Linguagens Específicas de Domínio (DSL)

Linguagens Específicas de Domíno

- Linguagens para realizar uma tarefa específica:
  - Procurar por uma string em um arquivo texto (grep, sed, awt)
  - Realizar uma query em um banco de dados (SQL)
  - Especificar um analisador léxico (Lex)
  - Especificar um analisador sintático (Yacc)
  - Desenhar um grafo (Graphviz)
  - Especificar dependências de compilação (makefile)
  - Especificar o layout de uma página Web (CSS)

- Objetivo:
  - Simplificar a programação de tarefas específicas

- Linguagens de Próposito Geral vs Linguagens Específicas de Domínio

DSL e Arquiteturas de Software

- DSL **não** são um conceito novo!
- Tendência recente: incentivar o uso de DSLs em vários sistemas

Domain-Specific Languages

Martin Fowler,
Addison-Wesley, 2011

Exemplo Introdutório: Gothic Security

- Empresa que vende sistemas de segurança.

- Principal sistema:
  - Permite ter um "compartimento secreto" em casas
  - Compartimento é aberto após uma sequência de eventos

- Exemplo (Miss Grant): compartimento é aberto quando ela:
  - fecha a porta do quarto
  - abre uma gaveta
  - liga uma luz

- Sistema inclui: sensores (que enviam mensagens) e controlador

Exemplo Introdutório

- Clientes podem "configurar" seus compartimentos secretos
- Família de sistemas:
  - Reúsa componentes e comportamentos
  - Mas com diferenças importantes
- Objetivo:
  - Projetar esse sistema de forma a facilitar sua instalação

Miss Grant's Controller

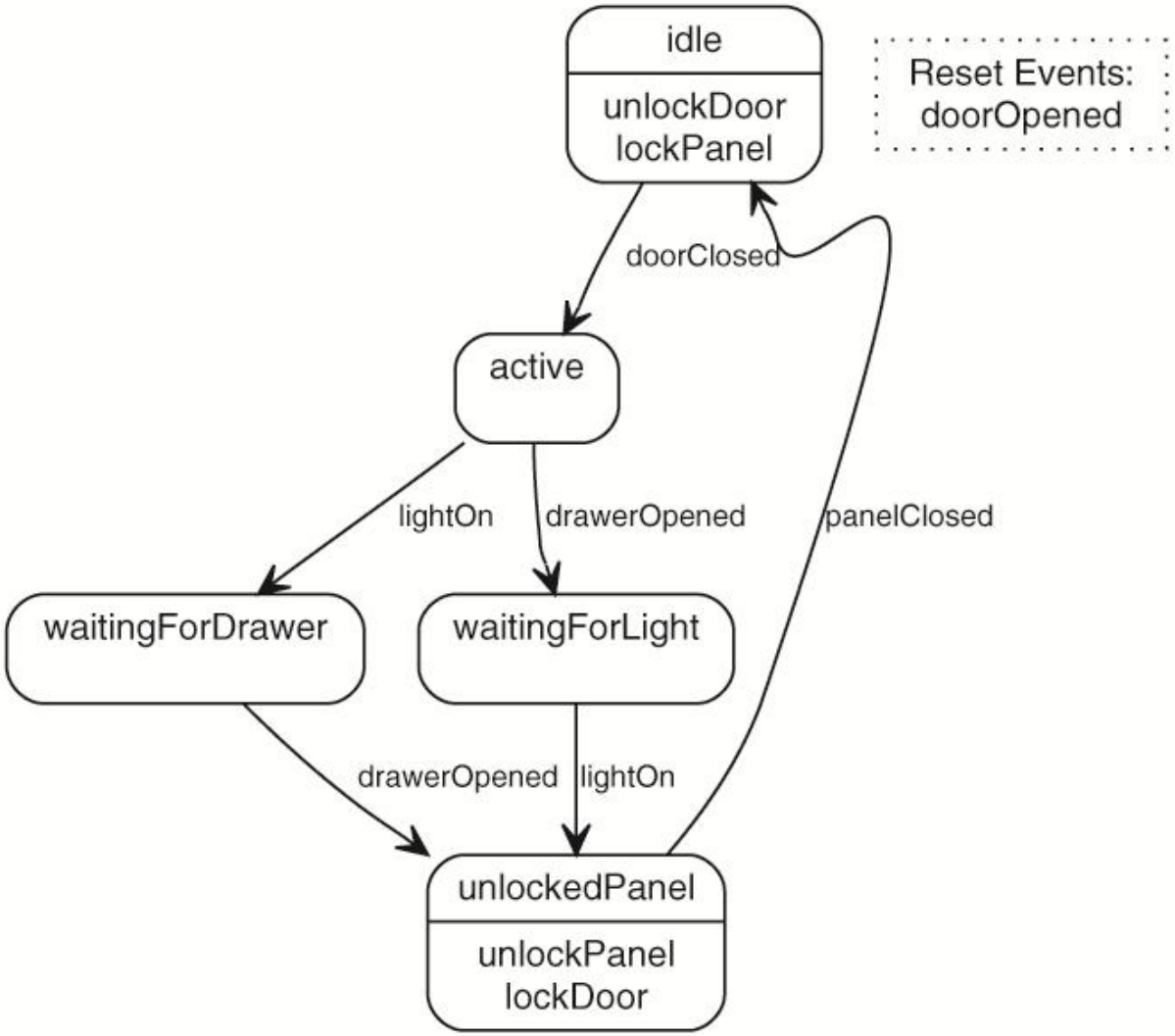
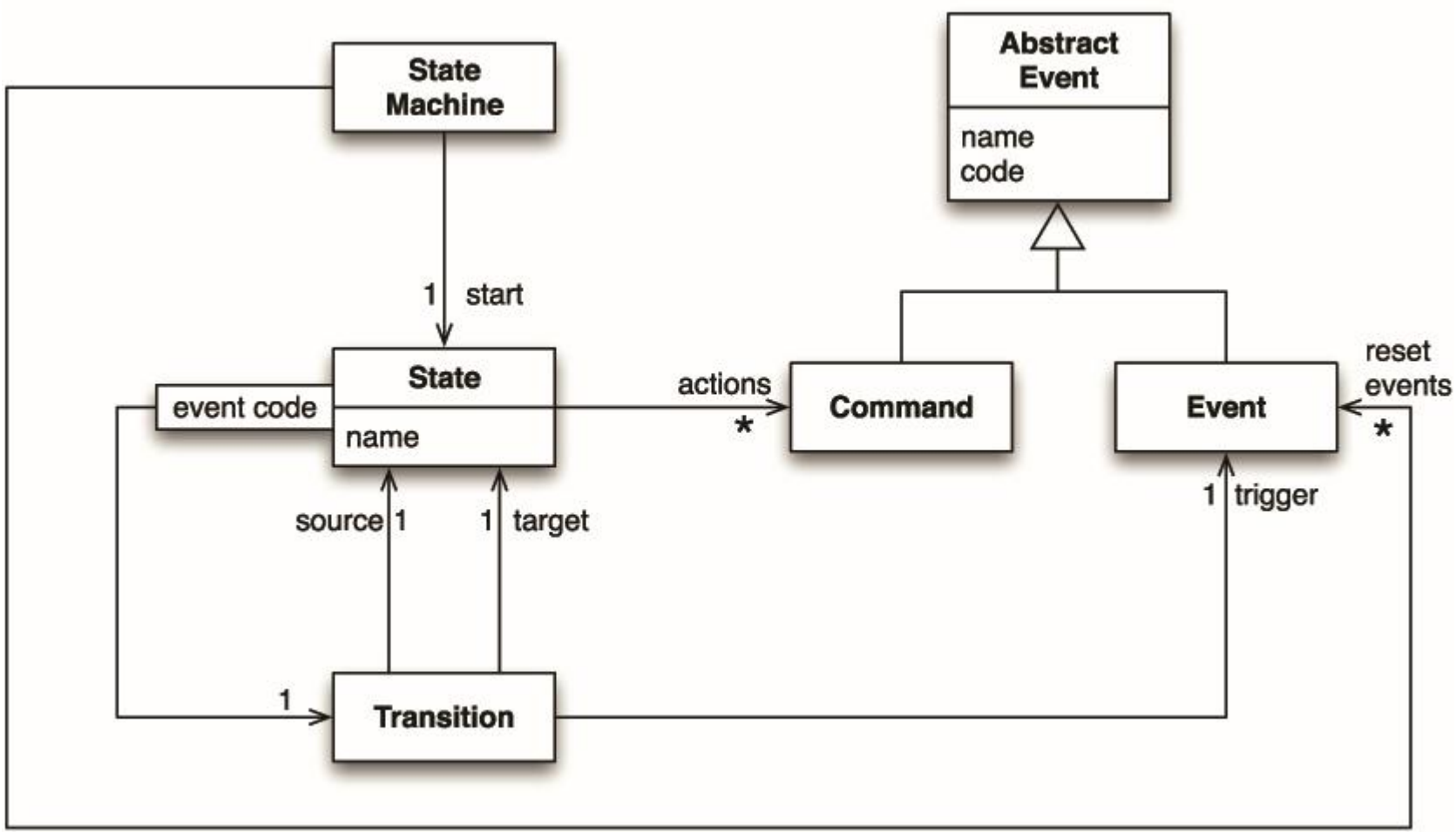


