THE ARCHITECTURAL DESIGN OF "ILUTE", AN INTEGRATED DYNAMIC MICROSIMULATION MODELING FRAMEWORK

by

Paul A. Salvini

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science

Graduate Department of Civil Engineering

University of Toronto

Supervisor: Professor Eric J. Miller

© Copyright by Paul A. Salvini, 1998
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-34124-0
THE ARCHITECTURAL DESIGN OF “ILUTE”, AN INTEGRATED DYNAMIC MICROSIMULATION MODELING FRAMEWORK

Master of Applied Science

1998

Paul A. Salvini
Graduate Department of Civil Engineering
University of Toronto

Abstract

This thesis describes the object-oriented analysis and design of the framework for a large-scale, activity-based microsimulation model of urban land use and transportation. The goal of the Integrated Land Use, Transportation, Environment (ILUTE) model is to capture the interactions between transportation, land use, and environmental issues. The final framework is a flexible environment where individual sub-models can easily be incorporated and tested.
Table of Contents

1 Introduction .................................................................................................................. 1
   1.1 ILUTE Goal ............................................................................................................. 2
   1.2 Basic ILUTE Structure ......................................................................................... 3
   1.3 Simulation ............................................................................................................... 4
   1.4 Microsimulation ..................................................................................................... 7
2 Integrated Land Use – Transportation Modeling .......................................................... 10
   2.1 Temporal Dynamics ............................................................................................... 11
   2.2 Spatial Dynamics .................................................................................................... 15
   2.3 Chaotic Behaviour in Simulated and Real World Systems ..................................... 16
3 ILUTE Framework Design Principles .......................................................................... 19
   3.1 Openness and Extensibility .................................................................................... 19
   3.2 Data Persistence ..................................................................................................... 20
   3.3 Memory Constraints .............................................................................................. 22
   3.4 Performance and Computational Tractability ......................................................... 22
   3.5 Simulation Accuracy ............................................................................................... 23
   3.6 Accessibility and Usability ...................................................................................... 23
   3.7 Client-Server Configuration .................................................................................... 23
4 The Object-Oriented Paradigm ..................................................................................... 25
   4.1 Limitations of Procedural Programming ............................................................... 25
   4.2 Introduction to the Object Oriented Paradigm ......................................................... 26
   4.3 Classes ..................................................................................................................... 27
   4.4 Class Hierarchies .................................................................................................... 28
   4.5 Object Behaviour ................................................................................................... 30
   4.6 Object State ............................................................................................................ 31
   4.7 Representation of Pseudo-Entities .......................................................................... 32
   4.8 Sample Object Structure ....................................................................................... 33
5 ILUTE Architectural Overview: the Engines ................................................................. 34
   5.1 The Simulation Engine ........................................................................................... 34
   5.2 The Evolutionary Engine ....................................................................................... 35
   5.3 The Activity Engine ............................................................................................... 36
6 Development Phase I: Conceptualization ................................................................... 37
   6.1 Purpose of Conceptualization .................................................................................. 37
   6.2 System Prototypes ................................................................................................... 37
   6.3 Selecting a Programming Language ....................................................................... 38
   6.4 Selecting an Operating System ............................................................................. 38
7 Development Phase II: Analysis .................................................................................. 39
   7.1 Purpose of Analysis ............................................................................................... 39
7.2 Implementation Challenges ................................................. 39
7.3 Identifying Classes and Objects in the Problem Domain ............ 40
7.4 CRC Modeling (Classes, Responsibilities, Collaborators)........... 41
7.5 Identifying Classes and Objects in the Solution Domain ............ 43
7.6 Scenario Development ......................................................... 43
7.7 Identifying the Semantics of Classes and Objects .................... 44
7.8 Identifying the Relationships Between Objects and Classes ......... 44

8 Development Phase III: Design .................................................. 46
8.1 Purpose of Design .................................................................. 46
8.2 Software Design of the Simulation Engine ............................... 46
8.3 Software Design of the Evolutionary Engine ......................... 48
8.4 Software Design of the Activity Engine ................................. 49
8.5 Software Design of the Plug-In Models ................................. 49
8.6 ILUTE User Interface ............................................................ 50

9 Building the Initial Database: Population Synthesis .................... 52
9.1 The Need for Detailed Data .................................................. 52
9.2 Population Synthesis Techniques ......................................... 52
9.3 Synthesizing Other Objects ................................................ 53

10 Further Research ................................................................... 54

References ................................................................................... 56
Appendix A: Sample Sub-Model Dynamic Link Library (DLL) Code .... 57
Appendix B: Sample Items from the Initial Data Dictionary ............... 64
1 Introduction

This thesis documents the architecture of an extensible and open dynamic microsimulation framework for studying urban travel behaviour.

Chapter 1 provides an introduction to the ILUTE model and its goals. The remainder of the chapter looks at simulation and the benefits of microsimulation.

Chapter 2 focuses on some of the fundamental issues of microsimulation modeling as applied to travel behaviour. Given the importance of land use and location choice to the ILUTE model, the first two issues involve the treatment of time and space. Chapter 2 ends with a discussion of system stability and predictability.

Chapter 3 examines the functional requirements of the ILUTE model including architectural issues of the framework: extensibility, data persistence, memory use, performance, accuracy, and usability.

