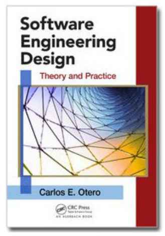
Chapter 8: Principles of Construction Design

Session I: Overview of Construction Design
Flow-, State-, and Table-Based Designs

Software Engineering Design: Theory and Practice
by Carlos E. Otero



Session's Agenda

- ➢ Overview of Construction Design
  - ✓ Algorithmic Viewpoint
  - ✓ Stylistic Viewpoint
- ➢ Algorithmic Viewpoint
  - ✓ Flow-based
    - ▪ UML Activity Diagrams
  - ✓ State-based
    - ▪ UML State Diagrams
  - ✓ Table-based
- ➢ What's next…

WHAT IS CONSTRUCTION DESIGN?

- Transition from the software design phase to the construction phase should occur with minimal effort.

- In some cases, component designs provide enough detail to allow their transformation from design artifact to code easily.

- In other cases, a more fine-grained level of design detail is required.

- Construction design is the lowest level of detailed design that addresses the modeling and specification of function implementations.
  - ✓ This is necessary to evaluate the quality of the system at the construction level, e.g., modifiability, testability, performance, complexity, etc.
  - ✓ Construction design deals mostly with the analysis and design of algorithms. The IEEE refers to this form of design as designing using a "The Algorithmic Viewpoint" [1]

WHAT IS CONSTRUCTION DESIGN?

- The algorithm viewpoint addresses construction design from a dynamic (behavioral) perspective, which provides the description of operations (such as methods and functions), the *internal details* and *logic* of each design entity [1] .

- The algorithmic viewpoint can be realized using the following:
  - ✓ Graphical Designs
    - Flow-based
    - State-based
  - ✓ Tabular Designs
    - Lead to table-based design and implementation

- The algorithm viewpoint minimizes complexity during construction by providing details required by programmers to implement the function's code.

WHAT IS CONSTRUCTION DESIGN?

- A separate but closely related task performed to achieve quality at the construction level is the enforcing of styles for software construction. We'll refer to this as the "Stylistic Viewpoint" of construction design.
  - ✓ These styles play a significant role in shaping the systems' modifiability quality attribute!

- In the construction design activity, styles are used to provide a consistent approach for structuring code by defining styles for code elements, such as:
  - ✓ Code formatting
  - ✓ Naming conventions
  - ✓ Documentation
  - ✓ Etc.

- The application of construction styles are mostly an activity that occurs during construction, however, due to the power of today's modeling tools, the application of styles are prevalent during the detailed design phase.

WHY STUDY CONSTRUCTION DESIGN?

- From the algorithmic viewpoint, construction design is important because it provides the means for evaluating different implementations for a particular function before committing to it.

- Behavioral designs at this level provide the means to:
  - ✓ Evaluate a function's completeness, complexity, testability and maintainability.
  - ✓ They also provide the means for analysts of algorithms in regard to time-space performance and processing logic prior to implementation [1]. This can have significant meaning when designing for *performance*!

- Finally, since they provide a representation of the code through graphical and tabular ways, they increase collaborative evaluation efforts, since other members without knowledge of programming languages can evaluate the design and contribute their input.
  - ✓ These collaboration efforts can lead to improvement in future phases, for example the testing phase, where construction designs can be used to generate unit test cases, or the maintenance phase, where construction designs can be used to increase knowledge and understanding of the software behavior.

WHY STUDY CONSTRUCTION DESIGN?

- From the stylistic viewpoint, construction design is important because it provides heuristics for establishing a common criteria for evaluating the quality of the structure of code, which has direct effect on code readability, and therefore maintenance.
- Code that exhibit low quality in terms of readability results in higher maintenance cost, since it requires more effort to understand [2].
- Construction styles are important during the design phase so that generation of code form design models can be done correctly.
- From the construction phase perspective, construction styles serve as blueprint that ensures consistency among teams of developers. Finally, as mentioned before, during the testing and maintenance phase, construction styles increase readability and understanding of the code, which results in minimized cost.

