H. Also identify collections by four ops:

1. *Additive*, *destructive*, *modifying*, *selective*.

2. I.e., ops to add, delete, edit, and find.

3. Coming up, we'll consider this to be a formal specification pattern.

XVI. Deriving a monthly view object.

XVI. **Deriving a monthly view object.**

A. Many objects will be derived from calendar `View` commands.

XVI. **Deriving a monthly view object.**

A. Many objects will be derived from calendar `View` commands.

B. As initial example, consider in a month view:

Monthly Agenda

Today **September 2014**

Sun	Mon	Tue	Wed	Thu	Fri	Sat
		1 8-9 AM Ra 11 AM-2 P	2	3 8-9 AM Ra	4	5
6	7 Labor Day	8 8-9 AM Ra 11 AM-2 P	9	10 8-9 AM Ra	11	12
13	14	15 8-9 AM Ra 11 AM-2 P	16 1. Send c	17 8-9 AM Ra	18	19
20	21 1. Colloq 9-10 AM O 10-11 AM 11 AM-12	22 Autumnal 8-9 AM Ra 9 AM-5 PM	23 8-9 AM St 8-9:30 AM 9-10 AM O 10-11 AM	24 Jim's Bir 1. Prepar 2. Buy so 8-9 AM Ra	25 8-9:30 AM 9-10 AM O 10-11 AM 11 AM-12	26
28	27 1. Colloq 9-10 AM O 10-11 AM 11 AM-12	29 8-9 AM Ra 9 AM-5 PM	30 9-10 AM O 10-11 AM 11 AM-12 2:30-4:30			

C. From this we can derive:

```
import java.util.Collection;
```

```
import java.util.Collection;

/**
 * A MonthlyAgenda contains a small daily view for each
 * day of the month, organized in the fashion typical
 * in paper calendars.
 */
class MonthlyAgenda {
    FullMonthName name;
    DayOfTheWeek firstDay;
    int numberOfDays;
    Collection<SmallDayView> items;
}
```

```
class FullMonthName {
    String month;
    int year;
}
```

```
class FullMonthName {
    String month;
    int year;
}

enum DayOfTheWeek { Sun, Mon, Tue, Wed, Thu, Fri, Sat }
```

```
class FullMonthName {
    String month;
    int year;
}

enum DayOfTheWeek { Sun, Mon, Tue, Wed, Thu, Fri, Sat }
/**
 * A SmallDayView has the number of the date and a list
 * of zero or more short item descriptions.
 */

class SmallDayView {
    int DateNumber;
    Collection<BriefItemDescription> items;
}
```

```
class BriefItemDescription {
    String title;
    Time startTime;
    Duration duration;
    Category category;
}
```

```
class BriefItemDescription {
    String title;
    Time startTime;
    Duration duration;
    Category category;
}

class Time { /* ... */ }
class Duration { /* ... */ }
class Category { /* ... */ }
```

XVII. Observations on requirements modeling.

XVII. **Observations on requirements modeling.**

A. Can derive in different ways.

XVII. **Observations on requirements modeling.**

A. Can derive in different ways.

1. E.g., should the Calendar of scheduled items or collection of years?

XVII. **Observations on requirements modeling.**

A. Can derive in different ways.

1. E.g., should the Calendar of scheduled items or collection of years?

2. Should dates be modeled as simple strings or a composite class?

XVII. **Observations on requirements modeling.**

A. Can derive in different ways.

1. E.g., should the Calendar of scheduled items or collection of years?

2. Should dates be modeled as simple strings or a composite class?

3. Which of these is "correct", "most accurate"?

Observations, cont'd

B. Answer -- model *as perceived by the end user*.

Observations, cont'd

B. Answer -- model *as perceived by the end user*.

1. Helps achieve model correctness, accuracy.

Observations, cont'd

B. Answer -- model *as perceived by the end user*.

1. Helps achieve model correctness, accuracy.

2. Don't model for computational efficiency.

Observations, cont'd

B. Answer -- model *as perceived by the end user*.

1. Helps achieve model correctness, accuracy.
2. Don't model for computational efficiency.
3. We'll discuss further in upcoming lectures.

XVIII. Some Milestone 4 Details

XVIII. Some Milestone 4 Details

A. Modeling for Milestone 4.

XVIII. **Some Milestone 4 Details**

A. Modeling for Milestone 4.

1. See M4 example for guide of how much.

XVIII. Some Milestone 4 Details

A. Modeling for Milestone 4.

1. See M4 example for guide of how much.

   a. Each team member must commit at least four model classes.

XVIII. Some Milestone 4 Details

A. Modeling for Milestone 4.

1. See M4 example for guide of how much.

   a. Each team member must commit at least four model classes.

   b. Classes can be in one or more `.java` files.

XVIII. Some Milestone 4 Details

A. Modeling for Milestone 4.

1. See M4 example for guide of how much.

   a. Each team member must commit at least four model classes.

   b. Classes can be in one or more `.java` files.

   c. Team coordination needed for shared objects and package structure.