Chapter 4 introduces the object-oriented programming paradigm used in the development of the ILUTE model. This chapter describes the essential elements of object-oriented programming that help to reduce the complexity of the model.
Chapter 5 provides an introductory look at the three engines that form the backbone of the ILUTE model: the simulation engine, the evolutionary engine, and the activity engine. The roles and responsibilities of these engines are described along with the mechanisms that these engines use to collaborate with the ILUTE sub-models.

Chapters 6 through 8 cover the conceptualization, analysis, and design phases of the model. In chapter 6, the concept of the model is developed in detail. In chapter 7, the analysis of the model is performed and the initial data dictionary is created. In chapter 8, the design of the model is completed.

Chapter 9 looks at the synthesis procedure that is used to produce the initial ILUTE database for the base year.

Chapter 10 looks to the future of the ILUTE model and recommends areas where further research is required.

1.1 ILUTE Goal

The goal of the ILUTE project is to create an environment for travel demand modelers to experiment with dynamic microsimulation sub-models of urban land development, location choice processes, and travel behaviour. The
framework for the ILUTE project contains a sample set of sub-models of several short run and long run dynamic events.

1.2 Basic ILUTE Structure

Figure 1 provides an overview of the basic structure of the ILUTE model as described in Miller and Salvini [1998]. The "behavioural core" for this system consists of four inter-related components: land development: this models the evolution of the built environment, and includes the initial development of
previously "vacant" land and the redevelopment over time of existing land uses

1. location choice: this includes the location choices of households (for residential dwellings), firms (for commercial locations), and workers (for employment)

2. activity/travel: this involves predicting the trip-making behaviour of the population, ultimately expressed in terms of origin-destination flows by mode and by time of day

3. auto ownership: this models household auto ownership levels — an important determinant of household travel behaviour

1.3 Simulation

A simulation is a model that approximates the behaviour of a system. Simulation is a technique that is commonly used in a variety of engineering disciplines. For example, a wind tunnel can be used in conjunction with a model of an airplane wing to simulate the actual wing’s aerodynamic properties.
1.3.1 Computer-based Simulation

Computer-based simulation uses computer hardware and software to simulate or predict the behaviour of a physical system. The earliest computer systems, for example, were used in artillery calculations to simulate the trajectory of a projectile. For the purposes of this thesis, “simulation” refers to computer-based simulation.

1.3.2 Simulating to Forecast Travel Demand

In travel demand modeling, simulation is used to forecast travel demand. Some models focus on short run predictions of one or two years while others ambitiously project ten or twenty years into the future. Naturally, the short run predictions are easier to make and tend to be more accurate than the long run predictions. For example, next Wednesday’s travel demand is expected to be very similar to last Wednesday’s; in contrast, the travel demand in two decades might be considerably different.

1.3.3 Simulation as a Planning Tool

The more useful simulation models allow planners to make changes to the simulated system and determine the effect or impact of those changes. For example, a public transit planner might use a simulation model to compare the outcome of a 50% transit fare increase with the outcome of a fare freeze (the
"do nothing" scenario). The simulation would take into consideration the short term and long term elasticities of demand for transit and determine the short term and long term consequences on the ridership. In a different example, an urban transportation planner might examine the short term and long term effects of removing a major road (such as Toronto’s Gardiner Expressway) or constructing a new road (such as Highway 407).

For the urban planner, the development of a new community could be simulated and the long term effects of the development could be studied (without building the actual community). Planners for Don Mills, for example, hoped that the community would be self-contained and that residents would find work within the community. In reality, however, the houses were built before the jobs were created and most of the Don Mills residents worked elsewhere.

1.3.4 Simulation as a Research/Learning Tool

As a research or learning tool, simulations provide an opportunity to study challenging transportation problems such as the nature and extent of the interaction between transportation and particular kinds of land use. With a model, a variety of scenarios can be tested with complete control over the other parameters in the model.
1.3.5 Short Run vs. Long Run Simulations

One of the most interesting aspects of transportation modeling is the adaptive behaviour of the people in response to a change in their environment. For example, in response to a dramatic hike in public transit fares, riders might initially pay the new rate since they have no other means of getting to school or work. Over time, however, some would find an alternative such as carpooling, bicycling, or even moving closer to work.

There is no clear boundary between short run and long run changes. Closing a road would cause an immediate change in the use of the road (a short run change); but if the road remains closed, people will adapt to the change and the shape of the city will be changed.

Residential and business mobility and the development of the city are long run changes. Capturing these long run changes involves simulating the shape of the urban area over time.

1.4 Microsimulation

Microsimulation simulates the behaviour and state of individual objects in a system (such as persons, firms, households, and vehicles). Simulating individual objects means knowing when new objects enter the system and when existing objects leave. For persons, this implies simulating the processes
of birth and death as well as the processes of in-migration and out-migration.
For firms, this means simulating firms opening, closing, and relocating.

1.4.1 Benefits of Microsimulation

Conceptually, microsimulation is easier to understand than more aggregate methods since microsimulation simulates the behaviour of individual persons rather than the behaviour of a group of persons. Similarly, it is easier to write a program that characterizes the behaviour of a single object than a group of objects. Thus, an important benefit of microsimulation is that it simplifies the task of programming.