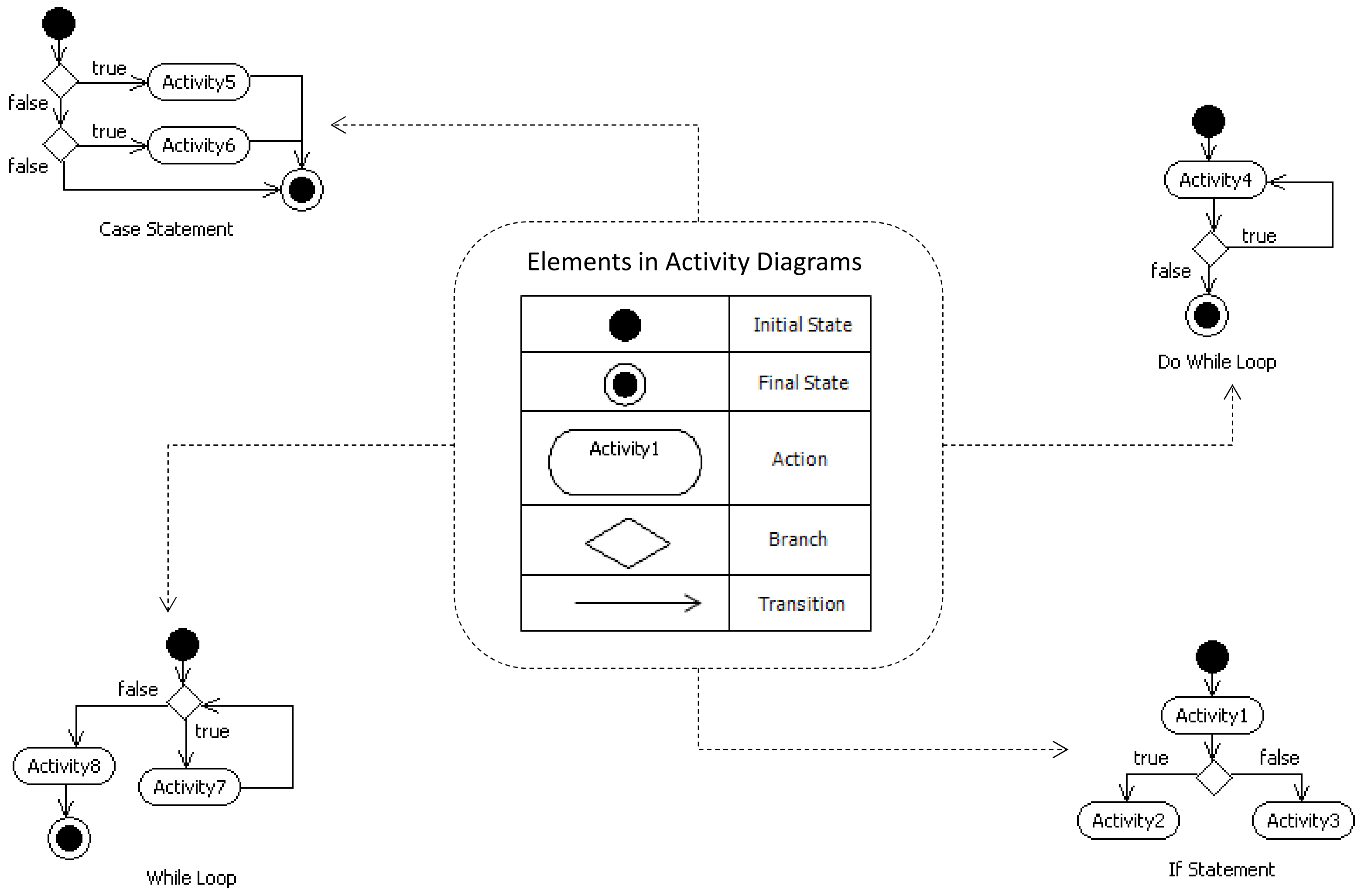
BEHAVIORAL CONSTRUCTION DESIGN

- Behavioral designs at the construction level are used to model complex logic that is unknown or difficult to understand. Popular examples include:
  - ✓ Flow-based design
  - ✓ State-based design
  - ✓ Table-based design

- Flow-Based design provide a systematic methodology for specifying the logic and structure of operations using a graphical approach. Two popular approaches for creating flow-based designs include:
  - ✓ Flowcharts
  - ✓ UML activity diagrams

- Both work well for modeling the internal flow of routines because they can be defined using sequential process flows, loops, and other complex business logic or algorithms.
  - ✓ UML activity diagrams provide powerful constructs for modeling complex logic at different stages of the SDLC, however, when applied towards modeling logic, activity diagrams are just another version of flowcharts.