Milestone 4, cont'd

2. Create package sub-directories under `specification` directory.

Milestone 4, cont'd

2. Create package sub-directories under `specification` directory.

3. Put `.java` files in appropriate package dirs.

Milestone 4, cont'd

2. Create package sub-directories under `specification` directory.

3. Put `.java` files in appropriate package dirs.

4. ***The files must compile with `javac`.***

Milestone 4, cont'd

2. Create package sub-directories under `specification` directory.

3. Put `.java` files in appropriate package dirs.

4. ***The files must compile with `javac`.***

5. ***Documentation must be generated with `javadoc`.***

Milestone 4, cont'd

B. Requirements inspection testing.

Milestone 4, cont'd

B. Requirements inspection testing.

1. Review procedure in the SOP Vol. 2.

Milestone 4, cont'd

B. Requirements inspection testing.

1. Review procedure in the SOP Vol. 2.

2. Decide as team the time of pre-testing check-in, so librarian can release by 11:59PM.

XIX. Guidelines for modularizing a model.

XIX. Guidelines for modularizing a model.

A. To *modularize* means subdivide into independent units.

XIX. **Guidelines for modularizing a model.**

A. To *modularize* means subdivide into independent units.

B. Dictionary definition of a *module* --

"... an independent unit that can be used to construct a more complex structure".

Modularization, cont'd

C. In Java, modules defined as `packages`.

Modularization, cont'd

C. In Java, modules defined as `packages`.

D. Good heuristic uses large-grain UI structure.

Modularization, cont'd

C. In Java, modules defined as `packages`.

D. Good heuristic uses large-grain UI structure.

1. Each menu in a menu-based UI is a module.

Modularization, cont'd

C. In Java, modules defined as `packages`.

D. Good heuristic uses large-grain UI structure.

1. Each menu in a menu-based UI is a module.

2. Similarly, top-level UI toolbars can be considered modules.

Modularization, cont'd

E. Given these heuristics, packaging structure of Calendar Tool can look like this:

Modularization, cont'd

E. Given these heuristics, packaging structure of Calendar Tool can look like this:

```
package file;
package edit;
package schedule;
package view;
package admin;
package options;
```

Modularization, cont'd

F. Within each package are appropriate classes.

Modularization, cont'd

F. Within each package are appropriate classes.

1. For Cal Tool focus is `schedule` and `view`.

Modularization, cont'd

F. Within each package are appropriate classes.

1. For Cal Tool focus is `schedule` and `view`.

2. Packaging structure is easy to view in `javadoc` form.

Modularization, cont'd

F. Within each package are appropriate classes.

1. For Cal Tool focus is `schedule` and `view`.

2. Packaging structure is easy to view in `javadoc` form.

3. Each package dir has `package.html`.

Additional material to read in this weeks lecture notes:

XX. Summary of core steps of modeling

Additional material to read in this weeks lecture notes:

XX. **Summary of core steps of modeling**

XXI. **Specific modeling guidelines.**

Additional material to read in this weeks lecture notes:

XX. **Summary of core steps of modeling**

XXI. **Specific modeling guidelines.**

XXII. **Details of object derivation.**

Additional material to read in this weeks lecture notes:

XX. **Summary of core steps of modeling**

XXI. **Specific modeling guidelines.**

XXII. **Details of object derivation.**

XXIII. **Details of operation derivation.**

Additional material to read in this weeks lecture notes:

XXIV. A detailed cal tool scheduling example

Additional material to read in this weeks lecture notes:

XXIV. **A detailed cal tool scheduling example**

XXV. **A detailed cal tool viewing example**

Additional material to read in this weeks lecture notes:

XXIV. **A detailed cal tool scheduling example**

XXV. **A detailed cal tool viewing example**

XXVI. **Summary Observations about Modeling**

Additional material to read in this weeks lecture notes:

XXIV. **A detailed cal tool scheduling example**

XXV. **A detailed cal tool viewing example**

XXVI. **Summary Observations about Modeling**

. . . and some additional topics

Now Some Topics from the Handout

Overview of Using Java as an Abstract Modeling and Specification Language

1. Tabular and Graphical Modeling Notations

1. Tabular and Graphical Modeling Notations

- Same model in tabular or graphical form.

1. Tabular and Graphical Modeling Notations

- Same model in tabular or graphical form.
- Tabular form called a "*data dictionary*".

1. Tabular and Graphical Modeling Notations

- Same model in tabular or graphical form.
- Tabular form called a "*data dictionary*".
- Graphical notation based on UML and others.

1.1. Data Dictionaries

1.1. Data Dictionaries

- A well-used notation.

1.1. Data Dictionaries

- A well-used notation.
- Shows objects, components, and descriptions.

1.1. Data Dictionaries

- A well-used notation.
- Shows objects, components, and descriptions.
- E.g.,

Object Name	Components	Description
Appointment	. . .	. . .
Calendar	. . .	. . .
ScheduledItem	. . .	. . .
Meeting	. . .	. . .
StaffMeeting	. . .	. . .
UserMeeting	. . .	. . .

Data Dictionaries, cont'd

- We'll use Javadoc for data dictionaries.

Data Dictionaries, cont'd

- We'll use Javadoc for data dictionaries.
- It's programmer-oriented, but OK.

1.2. Class Diagrams

1.2. Class Diagrams

- Depict object composition and inheritance.

1.2. Class Diagrams

- Depict object composition and inheritance.
- Show operations associated with objects.