Another important benefit of microsimulation is the elimination of aggregation bias. Traditional aggregate transportation models are inherently biased, a problem that is eliminated by working at the level of the individual. A discussion of the problems associated with using representative samples instead of the entire population is found in Mackett [1985].

Another advantage of microsimulation is that it makes it possible to simulate emergent behaviours. In a microsimulation model, the individual actors within the model are allowed to interact with the other actors in the system. This interaction has the potential to generate more complex emergent behaviours.
that would be very difficult to program into a more aggregate simulation model.
2 Integrated Land Use – Transportation Modeling

This chapter examines three fundamental issues in the development of an integrated land use and transportation microsimulation model: the treatment of time and space, and the issue of stability.

The first section of this chapter looks at the treatment of time and the problem of dealing with events that occur on significantly different temporal scales. The technique for linking the short run dynamics of the activity model and the long run dynamics of the evolutionary model (which includes location choice) is also discussed in this section. In the second section, the parallel issue of how to handle the spatial component of the model is discussed. The third section provides a brief discussion of stability as it applies to the development of microsimulation models.

The issues of time and space are fundamental to several components of the ILUTE model. In the activity-based travel demand component of the model, trips occur in both time and space. A trip from home to work involves traveling from one location to another and takes a certain amount of time. The residential mobility component of the model, on the other hand, deals with the decision of a person or household to relocate. Clearly, these two types of events occur on very different time scales. In the short run, trips are made
between given locations in space during certain times; in the long run, persons and households move from one residence to another.

2.1 Temporal Dynamics

Temporal dynamics is the time element of the dynamic microsimulation model. To develop a credible microsimulation model of urban travel behaviour, it is important to pick an appropriate time interval (or time increment) for each iteration of the simulation. This time interval is known as the temporal scale. Computationally, the smaller the temporal scale, the longer the simulation will take to run; however, a temporal scale that is too large will miss important events that occur between intervals.

2.1.1 Short Run, Medium Run and Long Run Dynamics

For example, a time increment of one second (or less) might be required to simulate the velocity and acceleration of a vehicle and its emissions. An increment of one minute would clearly be too large because it misses the changes in velocity that occur during that minute. A time increment of one second might also be appropriate for simulating a traffic signal control system but simulating every nanosecond of traffic flow in a network would likely be wasteful. Similarly, a time increment of one minute would be too large to capture any meaningful change in cycle splits.
Some events do not need to be simulated every second. The decision to engage in a trip-making activity, for example, might require a simulation time of fifteen minutes. There is little need to evaluate trip-making decisions every second, but clearly activities can change throughout a day. A time increment of one day, for example, would be too large to capture the spontaneous decision to go shopping after work.

Some decisions such as moving residences (and the associated location choice decision), getting married and buying a car have an even larger temporal scale. These decisions are usually not made regularly throughout a single day. It might be sufficient to evaluate these decisions on a yearly basis. Even if the result of the decision is to not make a change, it is important to simulate the process of making the decision.

2.1.2 Handling Multiple Temporal Scales in ILUTE

Different events can occur at different temporal scales. The ILUTE model supports multiple temporal scales within its framework. While the integration of multiple temporal scales complicates the design of the ILUTE model, it is a critical architectural decision to achieve accurate results with minimal computational cost.
2.1.3 Refining the Resolution of the Temporal Scale

While simulation requires that each event be assigned a suitable temporal scale, sub-models can still refine the scale as long as interaction with other sub-models is not required. The Canadian real estate market, for example, is known to be seasonal and certain months have stronger performance than other months. The residential mobility sub-model, which is responsible for matching movers with available housing stock, could not accurately model market forces on a yearly scale. Not all of the houses that are transacted in a given year are on the market simultaneously – to that end, simulating a typical day in the real estate market would seem impossible with a discrete temporal simulation step of one year. Fortunately, the resolution of the temporal scale can be refined within the sub-model so that monthly or weekly market forces can be accommodated without incurring the computational cost of triggering response events and iterating the simulation of the main model.

2.1.4 Interactions Between Long Run and Short Run Events

A major consideration in the ILUTE model is the interactions that occur between short run and long run events. Since one of the goals of the ILUTE model is to develop an integrated land-use and transportation model, the results of the low-level activity model must feed into the next iteration of the high-level evolutionary and location choice model. Additionally, the results of
the high-level evolutionary and location choice model must feed back into the low-level activity model.

The interaction between these two levels is fundamental to the design of the ILUTE model. Since the activity and evolutionary models share a common database, they can access the results of each other’s simulation. This feedback loop is what allows the ILUTE model to ensure synchronization between the activity and location choice models.

2.1.5 Sampling a Day of Activity Per Year

Transportation characteristics change over time. Traffic patterns vary by day of week, by week of year, and from year to year. For example, Mondays and Fridays are often excluded in traffic studies when a “typical day” is desired. Similarly, summer months often have lower volumes as a result of the decrease in school-related trips.

The activity-based portion of the ILUTE model is based on the simulation of a single “typical” day in the year. The justification for the decision to choose one day rather than a set of days is relatively straightforward. First, simulations do not suffer from the same variability found in the real world – there is no need to simulate many days to get a large enough sample to represent a “typical” day. Second, the activity model can only affect the
dynamics of the simulated area once in a given year (since the long run time scale is set to one year). Simulating multiple days, therefore, provides little numerical benefit, yet it carries a high computational cost.