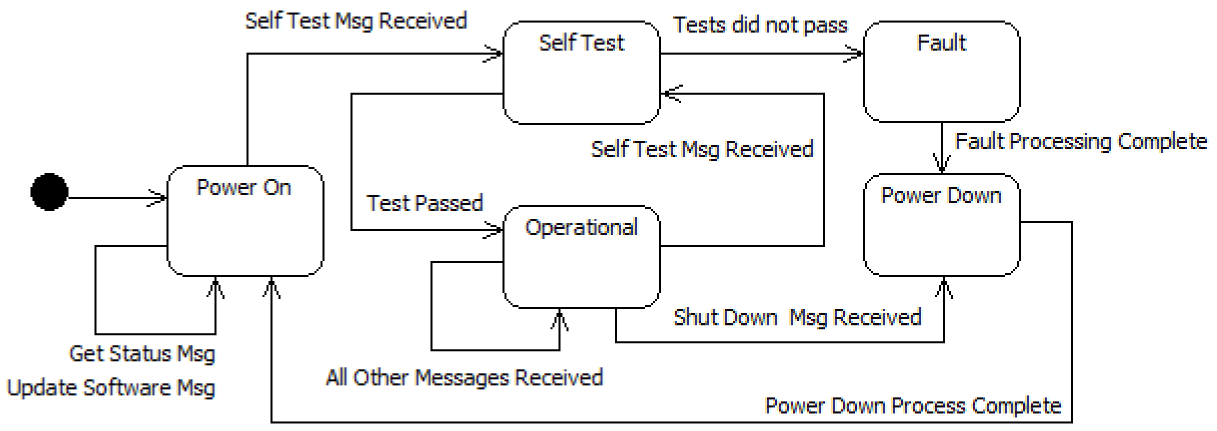
FLOW-BASED CONSTRUCTION DESIGN



STATE-BASED CONSTRUCTION DESIGN

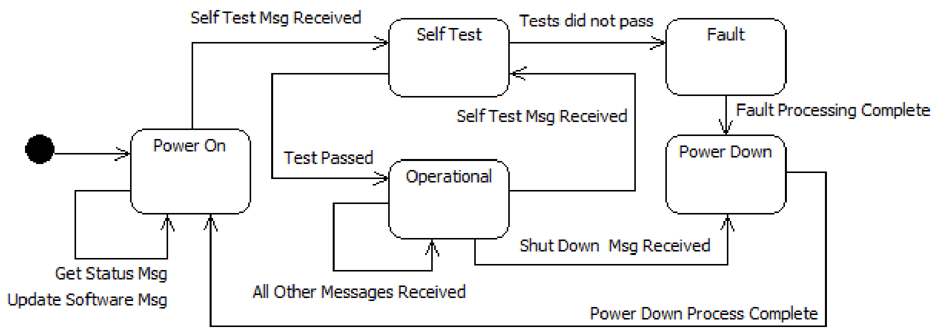
- Flow-based designs can be used to model operational logic by identifying the transitions from activity to activity required to perform an operation.
- However, in some cases, the operational logic of a function or system is dictated by the different states that the system exhibits during its lifetime. That is, certain activities can only be performed when a system is in a particular state.
- When this occur, the operational logic of a system can be modeled as a state machine using a (UML) state diagram.
- State diagrams are typically used to model the behavior of complete system. However, in many practical applications, the state diagram acts as model for designing the logical structure of one operation that executes the state machine.

State-Based Construction Design



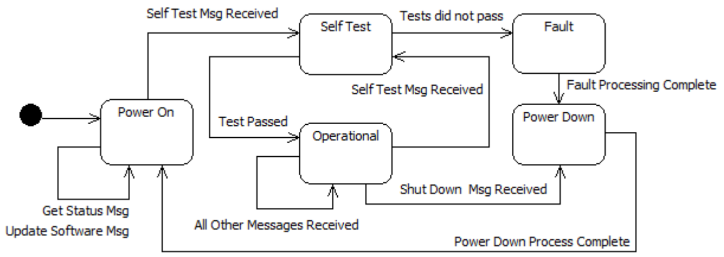
State-Based Construction Design

```
// The state machine's execute method.
void EmbeddedComponent::execute() {

  // Execute the state machine. _compnentState is a member variable of the
  // EmbeddedComponent class.
  switch( _componentState ) {

      case PowerOnState:
        // Execute in the power on state. When finished, allow the
        // executing function to determine if a state change is required
        // (or not) and set the state appropriately.  This capability is
        // provided by executing functions in all other states.
        executePowerOnState();
        break;

      case SelfTestState:
        // Execute in the self test state.
        executeSelfTestState();
        break;

      case OperationalState:
        // Execute in the operational state.
        executeOperationalState();
        break;

      case FaultState:
        // Execute in the fault state.
        executeFaultState();
        break;

      case PowerDownState:
        // Execute in the power down state.
        executePowerDownState();
        break;

      default:
        // invalid state, log error.
        break;
  }
}
```