1.2. Class Diagrams

- Depict object composition and inheritance.
- Show operations associated with objects.
- Elements are:

Class Diagrams, cont'd

1. three-part object boxes

Class Diagrams, cont'd

1. three-part object boxes
2. one-part object boxes

Class Diagrams, cont'd

1. three-part object boxes
2. one-part object boxes
3. ovals for operations

Class Diagrams, cont'd

4. connecting edges:

Class Diagrams, cont'd

4. connecting edges:

    a. hollow triangle for inheritance

Class Diagrams, cont'd

4. connecting edges:

    a. hollow triangle for inheritance

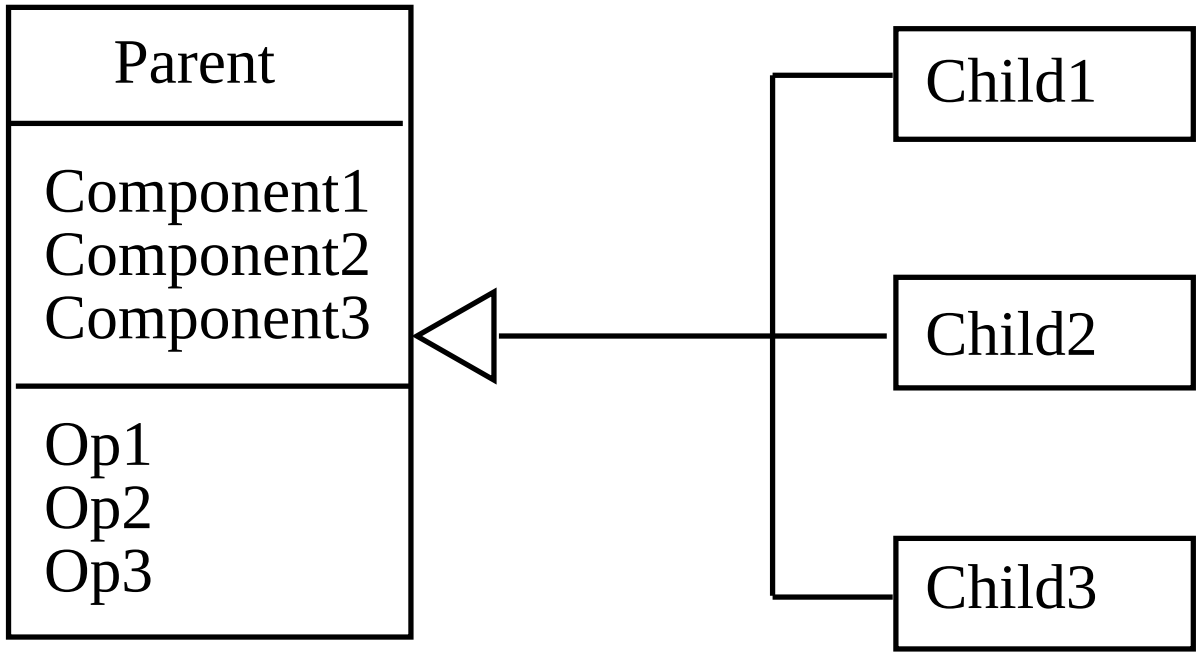
    b. hollow diamond for composition

Class Diagrams, cont'd

4. connecting edges:
    a. hollow triangle for inheritance
    b. hollow diamond for composition
    c. '*' or number for repetition