2.2 Spatial Dynamics

Spatial dynamics reflect the changes in location that occur as the simulation runs. In the short run, the location of each person and vehicle in the system is important (in the short run, these locations change). These persons engage in activities that take place at different locations in the study area. In the longer run, persons and firms relocate over time. Simulating these longer run dynamics allows control over the evolution of the growth and structure of the study area over time.

The challenge in microsimulation is to determine an appropriate level of spatial detail for the model. Spatial issues occur in many portions of the model. In the land use and location choice components, enough detail must be available to describe the land and its permitted uses. When dealing with housing and building stock, enough detail is needed to know the floor space of the building and its "footprint" on the lot. On the network side, the road and transit networks must be known in enough detail as to derive reasonably accurate travel times.
For ILUTE, the choice of spatial scale or scales has not yet been finalized. For the initial ILUTE model, most spatial data is based on census tracts and the road and transit networks are initially expected to contain most roads and routes respectively. To avoid the computation cost of calculating flows on local streets, the concept of a zonal centroid will be used by ILUTE. Walking trips in the ILUTE network will be based on straight-line distances.

2.3 Chaotic Behaviour in Simulated and Real World Systems

The goal of a simulation is to emulate or approximate the behaviour of a given system. To do this, the simulation must behave in a manner similar to the behaviour of the actual system. To develop a simulation model, it is important to determine if the system being simulated is stable or unstable.

Stable systems are those where small changes in input lead to small changes in output. A marble placed inside a bowl is a good example of a stable system. Even if the marble is moved, it will quickly return to its natural position at the bottom of the bowl. Balancing a marble on the top of a soccer ball, however, would result in an unstable system — even if the marble could be balanced, a small change in the system would result in a significant change in the position of the marble.
Extreme examples of unstable systems are described as “chaotic”. Chaotic systems are difficult to simulate because their behaviour is seemingly unpredictable. If the world is chaotic, there might be a number of equally possible yet radically different outcomes to a given change. Even worse, a small event (one thought to be too small to be modeled) may actually determine the outcome of the state of the system. Simulating all such seemingly inconsequential events is clearly not possible.

Compounding these issues is the difficulty of determining if a given system is stable. While some argue that complex systems are more susceptible to catastrophic failure than simple systems, others claim the “law of large numbers” suggests that although a very complex system may have a number of unstable sub-systems, the system itself is likely to be stable.

Historically, there are many examples of hard-to-predict events that had a dramatic impact on urban development. Stock market crashes, the election of new leaders, the development of new technologies, wars, and natural disasters are all examples of such events.

The ILUTE model assumes that the system under simulation is relatively stable over time; this assumption is consistent with current planning practices. Historical data from censuses and other sources will be used to validate,
verify, and calibrate the model in order to ensure that the model is providing reasonable predictions of land use and travel behaviour.
3 ILUTE Framework Design Principles

This chapter discusses the principles that guided the design of the ILUTE simulation framework. Each section describes a software requirement that was developed at the time of conceptualization. These requirements were important considerations during the analysis and design of the ILUTE model.

3.1 Openness and Extensibility

The ILUTE model provides a framework where independent researchers can build sub-models to distribute to other ILUTE users. As a result it is necessary to have a mechanism to allow sub-models to be plugged in to the ILUTE framework. The end result is a powerful and flexible framework that allows new model types and sub-models to be added without having to change or recompile the ILUTE source code.

Once a sub-model is added, the ILUTE framework automatically recognizes the sub-model and gives the user the option to include its components into the simulation. The ILUTE user interface automatically updates to reflect the choice in the new model and to allow the user to modify the sub-model's individual parameters. Naturally, it is not possible to predict which sub-models will be written or how many different sub-models might be developed.
for a given simulation variable. For example, several researchers might write a sub-model to simulate births. The user must choose which birth sub-model to run because an error would result if two birth sub-models were included in the same simulation run.

3.2 Data Persistence

In a simulation framework, it is desirable to be able to create branches from a given simulation run. Each branch represents a change in one of the inputs to the system such as a change in policy or a change in transportation infrastructure. The following chart illustrates how a branch can be used to compare the outcomes of two policy decisions. In this example, two policy options were considered in generating the outcomes for year $n+2$. Each branch of the tree can be computed for year $n+3$ and the resulting databases can be compared to see the impact of choosing policy X vs. policy Y.
Furthermore, it was recognized that users might not know in advance of the simulation run that a branch would need to be created. Therefore, two significant constraints had to be included in the design. First, the database had to have the same format every year regardless of whether it stored base data or simulated data. Second, the database needed to contain enough information from the output of the simulation to be able to allow branching in successive years without incurring the computational penalty of re-simulating the earlier years.

While these constraints added to the complexity of the project, they resulted in a framework flexible enough to allow users to experiment with a variety of exogenous factors as well as manipulate any of the endogenous parameters in the sub-models. Perhaps most exciting result for researchers is the ability to
test the difference in outcomes between different sub-models of a given sub-model type.

3.3 Memory Constraints

The ILUTE model is designed to run on systems with less than one gigabyte of main memory (a reasonable configuration at the time of the development of the model). As a result, it is not possible to keep all objects active in memory simultaneously. The consequence of these constraints is that the information must be kept on disk and only instantiated "as required" for the purpose of updating the system.