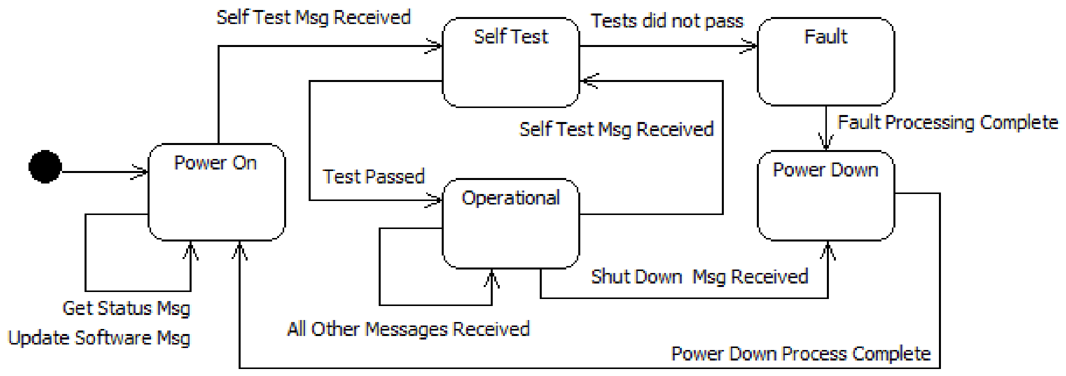
State-Based Construction Design

```
void EmbeddedComponent::executePowerOnState() {

  // Assume messages are received and placed in a blocking message queue.
  // Therefore, the messageQueue.read call is a blocking call.
  Message* message = messageQueue.read(WAIT_FOREVER);

  // Retrieve the message's id.
  MessageIdType messageId = message->getId();

  // This state only processes three messages according to the state
  // diagram.
  if( messageId == UpdateSoftwareMsgId ) {
    // Cast message to an UpdateSoftwareMsg.
    // Retrieve the software image from the message and update software.
  }
  else if( messageId == GetStatusMsgId ) {
    // Retrieve status from File System and return to client.
  }
  else if( messageId == SelfTestMsgId ) {
    // Cast message to a SelfTestMsg.
    // Retrieve the type of self test and change state.
    selfTestType_ = message->getTestType();
    _componentState = SelfTestState;
  }
  else {
    // Any other message received in this state results in an error.
    // Log the specific error here and do not change state.
  }
}
```



STATE-BASED CONSTRUCTION DESIGN

```
void EmbeddedComponent::executeSelfTestState() {

  // No messages are processed during self test.

  // Perform either a simple, normal, or advanced test. Advanced tests
  // perform a complete test of the system, therefore they take longer to
  // complete.
  if( performTest(_selfTestType) ) {

    // Software and hardware are working properly. Log results and change
    // state to the operational state.
    _componentState = OperationalState;
  }
  else {
    // Faulty system software or hardware!  Log results and change state
    // to the Fault state.
    _componentState = FaultState;
  }
}
```



STATE-BASED CONSTRUCTION DESIGN

```
void EmbeddedComponent::executeOperationalState() {

  // Assume messages are received and placed in a blocking message queue.
  // Therefore, the messageQueue.read call is a blocking call.
  Message* message = messageQueue.read(WAIT_FOREVER);

  // Retrieve the message's id.
  MessageIdType messageId = message->getId();

  // Process messages according to the state diagram.
  if( /* messageId == x */ ) {
    // Process message x.
  }
  else if( /* messageId == y */ ) {
    // Process message y.
  }
  else if (/* ... */ {
    // ...
  }
  else {
    // Invalid message. Log error.
  }
}
```

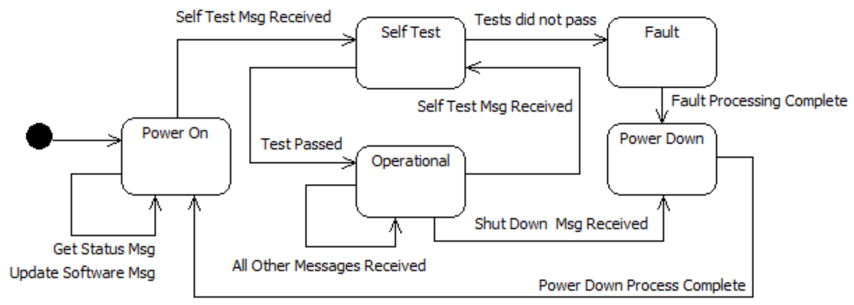


TABLE-BASED CONSTRUCTION DESIGN

- Many times, the internal logic of routines are made up of complex conditional statements; each statement evaluating a condition (i.e., a cause) and providing some action (i.e., an effect) as result.
- This can lead to an increasingly complex nesting structure that is error-prone, hard to read, and hard to maintain.
- In these cases, the logic design can be managed using a Decision Table [3].
- A *decision table* is a well structured table that provides the means to formulate, evaluate, improve the design of complex problems that deal with cause and effect.