Diagrama de Classes



Programming Miss Grant's Controller

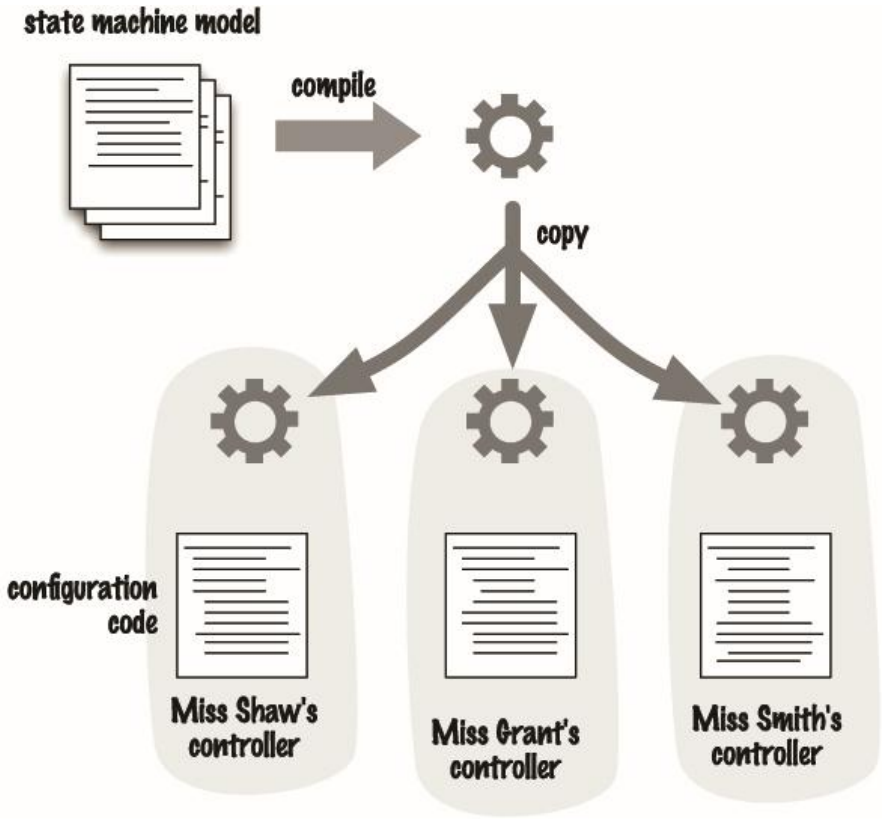
```
Event doorClosed = new Event("doorClosed", "D1CL");
Event drawerOpened = new Event("drawerOpened", "D2OP");
Event lightOn = new Event("lightOn", "L1ON");
Event doorOpened = new Event("doorOpened", "D1OP");
Event panelClosed = new Event("panelClosed", "PNCL");

Command unlockPanelCmd = new Command("unlockPanel", "PNUL");
Command lockPanelCmd = new Command("lockPanel", "PNLK");
Command lockDoorCmd = new Command("lockDoor", "D1LK");
Command unlockDoorCmd = new Command("unlockDoor", "D1UL");

State idle = new State("idle");
State activeState = new State("active");
State waitingForLightState = new State("waitingForLight");
State waitingForDrawerState = new State("waitingForDrawer");
State unlockedPanelState = new State("unlockedPanel");

StateMachine machine = new StateMachine(idle);
```

Programming Miss Grant's Controller

```
idle.addTransition(doorClosed, activeState);
 idle.addAction(unlockDoorCmd);
 idle.addAction(lockPanelCmd);

activeState.addTransition(drawerOpened,
 waitingForLightState);
 activeState.addTransition(lightOn,
 waitingForDrawerState);

waitingForLightState.addTransition(lightOn, unlockedPanelState);

waitingForDrawerState.addTransition(drawerOpened,
 unlockedPanelState);

unlockedPanelState.addAction(unlockPanelCmd);
 unlockedPanelState.addAction(lockDoorCmd);
 unlockedPanelState.addTransition(panelClosed,idle);

machine.addResetEvents(doorOpened);
```

Single library + multiple configurations

- Projeto permite separação entre:
  - Código comum (máquina de estados)
  - Código de configuração