3.4 Performance and Computational Tractability

Performance is a critical consideration in the development of most microsimulation systems. Because of memory constraints, the objects in the ILUTE model must be kept on disk. Since disk access speeds are considerably lower than memory access speeds, it is important to optimize the retrieval of objects from the ILUTE database into main memory to avoid unnecessary database transactions. In addition, the sequencing of updates and the ordering of event processing are also important design considerations.
3.5 Simulation Accuracy

The goal of the ILUTE project is not to predict future events with great accuracy; rather, it is to allow users to experiment with the interaction between land-use and transportation issues. As with all models, determining the required accuracy of the inputs to the model is difficult, as the sensitivity of the inputs is rarely known.

3.6 Accessibility and Usability

As with all computer software developed for users, it is important to make the software as intuitive as possible. The ILUTE project, complete with its sub-models, is an excellent tool for experimenting with policy decisions and observing the changes in simulation outcomes.

The amount of data returned by a dynamic simulation is enormous as the information is stored in disaggregate form. It is important to provide tools for querying the database to generate more aggregate reports.

3.7 Client-Server Configuration

To handle the reporting requirements, it was important for the database to support queries from the user. The standard Structured Query Language
(SQL) is used by ILUTE as the mechanism for user interaction with the database.

Separation of the client from the server (known as a client-server) model allows remote clients to access a centrally located ILUTE database. In this configuration, the client application sends queries (SQL statements) to the ILUTE server. The database engine running on the ILUTE server processes the queries and sends the results back to the client application.

The large size of the ILUTE database makes it essential for the server to execute the queries locally and return only the results to the client machine. The client-server model also makes it possible to allow remote researchers to access the ILUTE database over the Internet.
4 The Object-Oriented Paradigm

This chapter introduces the object-oriented programming paradigm that underlies the development of the ILUTE model. The chapter examines some of the limitations of the procedural programming paradigm and hence the motivation for object-based techniques. The second section of this chapter gives a brief overview of the object-oriented programming paradigm and describes some of its benefits. Section three introduces the “class” concept and explains the relationship between classes and objects. Section four examines the topic of classification and shows how objects can be classified based on their characteristics and/or behaviours. Sections five and six describe the concepts of object state and object behaviour and their application in object-oriented programming. Section seven looks at the issue of representing objects that are not found in the real world. This chapter concludes with section eight, which examines a real object from the ILUTE system that illustrates some of the concepts from the earlier sections.

4.1 Limitations of Procedural Programming

In the procedural approach, programming consists of breaking a problem down into a set of steps (or procedures) which can be further decomposed into a set of sub-procedures. This approach is also known as “top-down stepwise
refinement”. While procedural systems are well suited to many scientific applications, they are poorly suited to systems that require decomposition based on the objects found in the problem domain. In procedural programming, the design of the program is centered on the behaviour of the program. Consequently, procedural programs are robust only when the procedures remain constant and the input data is changed.

4.2 Introduction to the Object Oriented Paradigm

Object-oriented systems, on the other hand, concern themselves with a world comprised of objects that interact (collaborate) to perform complex behaviours. Providing the objects in the system remain the same, the resulting system remains stable even when the function of the program changes.

4.2.1 Inherent Stability of Objects

A dynamic microsimulation such as ILUTE is a complex and difficult system to implement using any paradigm. However, it is well suited to the object-oriented methods of analysis and design. In the simulated area, there are people, houses, firms, cars, and roads all interacting in interesting and complex ways. While these interactions may change over time (as well as the specific individuals involved in the interactions), the types of actors in the system generally do not change. The ability to represent real world objects in an
intuitive way is one of the main attractions of the object-oriented paradigm. All information and behaviour about a given object is encapsulated within the object’s definition.

4.3 Classes

Rather than program the behaviour of individual objects, the concept of classes allows the creation a class of objects from which given objects will be created. This classification of objects creates generalized descriptions of common behaviours and attributes for all objects within the given class.

4.3.1 Objects as Instantiations of Classes

For example, King Street between Bathurst and Portland is an instance of the Road Link class. All Road Links have common properties such as length, number of lanes in each direction, parking capacity, and speed limit. Every instance of the Road Link class has values corresponding to these properties: King Street, for example, might have a length of 150 m, 2 lanes in each direction, parking for 24 cars and a speed limit of 50 km/h.
4.4 Class Hierarchies

Sometimes it is useful to arrange the classes into hierarchies. A hierarchy allows information common to several classes to be placed in one location.

The following chart shows a simple class hierarchy:

```
  Vehicles
    Automobiles  Passenger Vehicles  Trucks
        School Buses
```

While most of the information pertaining to vehicles will be stored in the "Vehicles" class, some information is specific to one of the vehicle types. For example, trucks might store additional information on axle weights or customs status (bonded carrier, etc.) that is not applicable to other vehicle types. Similarly, school buses have morning and afternoon routes that do not apply to other passenger vehicle types.

4.4.1 Classification

Many fields of science have found the concept of classification useful. The world of living things, for example, is currently broken down into five subclasses called kingdoms: plants, fungi, animals, protoctista, and bacteria. Each of these kingdoms is further broken down into a number of subclasses.
called phyla. These phyla are further broken down and the final hierarchy is eight levels deep. The following chart shows a very small subset of these classes to illustrate the classification. Each vertical level in the chart represents a further breakdown of the parent class.