TABLE-BASED CONSTRUCTION DESIGN

- The fundamental structure of a decision table contains four main sections:
  - ✓ Condition
  - ✓ Action
  - ✓ Condition Entry
  - ✓ Action Entry

Condition	**Condition Entry**
Action	**Action Entry**

- The first section is the *Condition* section, which contains a list of all of the conditions present in the decision problem.
- The second section is the *Action* section, which contains a list of all possible outcomes that can result from one or more conditions occurring.
- The third and fourth sections are found in matrix form adjacent to the *Condition* and *Action* sections.
  - ✓ The matrix adjacent to the *Condition* section indicate all possible combinations of conditions for the decision problem, while the matrix adjacent to the *Action* section indicates the corresponding actions.

TABLE-BASED CONSTRUCTION DESIGN

- Three types of decision tables are as follow [3]:
  - ✓ Limited Entry Decision Table
  - ✓ Extended Entry Decision Table
  - ✓ Mixed Entry Decision Table

- Limited Entry Decision Table (LEDT)
  - ✓ Simplest type of decision table in which the condition section of the LEDT presents Boolean conditional statements.
  - ✓ That is, the condition section of the LEDT presents features of the design problem that are either present or not and their combined presence (or absence) trigger specific actions.
  - ✓ Therefore, the condition entry section of the LEDT consists of Boolean values, such as true or false, or yes or no that can be used to define different policies in the decision problem.
  - ✓ For a LEDT, the number of distinct elementary policies is 2^n, were n is the number of conditions in the condition section.

Table-Based Construction Design

- Limited Entry Decision Table (LEDT) – Example
  - ✓ Consider the LEDT design for a function that computes discounts for the purchase of mobile phones.
  - ✓ Two types of discounts are available, a store discount of $15, and a manufacturer discount of $30.

Get Phone Discount	**P1**	**P2**	**P3**	**P4**
Store Discount	T	T	F	F
Manufacturer Discount	T	F	T	F
$15 Discount	x	x		
$30 Manufacturer Discount	x		x	
No Discount ($0)				x

TABLE-BASED CONSTRUCTION DESIGN

- Sample Implementation…

Get Phone Discount	P1	P2	P3	P4
Store Discount	T	T	F	F
Manufacturer Discount	T	F	T	F
$15 Discount	x	x		
$30 Manufacturer Discount	x		x	
No Discount ($0)				x

```
int getPhoneDiscount() {

  const int StoreDiscount = 15;
  const int ManufacturerDiscount = 30;

  // The total added phone discount.
  int phoneDiscount = 0;

  // Determine if the store discount applies.
  if( isStoreDiscountActive ) {

    // Apply the store's discount.
    phoneDiscount += StoreDiscount;
  }

  // Determine if the manufacturer discount applies.
  if( isManufacturerDiscountActive ) {

    // Apply the manufacturer's discount.
    phoneDiscount += ManufacturerDiscount;
  }

  // Return the total added phone discount.
  return phoneDiscount;
}
```

TABLE-BASED CONSTRUCTION DESIGN

- Extended Entry Decision Table (EEDT)
  - ✓ Whereas the Condition and Action sections of LEDTs contain complete questions and actions, the Condition and Action sections of the extended entry decision table (EEDT) are extended into the Action Entry section.
  - ✓ That is, in LEDTs, the Condition section contained information that could be used to ask a complete questions, such as "*is there a store discount in effect?*"
  - ✓ In EEDT, the Condition and Condition Entry sections of the table are required to formulate a complete question, such as "*Is the customer a regular, preferred, or VIP customer?*"
  - ✓ Similarly, the Action section must be combined with the Action Entry section of the decision table to formulate a complete action, such as "*add a free car kit to the purchase.*"
  - ✓ In addition, the number of possible values for each condition and action in EEDTs are not bounded to two.