Alternativa I: Configuração XML

```
<stateMachine start = "idle">
 <event name="doorClosed" code="D1CL"/>
 <event name="drawerOpened" code="D2OP"/>
 <event name="lightOn" code="L1ON"/>
 <event name="doorOpened" code="D1OP"/>
 <event name="panelClosed" code="PNCL"/>

 <command name="unlockPanel" code="PNUL"/>
 <command name="lockPanel" code="PNLK"/>
 <command name="lockDoor" code="D1LK"/>
 <command name="unlockDoor" code="D1UL"/>

 <state name="idle">
 <transition event="doorClosed" target="active"/>
 <action command="unlockDoor"/>
 <action command="lockPanel"/>

 </state>
 <state name="active">
 <transition event="drawerOpened" target="waitingForLight"/>
 <transition event="lightOn" target="waitingForDrawer"/>
```

Alternativa I: Configuração XML

- Vantagem:
  - Não precisa ser compilado.
  - Novas instalações não requerem a distribuição de um novo JAR
  - “Cada vez mais, programamos mais em XML e menos em LPs”

- Desvantagem: sintaxe

Alternativa II: DSL

```
events
 doorClosed D1CL
 drawerOpened D2OP
 lightOn L1ON
 doorOpened D1OP
 panelClosed PNCL end

resetEvents
 doorOpened
end

commands
 unlockPanel PNUL
 lockPanel PNLK
 lockDoor D1LK
 unlockDoor D1UL
 end
```

```
state idle
 actions {unlockDoor lockPanel}
 doorClosed => active
end

state active
 drawerOpened => waitingForLight
 lightOn => waitingForDrawer
end

state waitingForLight
 lightOn => unlockedPanel
end

state waitingForDrawer
 drawerOpened => unlockedPanel
end

state unlockedPanel
 actions {unlockPanel lockDoor}
 panelClosed => idle
end
```

DSL Externas e Internas

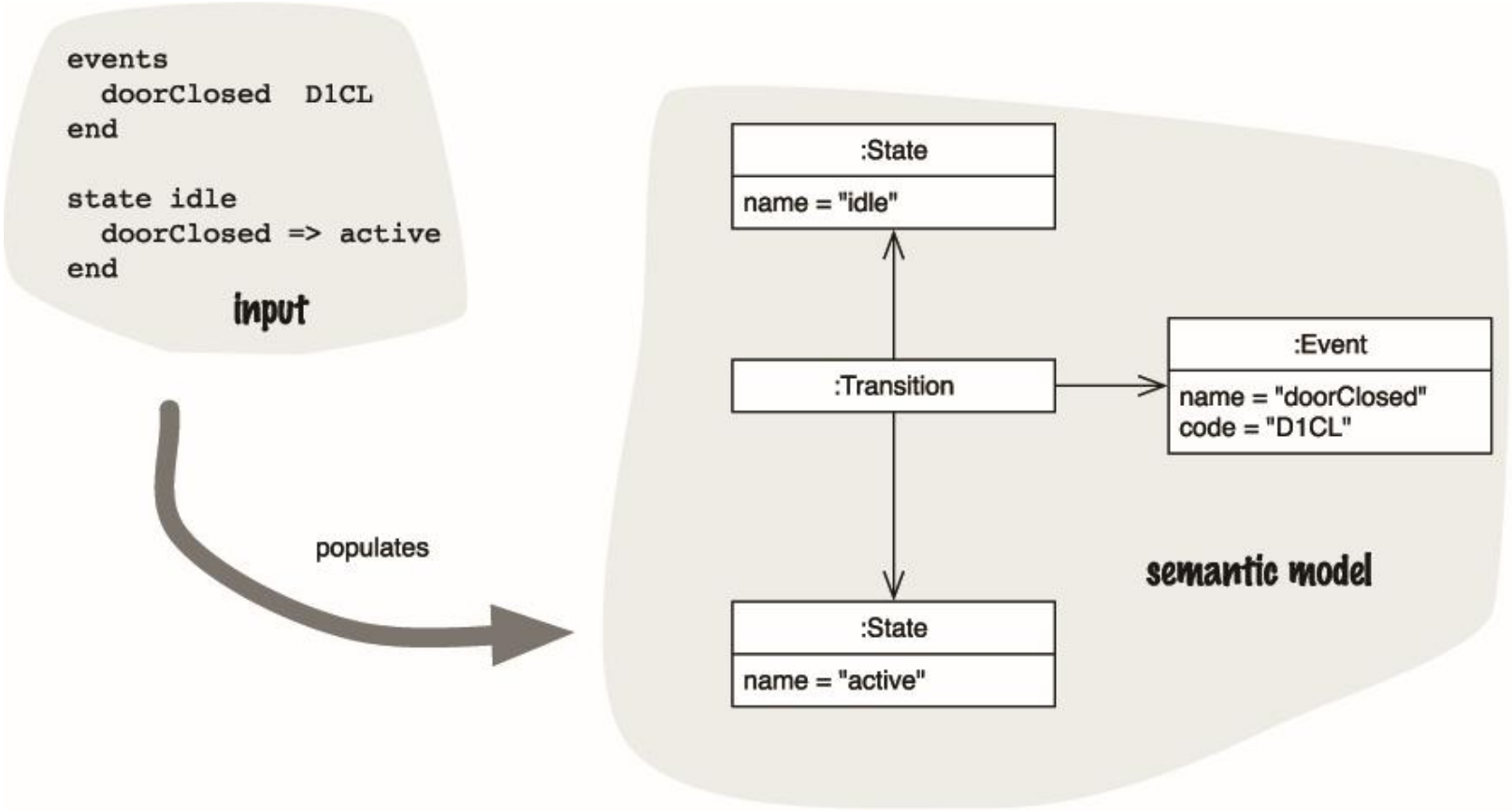
- Externas: DSL com uma sintaxe própria
- Internas: DSL que “pega carona” na sintaxe de uma LP

Exemplo de DSL Interna

```
public class BasicStateMachine extends StateMachineBuilder {
 Events doorClosed, drawerOpened, lightOn, panelClosed;
 Commands unlockPanel, lockPanel, lockDoor, unlockDoor;
 States idle, active, waitingForLight, waitingForDrawer,
   unlockedPanel;
 ResetEvents doorOpened;

 protected void defineStateMachine() {
   doorClosed. code("D1CL");
   drawerOpened. code("D2OP");
   lightOn. code("L1ON");
   panelClosed.code("PNCL");

   doorOpened. code("D1OP");
   unlockPanel.code("PNUL");
   lockPanel. code("PNLK");
   lockDoor. code("D1LK");
   unlockDoor. code("D1UL");

   idle
   .actions(unlockDoor, lockPanel)
   .transition(doorClosed).to(active);
```