Class Diagrams, cont'd

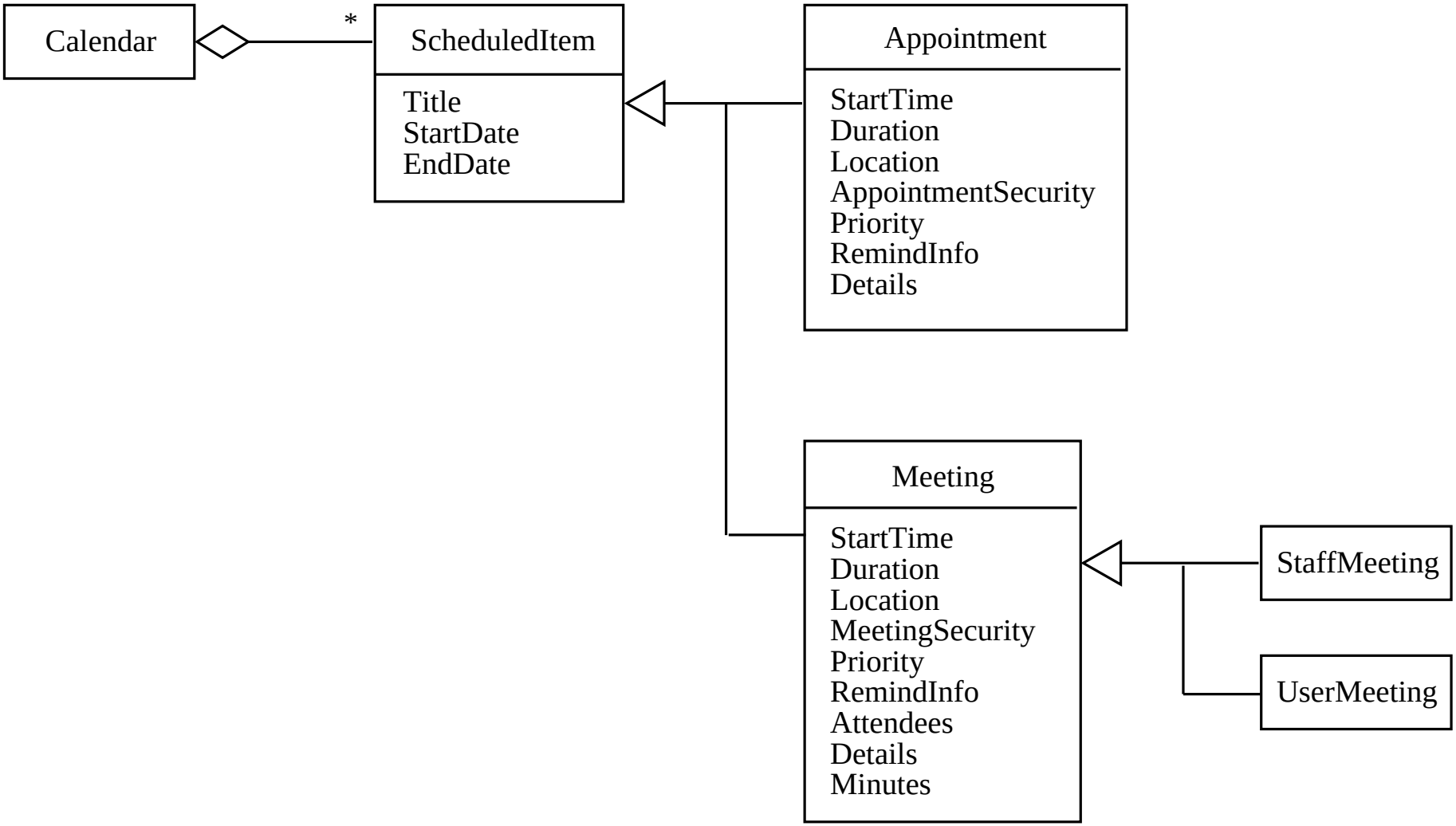
Sample Class Diagram



Class Diagrams, cont'd

```
class Parent {
    Comp1 c1;
    Comp2 c2;
    Comp3 c3;
    void Op1();
    void Op2();
    void Op3();
}
class Child1 extends Parent {}
class Child2 extends Parent {}
class Child3 extends Parent {}
```

Calendar Tool example:



1.3. Dataflow Diagrams

1.3. Dataflow Diagrams

- Depict flow of objects between operations.

1.3. Dataflow Diagrams

- Depict flow of objects between operations.
- Elements:

1.3. Dataflow Diagrams

- Depict flow of objects between operations.
- Elements:
  1. circular/oval nodes for operations

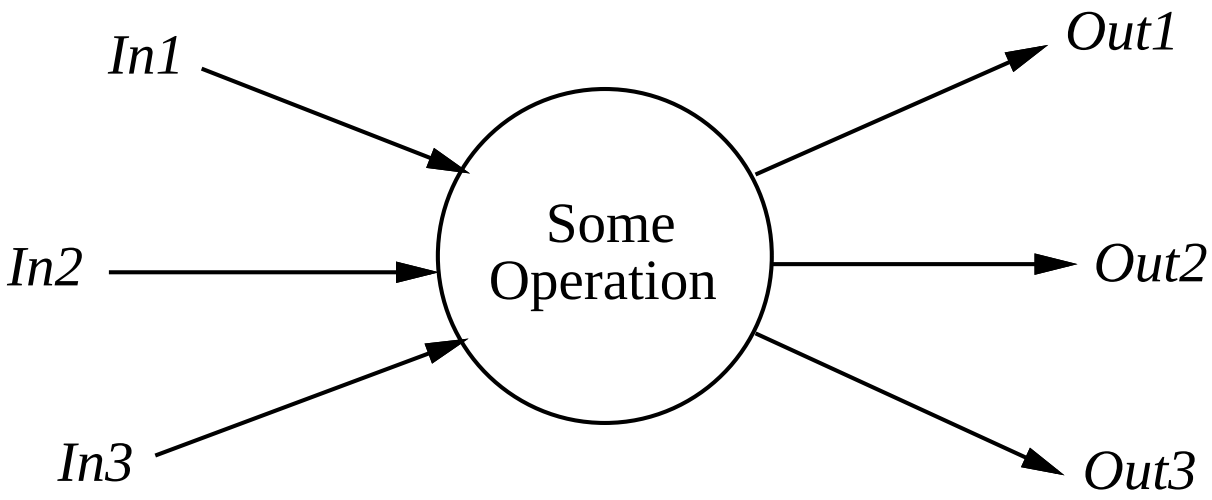
1.3. Dataflow Diagrams

- Depict flow of objects between operations.
- Elements:
  1. circular/oval nodes for operations
  2. directed edges, for i/o

1.3. Dataflow Diagrams

- Depict flow of objects between operations.
- Elements:
  1. circular/oval nodes for operations
  2. directed edges, for i/o
  3. graph levels for operation hierarchy