Once a class hierarchy is established, the benefit is clear; the hierarchy itself describes the common features in the different classes. For example, the above hierarchy shows that wolves and dogs are carnivorous whereas honeybees are not.
4.4.2 Difficulty of Classification

While classification has some obvious benefits, it is not always easy to create a hierarchy that works for all objects. The animal kingdom hierarchy, for example, has been debated since its creation and has already been changed several times. Classifying objects can be extremely difficult and there may be several valid classifications for any group of objects.

4.5 Object Behaviour

Once a set of classes has been determined and a hierarchy established, the task of programming the behaviour of the objects of each class begins. This process involves understanding what each object needs to know (its state) and do (its behaviour). Object-oriented programming involves creating abstractions for each object and giving behaviours to these abstractions. Behaviour is what makes real world objects unique and programming these behaviours is the fundamental task of object-oriented programming.

The most important and complex objects in the ILUTE system are the person objects. Each person object exhibits a certain behaviour that is unique to his or her situation. The person class is responsible for encapsulating all of the behaviours of a person object. These behaviours are stored within the class definition in special functions known as member functions.
In the ILUTE project, the person object engages in trip-making behaviours and non trip-making behaviours. Trip-making behaviours include trips to and from work, school, and shopping. Non trip-making behaviours include residential mobility decisions, location choice, marriage, birth, and death. Each of these behaviours is encapsulated within the person class.

4.6 Object State

In addition to behaviour, objects also have a state. An object's state stores information about the object. For example, a person object would have state information such as a date of birth, education level, marital status, etc. In programming terms, an object's state is stored in a variable that is also encapsulated within the object's class. Some variables are simple, such as a Boolean to represent marital status or an integer to represent the number of children. Other variables, however, are quite complex and may be objects themselves.

4.6.1 The Object Hierarchy ("Part of" Relationship)

The state of an object can also be stored in a hierarchy. This hierarchy is known as the "part of" or "object hierarchy" and represents the decomposition of an object into its physical parts and sub-parts. This
hierarchy allows the information that is stored for a given person to be organized.

4.7 Representation of Pseudo-Entities

Some real world concepts do not represent physical entities. A household, for example, is a term that is given to a set of persons living in the same dwelling unit. These households may be family households, non-family households (unrelated roommates), or even single-person households.

While it would simplify the design of the model to ignore the concept of the household, the importance of the household is too fundamental to ignore. It is clear that certain behaviours are household behaviours rather than individual behaviours. A family moving to a new residence, for example, affects the entire household.

Subtly buried within the concept of the household is the more abstract issue of decision-making. While moving a family to a new house affects the entire household, the decision to move is probably made by the parents rather than the children. In a non-family household, however, the decision might be made unilaterally be one of the residents. Independent of the household, then, is the concept of the decision-making unit. The decision-making unit changes depending on the decision under consideration.
Pseudo-Entities are represented within the ILUTE database by indexing the aggregated type to the relevant container object. For example, each person object contains a link that connects a person to his or her household. This link is known as the person's household identifier and is represented as \textit{household ID} in the person table. Using this index technique, the ILUTE software can then determine that person 46 and person 48 both belong to household 16.

4.8 Sample Object Structure

The following subset of the Person class will illustrate the conceptual clarity of the object-oriented paradigm as it applied to dynamic microsimulation.

<table>
<thead>
<tr>
<th>Class Name:</th>
<th>CPerson</th>
</tr>
</thead>
</table>
| Behaviours: | □ give birth  
               □ die  
               □ get married  
               □ get divorced  
               □ find a job  
               □ leave current job  
               □ leave current household  
               □ find a household  
               □ purchase a vehicle  
               □ move to a new residence |
| Parts (state): | □ person ID  
                   □ household ID  
                   □ marital status  
                   □ spouse ID  
                   □ date of birth  
                   □ job ID |
5  ILUTE Architectural Overview: the Engines

Chapter five gives a basic introduction to the roles and responsibilities of the three engines that form the foundation of the ILUTE framework: the simulation engine, the activity engine, and the evolutionary engine. Together, these engines are responsible for loading plug-in sub-models, setting up the simulation parameters, determining simulation output, and running the simulation. This chapter serves as background for chapters six through eight, which cover the first three phases of the object-oriented project lifecycle: conceptualization, analysis, and design. In chapter eight, the design of each of the three engines is given in detail.

5.1 The Simulation Engine

The simulation engine is responsible for managing the simulation. Once a simulation run has been configured, the simulation engine executes the simulation by using the services of the other two main ILUTE engines: the evolutionary engine and the activity engine. The simulation engine reports its progress to the user and notifies the user when the simulation has completed. The result of the simulation is a series of new databases for the target years specified in the simulation control panel.
5.2 The Evolutionary Engine

The purpose of the evolutionary engine is to evolve the study area by one time increment. The simulation engine calls the evolutionary engine each time it needs to update the study area.

The evolutionary engine updates the study area by executing a number of individual microsimulation sub-models that update the database. These sub-models use a variety of data sources during their execution including (but not limited to) the current and previous state of the system, results from the activity engine, and external data sources such a table predicting annual fertility rates.