Modelo Semântico

- DSLs devem ser usadas para popular um modelo semântico
- Modelo semântico: representação de entidades do domínio



Modelo Semântico

- Modelo pode ser testado, analisado, estudado etc em separado
- Maior benefício: modelo bem planejado (e não a DSL em si)
- DSL é uma fachada para um modelo semântico do sistema alvo
- Processo de desenvolvimento:
  - Primeiro passo: Modelo
  - Segundo passo: DSL
- Comparação MDA/MDD:
  - Semelhança: modelo é a parte central do projeto
  - Diferença: não objetiva geração de código a partir do modelo

Chapter 2 - Using Domain-Specific Languages

Defining DSL

- Domain-specific language (noun): a computer programming language of limited expressiveness focused on a particular domain

- Two main categories:
  - External DSLs
  - Internal DSLs

External and Internal DSLs

- External DSL:
  - A language separate from the main language of the application
  - Has a custom syntax
  - But using another language's syntax is also common (e.g. XML)

- Internal DSL:
  - Particular way of using a general-purpose language (GPL)
  - A script in an internal DSL is valid code in its GPL, but only uses a subset of the language's features in a particular style
  - The result should have the feel of a custom language, rather than its host language

Exemplo de DSL Interna

```
Order o = new Order();
Product p1 = new Product(1,Product.find(“Billy”));
o.addProduct(p1);
Product p2 = new Product(2,Product.find(”Janso"));
o.addProduct(p2);
Product p3 = new Product(4,Product.find(“Traby"));
o.addProduct(p3);
o.setPriorityRush(true);
customer.addOrder(o);
```

Exemplo de DSL Interna

```
customer.newOrder()
        .with(1, "Billy")
        .with(2, "Janso")
        .with(4, "Traby")
        .priorityRush()
        .done();
```

Exemplo de DSL Interna

```
public class Customer {
    ...
    public OrderBuilder newOrder() {
      return new OrderBuilder(this);
    }
}
```

Exemplo de DSL Interna

```
public class OrderBuilder {
  // ...

  public OrderBuilder(Customer customer) {
    this.customer = customer;
    this.order = new Order();
  }

  public OrderBuilder with(int id, String name) {
     order.addProduct(new Product(id, name));
     return this;
  }

  public OrderBuilder priorityRush() {
    order.setPriorityRush(true);
    return this;
  }

  public void done() {
    customer.addOrder(this.order);
  }
```

Boundaries of DSL: Internal DSLs

- Differences between an internal DSL and a command-query API:
  - The methods of an internal DSL often only make sense in the context of a larger expression in the DSL
  - In the Java internal DSL example earlier, I had a method called **to** that specified the target state of a transition
  - Such a method would be a bad name in a command-query API, but fits inside a phrase like:
    - **transition(lightOn).to(unlockedPanel)**

- An internal DSL should have the feel of putting together whole sentences, rather than a sequence of disconnected commands

- When forming a DSL expression, you limit yourself to a small subset of the general language features.
- It's common to avoid conditions, looping constructs, and variables

Boundaries of DSL: External DSLs

- With external DSLs, the boundary is with general-purpose programming languages (GPL)
- Languages can have a domain focus but still be GPL.
- A good example of this is R, a language and platform for statistics;
  - It is very much targeted at statistics work, but has all the expressiveness of a GPL
  - Thus, despite its domain focus, I would not call it a DSL.
  - DSLs usually avoid the imperative control structures (conditions and loops), don't have variables, and can't define subroutines

Boundaries of DSL: External DSLs

- Another boundary with external DSLs is with serialized data structures.
- Is a list of property assignments (color= blue) in a config file a DSL?
- I think that here, the boundary condition is the language nature.
- A series of assignments lacks fluency, so it doesn't fit the criteria

Fragmentary and Stand-alone DSLs

- Fragmentary DSLs:
  - Little bits of DSL are used inside the host language code.
  - You can think of them as enhancing the host language with additional features
  - You can't really follow what the DSL is doing without understanding the host language

- For an external DSL, a good example of a fragmentary DSL is regular expressions
  - You don't have a whole file of regular expressions in a program, but you have little snippets interspersed with regular host code.

- Another example are SQL statements within a larger program

Why Use a DSL?

1. Improving Development Productivity
2. Communication with Domain Experts
3. Change in Execution Context
4. Alternative Computational Model

Improving Development Productivity

- The easier it is to read a lump of code, the easier it is to find mistakes, and the easier it is to modify the system.
- So, for the same reason that we encourage meaningful variable names, documentation, clear coding constructs — we should encourage DSL usage.
- People often underestimate the productivity impact of defects
  - Defects slow developers down by sucking up time in investigations and fixes
- The limited expressiveness of DSLs makes it harder to say wrong things and easier to see when you've made an error.

Communication with Domain Experts

- The hardest part of software projects, the most common source of project failure, is communication with the customers and users
- By providing a clear yet precise language to deal with domains, a DSL can help improve this communication
- When people talk about DSLs in this context, it's often along the lines of "Now we can get rid of programmers and have business people specify the rules themselves."
- I call this argument the “COBOL fallacy”

Communication with Domain Experts

- Despite the COBOL fallacy, DSLs can improve communication.
- It's not that domain experts will write the DSLs themselves; but they can read them and thus understand what the system is doing
- By being able to read DSL code, domain experts can spot mistakes
  - They can also talk more effectively to the programmers who do write the rules
- The biggest gain from using a DSL in this way comes when domain experts start reading

Change in Execution Context

- When talking about why we might want to express our state machine in XML, one strong reason was that the definition could be evaluated at runtime rather than compile time.

- This kind of reasoning -- shifting logic from compile time to runtime -- is a common driver for using a DSL

Alternative Computational Model

- Mainstream programming is pretty much all done using an imperative model of computation
- Imperative computation has become popular because it's relatively easy to understand and easy to apply to lots of problems.
- However, it isn't always the best choice.
- The state machine is a good example of this.
  - We can write imperative code to handle this kind of behavior.
  - But thinking of it as a state machine is often more helpful.

Problems with DSL

1. Language Cacophony
2. Cost of Building
3. Ghetto Language
4. Blinkered Abstraction

Language Cacophony

- The language cacophony problem:
  - Using many languages will be much more complicated than using a single one
- Even if they are relatively easy to learn, having many DSLs makes it harder to understand what's going on in a project.
- However, a project will always have complicated areas that are hard to learn:
  - Even if you don't have DSLs, you will typically have many abstractions in your codebase that you need to understand.
  - Even if you don't have to learn several DSLs, you still have to learn several libraries

Cost of Building

- A DSL may be a small incremental cost over its underlying library, but it's still a cost.
- There's still code to write, and above all to maintain.
- Thus, like any code, it has to pull its weight.
  - Not every library benefits from having a DSL wrapper over it.
  - If a command-query API does the job just fine, then there's no value in adding another API on top of it.
- Even if a DSL might help, sometimes it would just be too much effort to build and maintain for the marginal benefit.