Condition	**Condition Entry**
Action	**Action Entry**

TABLE-BASED CONSTRUCTION DESIGN

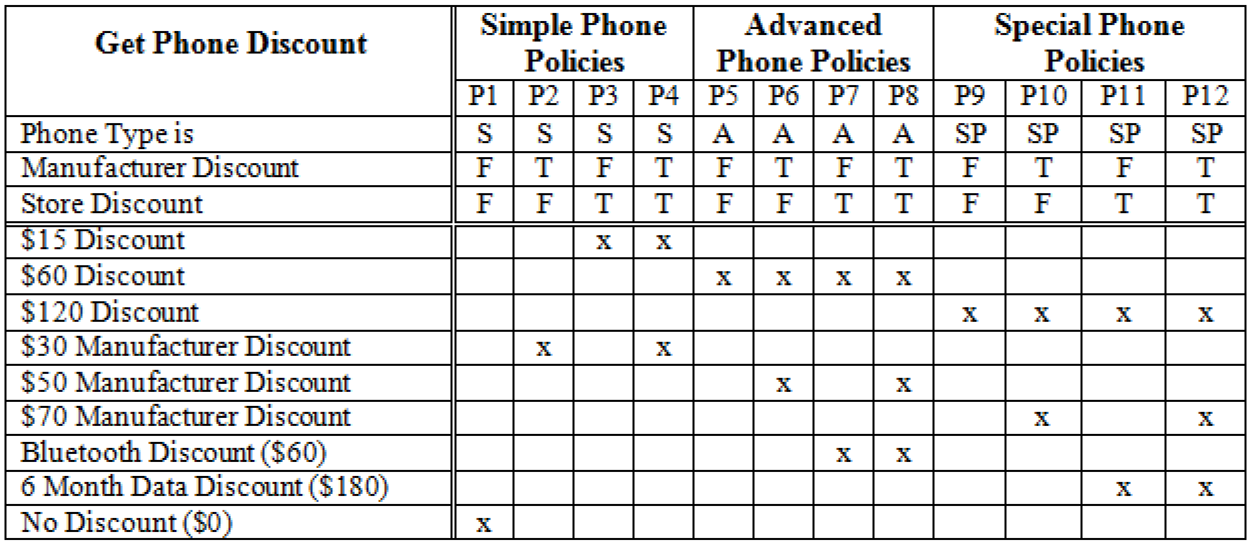
- Extended Entry Decision Table (EEDT) – Example

Get Phone Discount	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**
Customer Type is	REG	REG	PRE	PRE	VIP	VIP
Credit Score is	BAD	GOOD	BAD	GOOD	BAD	GOOD
Discount	$0	$15	$10	$25	$50	$100
Add a Free	HOLSTER	CHARGER	BLUE TOOTH	CAR KIT	DATA PLAN	CAR KIT & DATAPLAN

- Notice that:
  - ✓ The number of possible values for each condition and action in EEDTs are not bounded to two.
  - ✓ Therefore, the number of policies for EEDT is the product of the number of possible values for each condition, denoted by $\prod_{i=1}^{c} V_i = V_1 \times V_2 \times \ldots \times V_c$
    - Where c is the number of conditions
    - V_i is the number of values for condition i.
  - ✓ In this example, the number of policies are 3 x 2 = 6.
    - 3 different value types for Condition 1 (i.e., cutomer type is REG/PRE/VIP)
    - 2 different value types for Condition 2 (i.e., credit score is GOOD/BAD)

TABLE-BASED CONSTRUCTION DESIGN

- Mixed Entry Decision Table (MEDT)
  - ✓ Combines LEDTs and EEDTs into one MEDT
    - That is, conditions can be questions, with various answers or
    - Binary features that are present or not (true or false).



Get Phone Discount	**Simple Phone Policies**				**Advanced Phone Policies**				**Special Phone Policies**			
	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Phone Type is	S	S	S	S	A	A	A	A	SP	SP	SP	SP
Manufacturer Discount	F	T	F	T	F	T	F	T	F	T	F	T
Store Discount	F	F	T	T	F	F	T	T	F	F	T	T
$15 Discount			x	x								
$60 Discount					x	x	x	x				
$120 Discount									x	x	x	x
$30 Manufacturer Discount		x		x								
$50 Manufacturer Discount						x		x				
$70 Manufacturer Discount										x		x
Bluetooth Discount ($60)							x	x				
6 Month Data Discount ($180)											x	x
No Discount ($0)	x											

TABLE-BASED CONSTRUCTION DESIGN

```
int getPhoneDiscount(const Phone& phone) {

  int totalDiscount = 0; // The total computer discount.

  if( phone.getType() == SIMPLE_PHONE ) {

    if( phone.getDiscountType() == MANUFACTURER_DISCOUNT) {
      // Add $30 manufacturer's discount to totalDiscount.
    }
    else if( phone.getDiscountType() == STORE_DISCOUNT ) {
      // Add $15 store discount to totalDiscount.
    }
    else if( phone.getDiscountType() == COMBINED_DISCOUNT ) {
      // Add $30 and $15 to totalDiscount.
    }
    else { // No discount.
    }
  }
  else if( phone.getType() ==  ADVANCED_PHONE ) {

    // Add $60 default advanced phone discount to totalDiscount.

    if( phone.getDiscountType() == MANUFACTURER_DISCOUNT) {
      // Add additional $50 manufacturer's discount to totalDiscount.
    }
    else if( phone.getDiscountType() == STORE_DISCOUNT ) {
      // Add additional Bluetooth ear piece discount ($60) to
      // totalDiscount.
    }
    else if( phone.getDiscountType() == COMBINED_DISCOUNT ) {
      // Add additional $50 and $60 to totalDiscount.
    }
    else { // No additional discount.
    }
  }
  else if( phone.getType() ==  SPECIAL_PHONE ) {

    // Add $120 default special phone discount to totalDiscount.

    if( phone.getDiscountType() == MANUFACTURER_DISCOUNT) {

      // Add additional $70 manufacturer's discount to totalDiscount.
    }
    else if( phone.getDiscountType() == STORE_DISCOUNT ) {
      // Add additional 6 month data plan discount ($180).
    }
    else if( phone.getDiscountType() == COMBINED_DISCOUNT ) {
      // A additional $70 and $180 to totalDiscount.
    }
    else { // No additional discount.
    }
  }
  return totalDiscount; // Return the computed phone discount.
}
```