The evolutionary engine is an object that is created by the simulation engine. The input to the evolutionary engine is the database corresponding to the current simulation year. The flexibility of the evolutionary engine allows this database to represent a base year (i.e. produced directly by a data synthesis procedure) or a year previously created by another instance of the evolutionary engine. The output from the evolutionary engine is a database one time step greater than the input source (in the initial ILUTE model, this time step is one year).
5.3 The Activity Engine

The activity engine is responsible for dynamically microsimulating events or activities that would occur on a given day in the simulated year. The activity engine uses the system database as input and dynamically simulates the activities that occur within the system.

The output of the activity engine feeds back into the next iteration of the evolutionary engine. The residential mobility component, for example, uses travel times to calculate the household’s “utility”.

36
6 Development Phase I: Conceptualization

This chapter is the first of three chapters describing the development phases of the ILUTE model. This chapter describes the conceptualization phase, its purpose, and its results.

6.1 Purpose of Conceptualization

Software development projects usually begin with a conceptualization phase to establish the system’s core requirements and to determine the project’s major technical and non-technical risks. Interested readers are directed to Booch [1994, 1996] for a detailed description of the object-oriented software development process.

A major technical risk in the ILUTE project, for example, was the ability to process micro-level data in a reasonable amount of time. The requirements for data persistence made it necessary to store and update massive amounts of disk-based data representing millions of individual objects.

6.2 System Prototypes

During this phase of the project, a number of executable prototypes were created to assess the viability of the technology for this task. These prototypes
helped shape an understanding of the limits of current technology and gave insight to the need for an optimized database. For the data persistence example, a prototype database was built to measure the transaction performance of the database.

6.3 Selecting a Programming Language

The conceptualization phase also involves the selection of the primary language for the development of the system. For ILUTE, there were some initial questions as to whether it was best to write the project in C and use traditional procedural development techniques or whether C++ should be used along with the object-oriented programming methodology. Other languages given brief consideration included Visual Basic (which was considered too slow for writing the engine code), Ada, and Smalltalk. Ultimately, C++ was chosen for its excellent performance, support for the object-oriented paradigm, and universal availability.

6.4 Selecting an Operating System

Another technical consideration in the conceptualization of ILUTE was the choice of operating system. Windows NT was selected because of the performance of its file system (NTFS) and because it provided a commonly available and accessible platform for most of the project researchers.
7 Development Phase II: Analysis

This chapter details the analysis phase of object-oriented software development for the ILUTE project.

7.1 Purpose of Analysis

The purpose of the analysis phase was to develop a model of the behaviour of the system. At a macroscopic level, the concept of the ILUTE project was simply to provide a framework for various dynamic microsimulation models that could be tested in the domain of a computer-based laboratory. This laboratory would be used in the development of a dynamic microsimulation model of travel behaviour and vehicle emissions.

7.2 Implementation Challenges

A number of challenging implementation issues remained beyond the high-level view. The experimental nature of the ILUTE framework makes it difficult to create an exact specification of the system's intended purpose or behaviour. As a result, the ILUTE project needs to proceed as an evolving prototype with design decisions implemented as the project evolved. Fortunately, the object-oriented approach is well suited to this kind of iterative and incremental development approach.
Project team dynamics and the number of researchers involved also complicates the development of the project. Each researcher provides a slightly different view of the purpose, scope, and desired behaviour of the system. Ultimately the team approach benefits the project because it helps to surface and resolve many issues prior to the development of the initial project design.

The scope of the ILUTE project was narrowed in a series of weekly meetings lasting approximately six months. Although considerable progress was made, a number of fundamental issues remained unresolved. Early prototypes for various components of the project helped identify and resolve many of these issues.

The analysis phase concluded with a reasonable enough description of the behaviour of the evolutionary engine that the initial design work could begin.

7.3 Identifying Classes and Objects in the Problem Domain

Much of the appeal of object-oriented programming is that it provides a conceptually appealing connection between real-world elements and the objects that simulate them. One of the early tasks in developing the ILUTE model was identifying which objects were to exist in the abstraction. These
objects are known in object-oriented design parlance as the “key abstractions” found in the problem domain.

Dynamic microsimulation also has a conceptually appealing connection to the real world because instead of simulating group behaviour, it simulates the behaviour of individual actors within the system. Perhaps this is why microsimulation proponents find the object-oriented paradigm so appealing.

There are a number of real world objects that can be found as key abstractions in the ILUTE problem domain. The first phase in the object-oriented micro development process is to identify these key abstractions. This phase can be characterized as a process of discovery whereby the boundaries of the problem at hand are discovered. In this phase, it is roughly determined which objects are and are not of interest. The objects that are found are documented and placed into a data dictionary. This data dictionary serves as a list of the classes and objects that will be ultimately need to be developed within the architecture of the system.

7.4 CRC Modeling (Classes, Responsibilities, Collaborators)

The process of discovery in the ILUTE project was aided by the use of CRC cards (Classes, Responsibilities, and Collaborators).
The first CRC modeling exercise identified a number of classes and objects found in the problem domain. At the end of the exercise, an unordered list of classes and objects was created including some of the following candidate classes and objects:

- person
- household
- activity schedule
- spouse
- dwelling unit
- vehicle
- firm
- building
- job
- school
- neighbourhood
- land use
- location
- zone
- road network
- road node
- road link
- agenda
- transit network
- transit node
- transit link
- transit route

A more complete list of the initial classes and their responsibilities is given in the initial data dictionary in Appendix B.
7.5 Identifying Classes and Objects in the Solution Domain

While the conceptual clarity of an object-oriented system holds great appeal to an outside observer, the reality is that many of the abstractions required in the program are not found in the language of the problem domain. Instead, these objects exist only in the language of the solution domain and are largely hidden from the system's users.