Ghetto Language

- The ghetto language problem is a contrast to the language cacophony problem.
- Here, we have a company that's built a lot of its systems on an in-house language which is not used anywhere else.
- This makes it difficult for them to find new staff and to keep up with technological changes.
- Although you can use many of the DSL techniques for building general-purpose languages, I strongly urge you not to do so
- Building and maintaining a GPL condemns you to to life in a ghetto

Blinkered Abstraction

- The usefulness of a DSL is that it provides an abstraction that you can use to think about a subject area.
- However, any abstraction, be it a DSL or a model, always carries with it a danger — that of putting blinkers on your thinking.
- With a blinkered abstraction, you spend more effort on fitting the world into your abstraction than the other way around.
- You see this when you come across something that doesn't fit in with the abstraction — and you burn time trying to make it fit, instead of changing the abstraction to easily absorb the new behavior.

Chapter 7 – Alternative Computation Models

Imperative Computational Models

- Mainstream programming languages follow the imperative computational model.
- The imperative model defines computation through a sequence of steps: do this, do that, if (red) do the other.
- Conditionals and loops vary the steps, and steps can be grouped together into functions.
- Object-oriented languages add bundling together of data and process, as well as polymorphism — but are still grounded in an the imperative model.

Imperative Computational Models

- When we talk about ease of understanding, there are really two different kinds of understanding in play.
  - The first is understanding the intent of the program -- what are we trying to achieve with it.
  - The second form of understanding is of the implementation -- how the program works to satisfy the intent.

- Where the imperative approach doesn't always work so well is in the understanding of intent

- If the intent is a sequence of actions, fine; but often, our intentions aren't best expressed that way.
- In these cases, it is often worth considering a different model.

Alternative Computational Models

- Often, you run into situations where you need to state the consequences for different combinations of conditions.
- A small example of this might be scoring points to assess car insurance

Figure 7.1. A simple decision table for car insurance

has cell phone	Y	Y	N	N
has red car	Y	N	Y	N
points	7	3	2	0

conditions

consequences

```
public static int CalcPoints(Application a) {
  if ( a.HasCellPhone &&  a.HasRedCar) return 7;
  if ( a.HasCellPhone && !a.HasRedCar) return 3;
  if (!a.HasCellPhone &&  a.HasRedCar) return 2;
  if (!a.HasCellPhone && !a.HasRedCar) return 0;
  throw new ArgumentException("unreachable");
}
```

Alternative Computational Models

- There's similarity between the table and the code, but they're not quite the same.
- The imperative model forces the various if statements to be executed in a particular order, which isn't implied by the decision table

Alternative Computational Models

- Alternative computational models are also one of the compelling reasons for using a DSL.
- If your problem can be easily expressed using imperative code, then a regular programming language works just fine.
- The key benefits of a DSL — greater productivity and communication with domain experts — really kick in when you are using an alternative computational model.
- Domain experts often think about their problems in a non-imperative way, such as via a decision table.

A Few Alternative Models

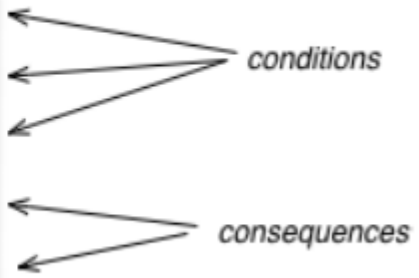
1. Decision Tables
2. Production Rule Systems
3. State Machine
4. Dependency Networks

Decision Tables

- The table consists of a number of rows of conditions, followed by a number of rows of consequences.

Figure 7.2. A decision table for handling orders

Premium Customer	X	X	Y	Y	N	N
Priority Order	Y	N	Y	N	Y	N
International Order	Y	Y	N	N	N	N
Fee	150	100	70	50	80	60
Alert Rep	Y	Y	Y	N	N	N



- The semantics are straightforward; in this case, you take an order and check it against the conditions
- If we have a premium customer with a domestic, priority order, there is a fee of $70 and we alert a representative to handle the order.

Production Rule Systems

- Production Rule System:
  - Rules: condition + consequent action

- Each rule can be specified individually in a style similar to a bunch of if-then statements in imperative code.

```
if
  passenger.frequentFlier
then
  passenger.priorityHandling = true;

if
  mileage > 25000
then
  passenger.frequentFlier = true;
```

Production Rule Systems

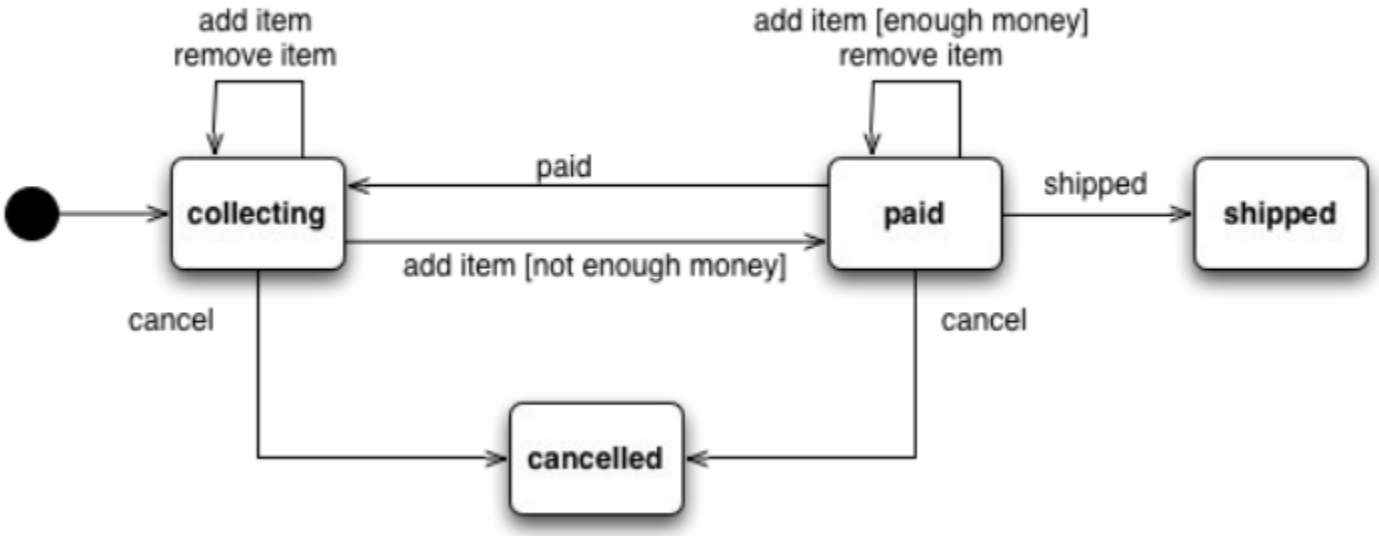
- We specify the rules in terms of their conditions and actions, but we leave it to the underlying system to execute and tie them together.