Get Phone Discount	Simple Phone Policies				Advanced Phone Policies				Special Phone Policies			
	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Phone Type is	S	S	S	S	A	A	A	A	SP	SP	SP	SP
Manufacturer Discount	F	T	F	T	F	T	F	T	F	T	F	T
Store Discount	F	F	T	T	F	F	T	T	F	F	T	T
$15 Discount			x	x								
$60 Discount					x	x	x	x				
$120 Discount									x	x	x	x
$30 Manufacturer Discount		x		x								
$50 Manufacturer Discount						x		x				
$70 Manufacturer Discount										x		x
Bluetooth Discount ($60)							x	x				
6 Month Data Discount ($180)											x	x
No Discount ($0)	x											

These are equivalent!

That 's a lot of conditional statements! Let's see how we can leverage of table-based construction to reduce complexity…

Table-Based Construction Design

- Table-based designs lead to efficient construction that has significant lower complexity.

```
struct Discounts {
  int smallStoreDiscount;          // For this example, it should be $15.
  int mediumStoreDiscount;         // For this example, it should be $60.
  int highStoreDiscount;           // For this example, it should be $120.
  int smallManufacturerDiscount;   // For this example, it should be $30.
  int mediumManufacturerDiscount;  // For this example, it should be $50.
  int highManufacturerDiscount;    // For this example, it should be $70.
  int bluetoothDiscount;           // For this example, it should be $60.
  int dataPlanDiscount;            // For this example, it should be $180.
};

// The discounts available for simple phones (SIM), advanced phones (ADV),
// and special phones (SPE), all accessible via discount keys (DK#).
Discounts discounts[] = {
  { 0, 0, 0, 0, 0, 0, 0, 0        }, // DK0, SIM/ No discounts.
  { 0, 60, 0, 0, 0, 0, 0, 0       }, // DK1, ADV/ Store's default discount.
  { 0, 0, 120, 0, 0, 0, 0, 0      }, // DK2, SPE/ Store's default discount.
  { 0, 0, 0, 30, 0, 0, 0, 0       }, // DK3, SIM/ Manufacturer's discount.
  { 0, 60, 0, 0, 50, 0, 0, 0      }, // DK4, ADV/ Manufacturer's discount.
  { 0, 0, 120, 0, 0, 70, 0, 0     }, // DK5, SPE/ Manufacturer's discount.
  { 15, 0, 0, 0, 0, 0, 0, 0       }, // DK6, SIM/ Special store discount.
  { 0, 60, 0, 0, 0, 0, 60, 0      }, // DK7, ADV/ Default & spec. store disc.
  { 0, 0, 120, 0, 0, 0, 0, 180    }, // DK8, SPE/ Default & spec. store disc.
  { 15, 0, 0, 30, 0, 0, 0, 0      }, // DK9, SIM/ All applicable discounts.
  { 0, 60, 0, 0, 50, 0, 60, 0     }, // DK10, ADV/ All applicable discounts.
  { 0, 0, 120, 0, 0, 70, 0, 180   }  // DK11, SPE/ All applicable disc.
};
```

Move the complexity of the previous conditional statements to a table!

TABLE-BASED CONSTRUCTION DESIGN

Compute the discount key!

Since we moved the complexity of the previous conditional statements to a table, all we have to do now to retrieve the discount is key into the table!

```
// The table-based version for retrieving discounts.
int getPhoneDiscount( const Phone& phone ) {

  // Compute the key for accessing the corresponding table row.
  int discountKey = phone.getType() + phone.getDiscountType();

  // Add all discounts associated with the discount key.
  int totalDiscount = discounts[discountKey].smallStoreDiscount +
                      discounts[discountKey].mediumStoreDiscount +
                      discounts[discountKey].highStoreDiscount +
                      discounts[discountKey].smallManufacturerDiscount +
                      discounts[discountKey].mediumManufacturerDiscount +
                      discounts[discountKey].highManufacturerDiscount +
                      discounts[discountKey].bluetoothDiscount +
                      discounts[discountKey].dataPlanDiscount;

  // Return the total discount.
  return totalDiscount;
}

// Create a simple phone with manufacturer's discount.
Phone phone(SIMPLE_PHONE, MANUFACTURER_DISCOUNT);

// Display the phone's discount.
cout<<"Total Phone Discount: "<<getPhoneDiscount( phone )<<endl;
```