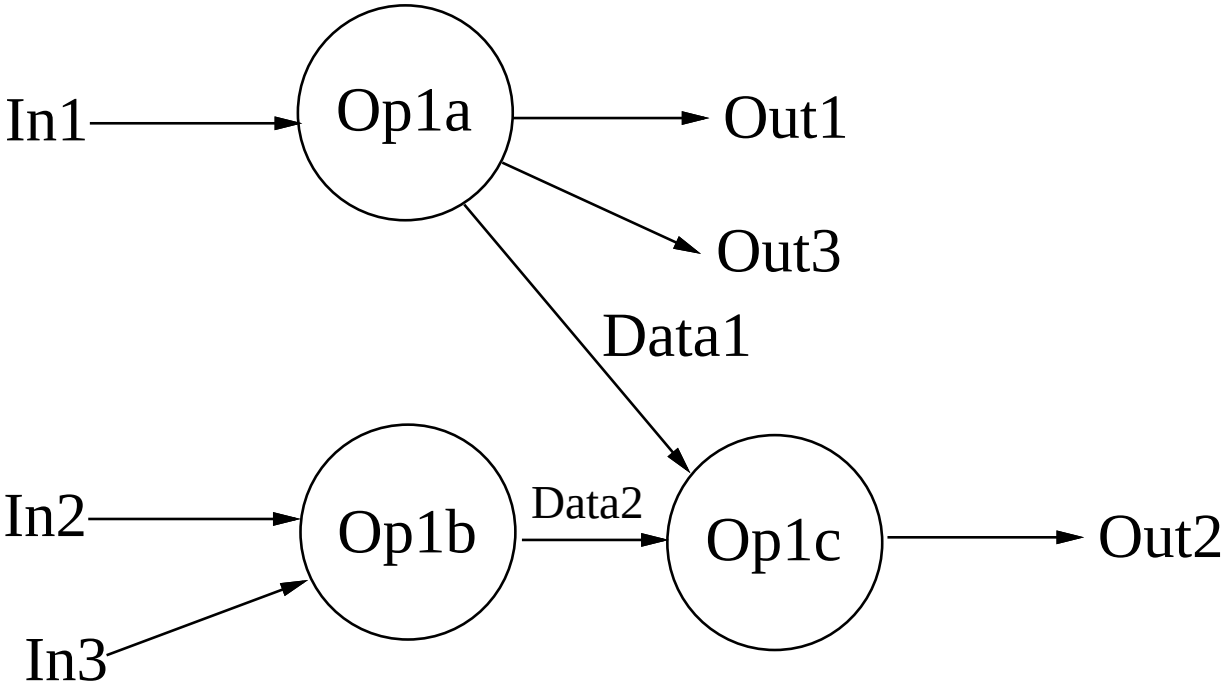
Top-Level Dataflow:



Corresponding Java:

```
class X {
    Outputs someOperation(In1, In2, In3);
}
class Outputs {
    Out1 o1;  Out2 o2;  Out3 o3;
}
```

Level 1 Expansion:



Corresponding Java:

```
Outputs1_2_3 Op1(In1, In2, In3);
Outputs1_3_D Op1a(In1);
Data2 Op1b(In2, In3);
Out2 Op1c(Data1, Data2);
```

1.4. Package Diagrams

1.4. Package Diagrams

- For large-grain modeling.

1.4. Package Diagrams

- For large-grain modeling.
- Depict relationship between modules.

1.4. Package Diagrams

- For large-grain modeling.
- Depict relationship between modules.
- Elements:

Package Diagrams, cont'd

1. folder-shaped rectangles for modules

Package Diagrams, cont'd

1. folder-shaped rectangles for modules
2. interconnection lines

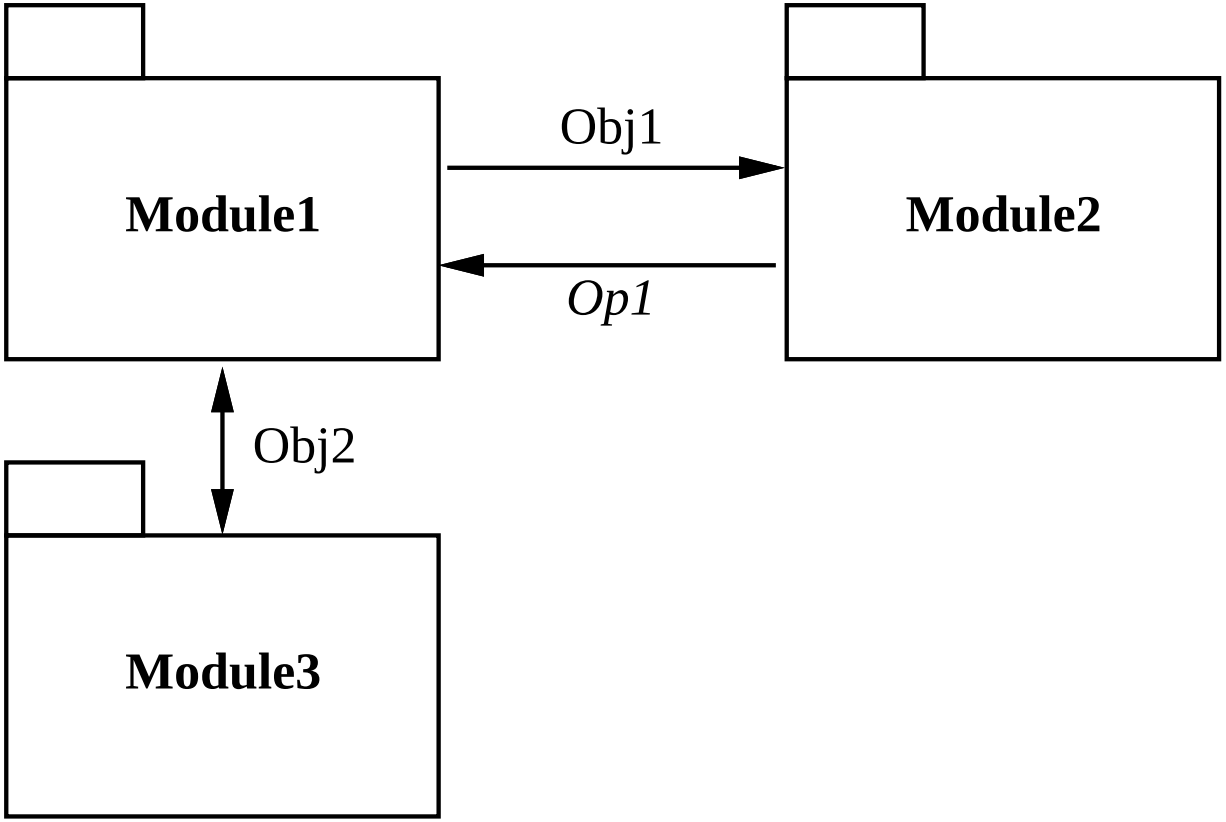
Package Diagrams, cont'd

1. folder-shaped rectangles for modules
2. interconnection lines
    a. data communication

Package Diagrams, cont'd

1. folder-shaped rectangles for modules
2. interconnection lines
    a. data communication
    b. functional communication

Package Diagrams, cont'd



References

References -- *Beginnings of Software Modeling*

References -- *Beginnings of Software Modeling*

[Ross 77] "Structured Analysis (SA)"

featuring the SADT diagram

References -- *Beginnings of Software Modeling*

[Ross 77] "Structured Analysis (SA)"

featuring the SADT diagram

[Teichroew 77] "PSL/PSA"

the Problem Statement Language

References -- *Beginnings of Software Modeling*

[Ross 77] "Structured Analysis (SA)"

featuring the SADT diagram

[Teichroew 77] "PSL/PSA"

the Problem Statement Language

[Greenspan 82] "Capturing World Knowledge"

adds inheritance to SADT

References -- *Getting Seriously Formal*

[Guttag 85] "Larch Family of Spec Languages"

major influence on JML

References -- *Getting Seriously Formal*

[Guttag 85] "Larch Family of Spec Languages"

major influence on JML

[Goguen 88] "Introducing OBJ3"

a different, mind-altering approach

References -- *The World Takes Notice*

[Rumbaugh 91] "Object-Oriented Modeling"

featuring the OMT diagram

References -- *The World Takes Notice*

[Rumbaugh 91] "Object-Oriented Modeling"

featuring the OMT diagram

[Booch, et al. 99] "UML Ref Manual"

featuring the "Boombaugh" diagram

References -- *The World Takes Notice*

[Rumbaugh 91] "Object-Oriented Modeling"

featuring the OMT diagram

[Booch, et al. 99] "UML Ref Manual"

featuring the "Boombaugh" diagram

[OMG 05] "UML 2.0 Ref Manual"

the OMG takes over

XXVII. Testing in the SE process.

XXVII. Testing in the SE process.

A. In a bad process, testing is the very last step.

XXVII. **Testing in the SE process.**

A. In a bad process, testing is the very last step.

1. Program code only is formally tested.

XXVII. **Testing in the SE process.**

A. In a bad process, testing is the very last step.

1. Program code only is formally tested.

2. Code testing is important, but should not be the only testing.

XXVII. **Testing in the SE process.**

A. In a bad process, testing is the very last step.

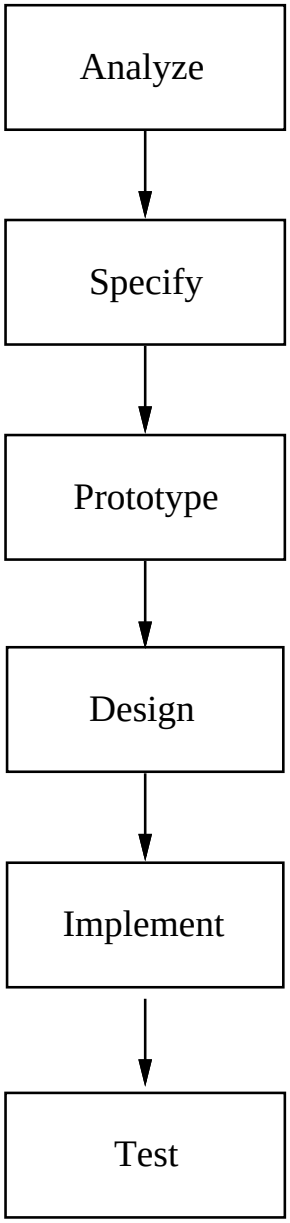
1. Program code only is formally tested.