- In the example, there is a link between the rules in that if the second rule is true, that may affect whether the first rule should be fired.

State Machine

- A State Machine models the behavior of an object by dividing it into a set of states and triggering behavior with events.

Figure 7.3. A UML state machine diagram for an order

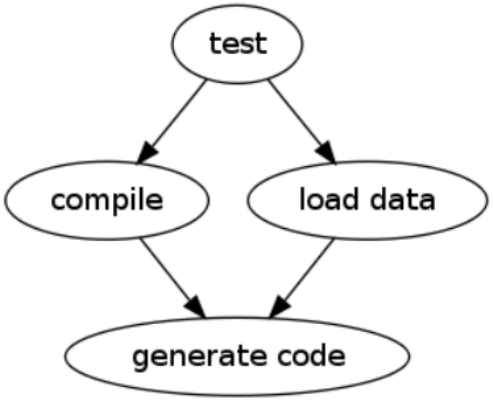


- The figure indicates that we can cancel an order in the collecting and paid states

Dependency Network

- One of the most familiar alternative models in the daily work of software developers is the Dependency Network.
- This model underpins build tools such as Make, Ant, and derivatives
- This model shows the tasks that need to be done and capture the prerequisites for each task.
- A Dependency Network is thus a good choice when you have computationally expensive tasks with dependencies between them.

Figure 7.4. A possible dependency network for building software

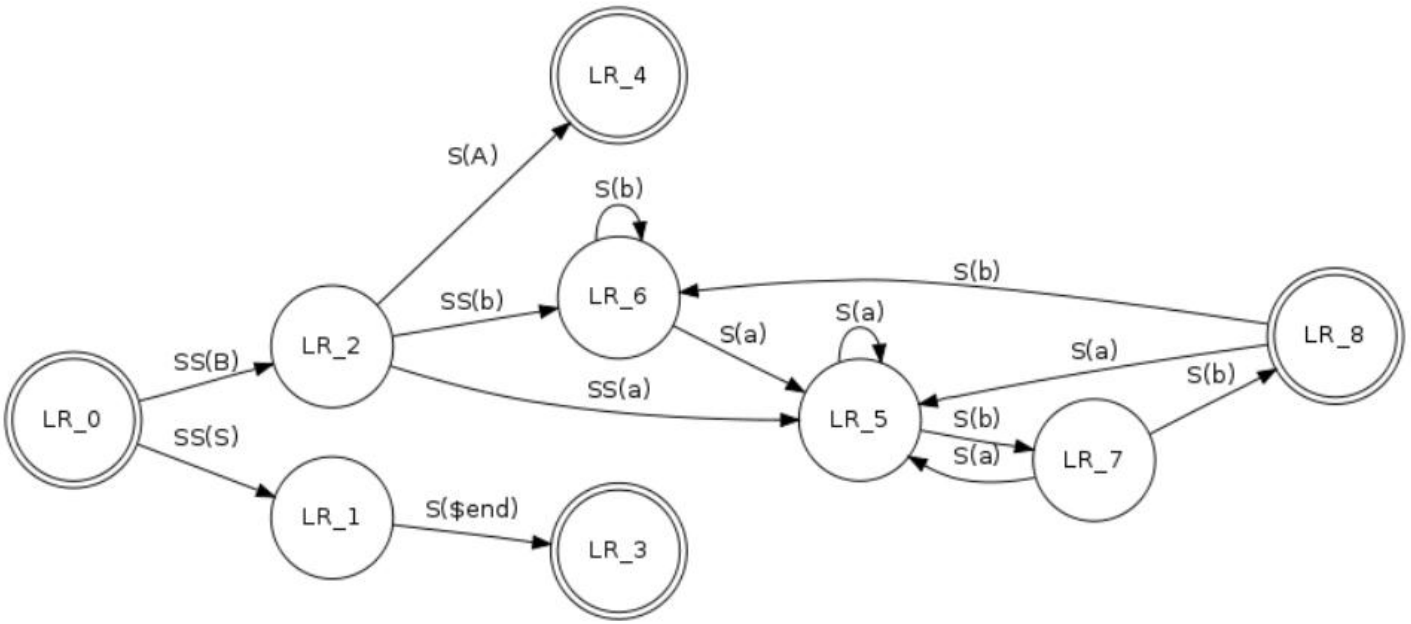


Chapter 10 – A Zoo of DSLs

Graphviz

- Library for producing graphical renderings of node-and-arc graphs

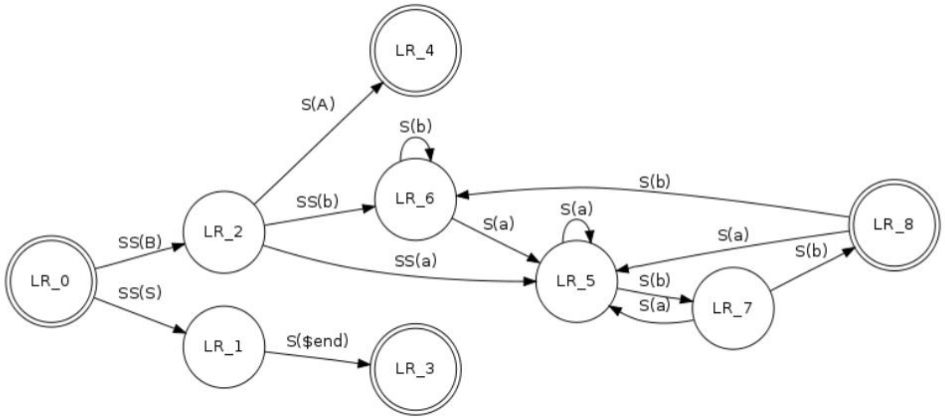
Figure 10.1. An example use of Graphviz



Graphviz

```
Graphvizdigraph finite_state_machine {
 rankdir=LR;
 size="8,5"
 node [ shape = doublecircle] ; LR_0 LR_3 LR_4 LR_8;
 node [ shape = circle] ;
 LR_0 -> LR_2 [ label = "SS(B) " ] ;
 LR_0 -> LR_1 [ label = "SS(S) " ] ;
 LR_1 -> LR_3 [ label = "S($end) " ] ;
 LR_2 -> LR_6 [ label = "SS(b) "
 LR_2 -> LR_5 [ label = "SS(a) "
 LR_2 -> LR_4 [ label = "S(A) " ]
 LR_5 -> LR_7 [ label = "S(b) " ]
 LR_5 -> LR_5 [ label = "S(a) " ]
 LR_6 -> LR_6 [ label = "S(b) " ]
 LR_6 -> LR_5 [ label = "S(a) " ]
 LR_7 -> LR_8 [ label = "S(b) " ]
 LR_7 -> LR_5 [ label = "S(a) " ]
 LR_8 -> LR_6 [ label = "S(b) " ] ;
 LR_8 -> LR_5 [ label = "S(a) " ] ;
}
```