TABLE-BASED CONSTRUCTION DESIGN

```
int getPhoneDiscount(const Phone& phone) {

  int totalDiscount = 0; // The total computer discount.

  if( phone.getType() == SIMPLE_PHONE ) {

    if( phone.getDiscountType() == MANUFACTURER_DISCOUNT) {
      // Add $30 manufacturer's discount to totalDiscount.
    }
    else if( phone.getDiscountType() == STORE_DISCOUNT ) {
      // Add $15 store discount to totalDiscount.
    }
    else if( phone.getDiscountType() == COMBINED_DISCOUNT ) {
      // Add $30 and $15 to totalDiscount.
    }
    else { // No discount.
    }
  }
  else if( phone.getType() ==  ADVANCED_PHONE ) {

    // Add $60 default advanced phone discount to totalDiscount.

    if( phone.getDiscountType() == MANUFACTURER_DISCOUNT) {
      // Add additional $50 manufacturer's discount to totalDiscount.
    }
    else if( phone.getDiscountType() == STORE_DISCOUNT ) {
      // Add additional Bluetooth ear piece discount ($60) to
      // totalDiscount.
    }
    else if( phone.getDiscountType() == COMBINED_DISCOUNT ) {
      // Add additional $50 and $60 to totalDiscount.
    }
    else { // No additional discount.
    }
  }
  else if( phone.getType() ==  SPECIAL_PHONE ) {

    // Add $120 default special phone discount to totalDiscount.

    if( phone.getDiscountType() == MANUFACTURER_DISCOUNT) {

      // Add additional $70 manufacturer's discount to totalDiscount.
    }
    else if( phone.getDiscountType() == STORE_DISCOUNT ) {
      // Add additional 6 month data plan discount ($180).
    }
    else if( phone.getDiscountType() == COMBINED_DISCOUNT ) {
      // A additional $70 and $180 to totalDiscount.
    }
    else { // No additional discount.
    }
  }
  return totalDiscount; // Return the computed phone discount.
}
```

Get Phone Discount	Simple Phone Policies				Advanced Phone Policies				Special Phone Policies			
	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Phone Type is	S	S	S	S	A	A	A	A	SP	SP	SP	SP
Manufacturer Discount	F	T	F	T	F	T	F	T	F	T	F	T
Store Discount	F	F	T	T	F	F	T	T	F	F	T	T
$15 Discount			x	x								
$60 Discount					x	x	x	x				
$120 Discount									x	x	x	x
$30 Manufacturer Discount		x		x								
$50 Manufacturer Discount						x		x				
$70 Manufacturer Discount										x		x
Bluetooth Discount ($60)							x	x				
6 Month Data Discount ($180)											x	x
No Discount ($0)	x											

These are equivalent!

```
// The table-based version for retrieving discounts.
int getPhoneDiscount( const Phone& phone ) {

  // Compute the key for accessing the corresponding table row.
  int discountKey = phone.getType() + phone.getDiscountType();

  // Add all discounts associated with the discount key.
  int totalDiscount = discounts[discountKey].smallStoreDiscount +
                      discounts[discountKey].mediumStoreDiscount +
                      discounts[discountKey].highStoreDiscount +
                      discounts[discountKey].smallManufacturerDiscount +
                      discounts[discountKey].mediumManufacturerDiscount +
                      discounts[discountKey].highManufacturerDiscount +
                      discounts[discountKey].bluetoothDiscount +
                      discounts[discountKey].dataPlanDiscount;

  // Return the total discount.
  return totalDiscount;
}

// Create a simple phone with manufacturer's discount.
Phone phone(SIMPLE_PHONE, MANUFACTURER_DISCOUNT);

// Display the phone's discount.
cout<<"Total Phone Discount: "<<getPhoneDiscount( phone )<<endl;
```

Notice how much simpler this version of the code looks!

WHAT'S NEXT…

- In this session, we presented construction design from the algorithmic viewpoint, including:
  - ✓ Flow-based design
  - ✓ State-based design
  - ✓ Table-based design

- In the next session, we will discuss another form of algorithmic design at the construction level, the Programming Design Language (PDL). We will also focuses on the stylistic view of construction design and on quality evaluation at the construction level.

References

- [1] IEEE. "IEEE Standard for Information Technology-Systems DESIGN-Software Design Descriptions." 2009, p. 175.
- [2] Collar, Emilio Jr. "An Investigation of Programming Code Textbase Readability Based on a Cognitive Readability Model." PhD thesis, University of Colorado at Boulder, 2005.
- [3] Hurley, Richard B. Decision Tables in Software Engineering. New York: Van Nostrand Reinhold, 1982.