2. Code testing is important, but should not be the only testing.

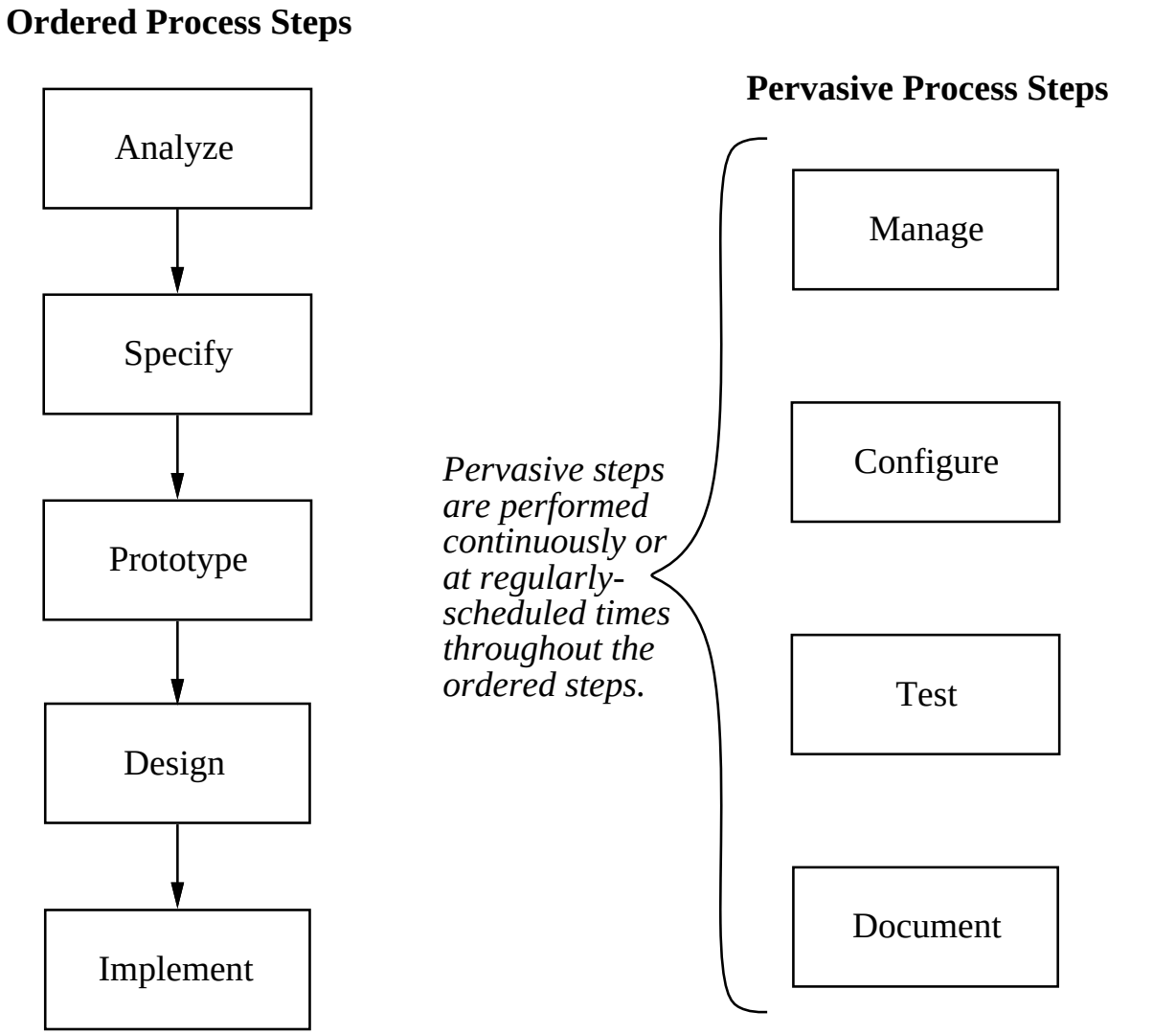
3. All artifacts can be tested formally -- requirements, specification, and design.

Testing in the SE process, cont'd

B. Figure 1 compares the position of testing.



a. *Traditional process, with testing at the end.*



b. Process with testing as a pervasive step.

Testing in the SE process, cont'd

1. Pervasive steps run continuously or at regularly-scheduled intervals.

Testing in the SE process, cont'd

1. Pervasive steps run continuously or at regularly-scheduled intervals.

2. Other pervasive steps are management, configuration, documentation.

Testing in the SE process, cont'd

C. Three types of testing.

Testing in the SE process, cont'd

C. Three types of testing.

1. *Inspection testing* entails systematic human inspection of all artifacts.

Testing in the SE process, cont'd

C. Three types of testing.

1. *Inspection testing* entails systematic human inspection of all artifacts.

2. *Functional testing* is performed by programmers on executable code.

Testing in the SE process, cont'd

C. Three types of testing.

1. *Inspection testing* entails systematic human inspection of all artifacts.

2. *Functional testing* is performed by programmers on executable code.

3. *Acceptance testing* is performed by end users on released product.

XXVIII. Testing the requirements.

XXVIII. **Testing the requirements.**

A. Customer reviews

XXVIII. **Testing the requirements.**

A. Customer reviews

1. Same as for any kind of product. A means to "debug" requirements spec.

XXVIII. **Testing the requirements.**

A. Customer reviews

1. Same as for any kind of product.

2. Namely, assure we're on track and meeting customer needs.

XXVIII. **Testing the requirements.**

A. Customer reviews

1. Same as for any kind of product.

2. Namely, assure we’re on track and meeting customer needs.

3. A means to "debug" requirements spec.

Requirements testing, cont'd

B. Formal inspection testing

Requirements testing, cont'd

B. Formal inspection testing

1. Starting week 4, requirements are inspected by *inspection test engineer*.

Requirements testing, cont'd

B. Formal inspection testing

1. Starting week 4, requirements are inspected by *inspection test engineer*.

2. Each team member reviews another member's requirements.

Requirements testing, cont'd

B. Formal inspection testing

1. Starting week 4, requirements are inspected by *inspection test engineer*.

2. Each team member reviews another member's requirements.

3. Details in handout, to be discussed next week.

Requirements testing, cont'd

C. Model building.

Requirements testing, cont'd

C. Model building.

1. Common practice among engineers.

Requirements testing, cont'd

C. Model building.

1. Common practice among engineers.

2. For 307, model building is next step of software process.

CSC 307 Standard Operating Procedures,

Volume 2: Requirements Testing

Introduction

Introduction

- Purpose is to ensure quality.