Figure 10.1. An example use of Graphviz



Graphviz

- Graphviz uses a Semantic Model in the form of a C data structure.
- The Semantic Model is populated by a parser written in Yacc.
- The lexer is handwritten

JMock

- JMock is a Java library for Mock Objects.
- Mock objects are used in testing. You begin the test by declaring expectations, which are methods that an object expects will be called on it during the test.
- You then plug in the mock object to the actual object you are testing and stimulate that actual object.
- The mock object then reports if it received the correct method calls, thus supporting Behavior Verification

JMock

- Example:

```
mainframe.expects(once())
   .method("buy").with(eq(QUANTITY))
   .will(returnValue(TICKET));
```

- This says that, as part of a test, the mainframe object (which is a mock mainframe) expects the buy method to be called once on it.
- The parameter should be equal to the QUANTITY constant.
- When called, it will return the value in the TICKET constant.

- Mock expectations need to be written in with test code as a fragmentary DSL, so an internal DSL is a natural choice for them

CSS

- CSS is a style sheet language used to describe the presentation semantics (the look and formatting) of a document written in a markup language
- Example:

```
h1, h2 {
    color: #926C41;
    font-family: sans-serif;
  }
 b {
    color: #926C41;
 }
 *. sidebar {
    color: #928841;
    font-size: 80%;
    font-family: sans-serif;
}
```

This declarative nature introduces a good bit of complexity into figuring out what's going on.

In my example, an h2 element inside a sidebar div matches two different color rules.

CSS has a somewhat complicated specificity scheme to figure out which color will win in such situations.

CSS

- CSS is an excellent example of a DSL for many reasons.

- Primarily, it's a good example because most CSS programmers don't call themselves programmers, but web designers.
- CSS is thus a good example of a DSL that's not just read by domain experts, but also written by them.

- CSS is also a good example because of its declarative computational model, which is very different from imperative models.
- You simply declare matching rules for HTML element

Hibernate Query Language (HQL)

- Hibernate is a widely used object-relational mapping system which allows you to map Java classes onto the tables of a RDBMS.
- HQL provides the ability to write queries in a SQLish form in terms of Java classes that can be mapped to SQL queries against a real database.
- Example:

```
select person from Person person, Calendar calendar
where calendar.holidays[' national day' ] = person.birthDay
    and person.nationality.calendar = calendar
```

- This allows people to think in terms of Java classes rather than database tables, and also avoid dealing with the various annoying differences between different databases' SQL dialects.

Hibernate Query Language (HQL)

- The essence of HQL processing is to translate from an HQL query to a SQL query:
  - HQL input text is transformed into an HQL AST
  - The HQL AST is transformed into a SQL AST
  - A code generator generates SQL code from the SQL AST

- This path of transformations,

  input text → input AST → output AST → output text,

  is a common one with source-to-source transformation

- You can think of the SQL AST as the Semantic Model for this case.

XAML

- XAML files are XML files that can be used to define screen layouts.
- Example:

```
<Window x:Class="xamlExample.Hello"
  xmlns=" http: //schemas.microsoft.
com/winfx/2006/xaml/presentation"
  xmlns: x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Hello World">
  <WrapPanel>
    <Button Click='HowdyClicked' >Howdy! </Button>
    <Button>A second button</Button>
    <TextBox x: Name='_text1' >An editable text box</TextBox>
    <CheckBox>A check box</CheckBox>
    <Slider Width='75' Minimum='0' Maximum='100' Value='50' />
  </WrapPanel>
</Window>
```

XAML

- XAML is about how to organize relatively passive objects into a structure.
- Program behavior usually doesn't depend strongly on the details of how a screen is laid out.
- Indeed, one of XAML's strengths is that it encourages separating the screen layout from the code that drives the behavior of the screen.

XAML

- A XAML document logically defines a C# class, and indeed there is some code generation. The code is generated as a partial class, in this case xamlExample.Hello.
- I can add behavior to the screen by writing code in another partial class definition.
- Example:

```
public partial class Hello: Window {
  public Hello() {
     InitializeComponent();
    }
  private void HowdyClicked(object sender, RoutedEventArgs e) {
      _text1.Text = "Hello from C#";
  }
}
```

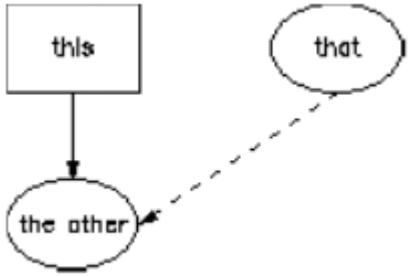
XAML

- This code allows me to wire behavior together. For any control defined in the XAML file, I can tie an event on that control to a handler method in the code (HowdyClicked).
- The code can also refer to controls by name in order to manipulate them (_text1).
- By using names like this, I can keep the references free of the structure of the UI layout, which allows me to change it without having to update the behavior code.
- DSLs to lay out graphical structures like this are quite common. Swiby uses a Ruby internal DSL to define screen layout

PIC

- Another DSLs for graphical layout, created in the very early days of Unix, when graphical screens were still unusual.
- It allows you to describe a diagram in a textual format and then process it to produce the image
- Example:

```
.PS
A: box "this"
move 0.75
B: ellipse "that"
move to A.s; down; move;
C: ellipse "the other"
arrow from A.s to C.n
arrow dashed from B.s to C.e
.PE
```



- A.s means the "south" point on shape A.

FIT

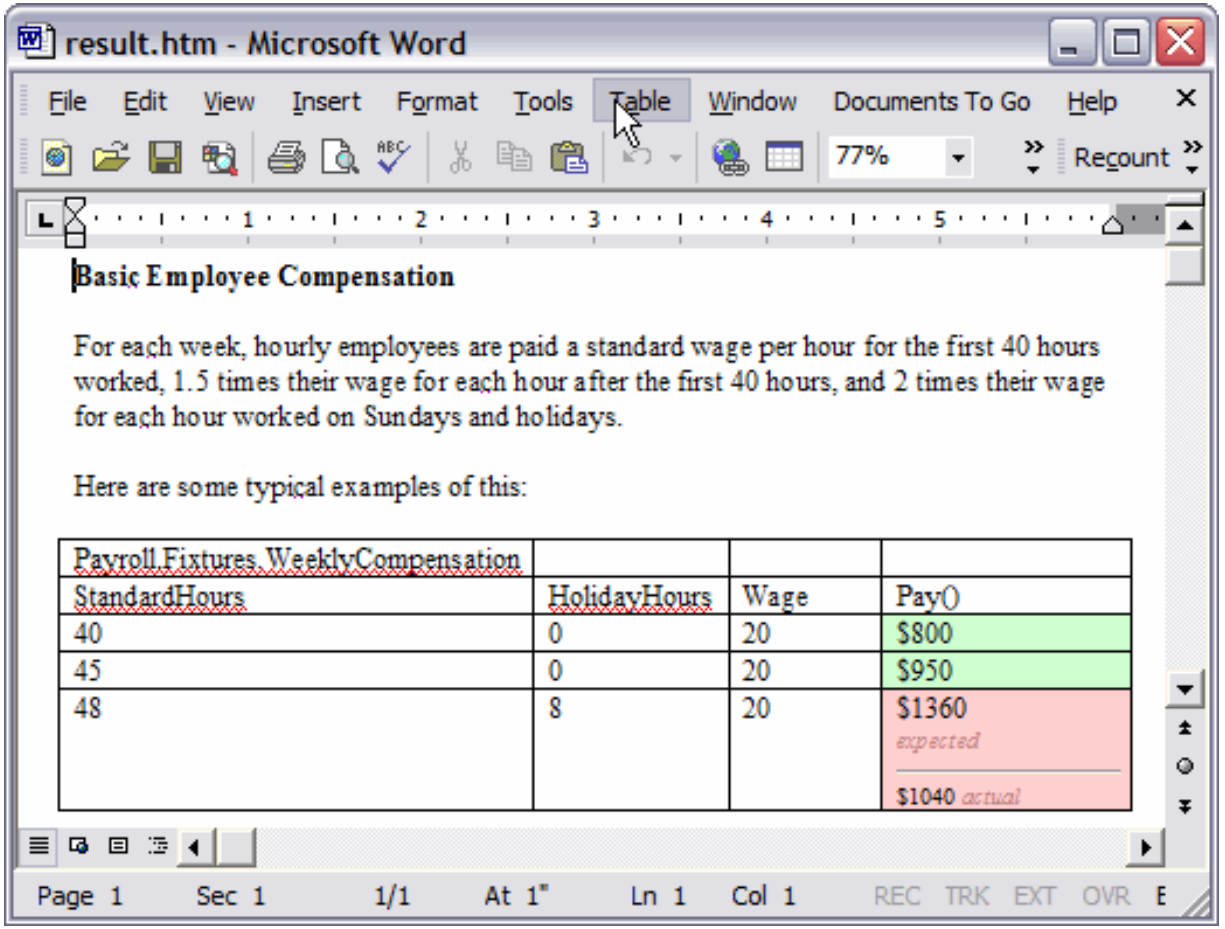
- FIT (Framework for Integrated Test) is a testing-oriented DSL, developed by Ward Cunningham
- In recent years, there's been quite a growth in interest for automated testing tools, with several DSLs created for organizing tests. Many of these have been influenced by FIT.
- Fit allows customers and testers to use tools like Microsoft Office to give examples of how a program should behave -- without being programmers themselves.
- Fit automatically checks those examples against the actual program, thus building a bridge between the business and software engineering worlds.

FIT

- Fit works by reading tables in HTML files, produced with a tool like Microsoft Word.
- Each table is interpreted by a "fixture" that programmers write.
- The fixture checks the examples in the table by running the actual program.

FIT Example

- The team is building a product to calculate employee pay. The team has worked together to create a Fit document that includes some examples of how hourly pay should be calculated.



FIT

- The tables contain the examples.
  - The first row tells Fit how to read the table.
  - The second row gives headers for the examples
  - The remaining rows provide examples.

- For example, the first example in the first table says, "If someone works for 40 standard hours and zero holiday hours, and is paid $20 per hour, then his or her total pay is $800.“

- Fit automatically checks the examples in the tables against the software.

FIT

- In this case, the software is reporting the correct result for the first two cases, so Fit has colored the table cells green.

- In the last case, the software is reporting the wrong result, so Fit has colored the table cell red.

FIT

- In order to make Fit work with this table, the team's programmers created a "fixture" that tells Fit how to talk to their software.
- This is what it looks like:

```
public class WeeklyCompensation : ColumnFixture
{
    public int StandardHours;
    public int HolidayHours;
    public Currency Wage;

    public Currency Pay()
    {
        WeeklyTimesheet timesheet = new WeeklyTimesheet(StandardHours, HolidayHours);
        return timesheet.CalculatePay(Wage);
    }
}
```

FIT

- The programmers used a ColumnFixture to map the columns in the table to variables and methods in the fixture.

- The first three columns, which provide information, correspond to variables in the fixture.
- The last column, which has the expected result, corresponds to the Pay() method in the fixture.
- To calculate the answer, the programmers used the WeeklyTimesheet class from their application

FIT

- Fit gives customers and programmers a way to communicate precisely about their software.
- Customers' concrete examples give programmers insight into the product being built.
- Programmers' work on fixtures and the software allow customers to experiment with different examples and gain insight into how the software really works.

Make

- In the early days of Unix, the Make tool provided a platform for structuring builds.
- The issue with builds is that many steps are expensive and don't need to be done every time, so a Dependency Network is a natural choice of programming model.
- A Make program consists of several targets linked through dependencies. Make is a familiar external DSL.
- Example:

```
edit: main.o kbd.o command.o display.o
            cc -o edit main.o kbd.o command.o
main.o: main.c defs.h
            cc -c main.c
kbd.o: kbd.c defs.h command.h
            cc -c kbd.c
command.o: command.c defs.h command.h
            cc -c command.c
```

Make

- The first line of this program says that edit depends on the other targets in the program;
- So, if any of them is not up-to-date, then, after building them, we must also build the edit target.
- A Dependency Network allows me to minimize build times to a bare minimum while ensuring that everything that needs to be built is actually built.

Rake

- An internal Ruby-based DSL for building files
- Example (Rakefile that builds this book):

```
docbook_out_dir = build_dir + "docbook/"
docbook_book = docbook_out_dir + " book.docbook"
desc "Generate Docbook"
task :docbook => [:docbook_files, docbook_book]
file docbook_book => [ :load] do
  require ' docbookTr'
  create_docbook
end
def create_docbook
  puts "creating docbook"
  mkdir_p docbook_out_dir
  File.open(docbook_book, ' w' ) do | output|
    File.open(' book. xml' ) do | input|
      root = REXML: : Document. new( input).root
      dt = SingleDocbookBookTransformer.new(output,
                    root, ServiceLocator.instance)
      dt.run
    end
  end
end
```

This line is is the Dependency Network, saying that the :docbook depends on the other two targets.