Introduction

- Purpose is to ensure quality.
- Done by careful human inspection.

Introduction

- Purpose is to ensure quality.
- Done by careful human inspection.
- Each group appoints inspectors.

Introduction

- Purpose is to ensure quality.
- Done by careful human inspection.
- Each group appoints inspectors.
- Duties entered in `administration/inspection-roster.html`.

Introduction

- Purpose is to ensure quality.
- Done by careful human inspection.
- Each group appoints inspectors.
- Duties entered in `administration/inspection-roster.html`.
- Inspector enforces testing standards.

Inspection of Functional Requirements

Inspection of Functional Requirements

- Apply to Section 2 of requirements doc.

Inspection of Functional Requirements

- Apply to Section 2 of requirements doc.
- Test procedure defined in terms of the HTML document elements.

Inspection of Functional Requirements

- Apply to Section 2 of requirements doc.
- Test procedure defined in terms of the HTML document elements.
- Specifically:

Tag	Denotation	Required Inspection
<Hn>	Numbered section, level $1 \leq n \leq 6$	spelling, grammar, presentation style
<P>	Paragraph Pn, $n \geq 1$	spelling, grammar, presentation style
<IMG>	Image In, $n \geq 1$.	existence of image file, spelling, grammar, presentation style, image quality, aesthetics
<A>	Anchor An, $n \geq 1$.	existence of anchor target

Inspection Test Plan and Record

Inspection Test Plan and Record

- *Test plan* is a five-column table.

Inspection Test Plan and Record

- *Test plan* is a five-column table.
- A row for each component of rqmts.

Inspection Test Plan and Record

- *Test plan* is a five-column table.
- A row for each component of rqmts.
- Columns are:

Test Plan and Record, cont'd

i. component denotation, e.g., 2.2.1 I3

Test Plan and Record, cont'd

i. component denotation, e.g., 2.2.1 I3

ii. inspector initials

Test Plan and Record, cont'd

i. component denotation, e.g., 2.2.1 I3

ii. inspector initials

iii. date

Test Plan and Record, cont'd

i. component denotation, e.g., 2.2.1 I3

ii. inspector initials

iii. date

iv. status, "DONE" or "FIX"

Test Plan and Record, cont'd

i. component denotation, e.g., 2.2.1 I3

ii. inspector initials

iii. date

iv. status, "DONE" or "FIX"

v. remarks

Test Plan and Record, cont'd

- *Test record* is a completed plan.

Test Plan and Record, cont'd

- *Test record* is a completed plan.

- For example:

Component	Inspector	Date	Status	Remarks
2.1	GLF	24oct14	DONE	
2.1P1	GLF	24oct14	DONE	
2.1I1	GLF	24oct14	FIX	image quality is poor due to small size; rescan from original into larger GIF format
2.1A1	GLF	24oct14	FIX	the anchor target is not found; check file existence and protection

2.1.1	GLF	24oct14	DONE	
2.1.1P1	GLF	24oct14	DONE	
2.1.1I1	GLF	24oct14	DONE	
2.1.1P2	GLF	24oct14	DONE	grammatical error in sentence 1: "num-ber" => "numbers"
2.1.1P3	GLF	24oct14	DONE	
. . .	. . .	. . .	. . .	

Plan and Record, cont'd

- Plans stored in project subdirectory

  `testing/requirements`

Plan and Record, cont'd

- Plans stored in project subdirectory

  `testing/requirements`

- Testing file has same root filename as rqmts file, with added suffix "-test".

Plan and Record, cont'd

- For example, test plan for

  `requirements/ui-overview.html`

  is stored in

  `testing/requirements/`
  `ui-overview-test.html`

Procedural Details

Procedural Details

- Inspector job is *identify* problems, not correct.

Procedural Details

- Inspector job is *identify* problems, not correct.
- Inspector responsible for test plans.

Procedural Details

- Inspector job is *identify* problems, not correct.
- Inspector responsible for test plans.
- Inspector responsible for performing tests.

Procedural Details, cont'd

- Time-line:

Procedural Details, cont'd

- Time-line:

1. Each member checks in work.

Procedural Details, cont'd

- Time-line:

1. Each member checks in work.
2. Everyone updates their work directory.

Procedural Details, cont'd

- Time-line:

1. Each member checks in work.
2. Everyone updates their work directory.
3. Inspectors perform inspection.

Procedural Details, cont'd

- Time-line:

1. Each member checks in work.
2. Everyone updates their work directory.
3. Inspectors perform inspection.
4. Inspectors check in test record.

Procedural Details, cont'd

- Time-line:

1. Each member checks in work.
2. Everyone updates their work directory.
3. Inspectors perform inspection.
4. Inspectors check in test record.
5. Librarian updates release directory.

Procedural Details, cont'd

- Time-line:

1. Each member checks in work.
2. Everyone updates their work directory.
3. Inspectors perform inspection.
4. Inspectors check in test record.
5. Librarian updates release directory.
6. At subsequent team meeting, discuss.