Since these classes and objects are hidden from view, they are more difficult to identify as necessary parts of the system; however, their discovery is essential to the creation of any complex system.

7.6 Scenario Development

Scenarios are often used in object-oriented analysis sessions to help discover new abstractions required by the system. Each scenario described by the object-oriented modeling team represents a possible path of behaviour that must ultimately be handled by the system; for example, a person who lives alone decides to move to a new home that is closer to his or her new job. By examining this scenario, it is clear that objects representing the concepts of a person, a household (to know the person lives alone), a job, a dwelling unit, and a location are needed.
7.7 Identifying the Semantics of Classes and Objects

Once a candidate set of classes and objects has been established, the next phase is to determine the meaning of each of these abstractions in terms of what they know (their state information) and what they do (their behaviours). During analysis, the goal of this step is to identify the roles and responsibilities of each of the classes (real world and otherwise) found in the initial discovery stage.

The birth sub-model, for example, is an example of an object that was discovered early in the identification phase. The birth sub-model is responsible for generating all of the births that occur in the study area in a given year. In order to perform its function, it must have some mechanism to determine factors such as the number of births, to whom they occur, and the gender of the newborn child or children.

7.8 Identifying the Relationships Between Objects and Classes

Two important relationships supported by object-oriented programming languages are links and aggregation. Five important class relationships are association, inheritance, aggregation, using, and instantiation (through template classes).

44
Once the various objects and classes have been identified and their roles and responsibilities established, the relationships between these entities must be determined.

For example, the Road Network, Road Node, and Road Link classes have the relationship that a Road Network is a collection of Road Nodes and Road Links. This containment suggests an aggregation relationship. Similarly, the Transit Link class might be inherited from the Road Link class since the only difference is an additional information about the transit stop located on the link. Consequently, this is an example of an inheritance relationship.

The identification of the relationships between the various objects and classes in the system helps to significantly simplify the structure of the system. As a result, the design of the system becomes easier and the complexity of the overall model is reduced.
8 Development Phase III: Design

This chapter outlines the design of the various components of the ILUTE model. The chapter begins with a detailed design of the three engines introduced in chapter five. In section five of this chapter, the design of the plug-in models and their interface is discussed. In section six, the design of the user interface is covered along with the associated standards.

8.1 Purpose of Design

A software design is used to create the architecture for the evolving system. Before a programming project begins, the software must first be carefully designed. This design serves as the blueprint for the actual programming or implementation of the system. The major design components of the ILUTE project were the three engines introduced in chapter five. Their design is given in the following sections.

8.2 Software Design of the Simulation Engine

The simulation engine is encapsulated in the CSimulationEngine class. This class is responsible for interacting with the user to create and edit simulation runs. A simulation run includes information such as the base year, the number of years to simulate, the choice of which models to use, the parameters for
those models, and the years for which complete databases should be kept. Once a simulation run has been created, it can be saved and reloaded at any time.

The simulation engine is a Single Document Interface (SDI) executable application that is statically linked with the evolutionary engine and the activity engine. The simulation engine creates the other engines and executes them once per year.

The simulation engine is responsible for all user interactions and must communicate model and parameter choices to the other engines. The simulation engine is also responsible for serializing this data in a format that is suitable for storage in a simulation file (.sim) file. Each simulation file stores all of the information required by the simulation engine to execute or run a simulation.

When the simulation engine runs, it first checks the sub-model directory to find which sub-models are available for inclusion in the simulation run. The terms "sub-model" and "sub-model type" have specific meanings in ILUTE. A sub-model type refers to a type or class of sub-model such as a sub-model of births or marriages. A sub-model is a specific implementation of a sub-model
type such as Researcher X’s birth sub-model vs. Researcher Y’s birth sub-model.

Each sub-model is coded with a sub-model type and a sub-model name to assist in the classification of the sub-model and to ensure that two implementations of the same sub-model type are not simultaneously included in the same simulation run.

8.3 Software Design of the Evolutionary Engine

The evolutionary engine is responsible for updating the database from one year to the next. The evolutionary engine does this by executing a number of sub-models — one corresponding to each sub-model type included in the simulation.

The evolutionary engine invokes each of the requested sub-models in the specified order to update the database from one year to the next. Depending on the model, the unit of iteration could be a person, household, firm, or any other object, aggregate or not, that the sub-model specifies. Once the evolutionary engine has updated the system from one time step to the next, control of execution returns from the evolutionary engine back to the simulation engine. The simulation engine then executes the activity engine on the newly created database.
8.4 Software Design of the Activity Engine

The role of the activity engine is to simulate the activities of the study area over a "typical" day. The purpose of this engine is to simulate all trip-making activities that would occur on that day. These location-based activities are loaded onto the transportation network (auto, transit, or otherwise) as part of the execution of the simulation.

At the time of writing of this paper, the actual operation of the activity engine has not been completed. The ILUTE team must still develop the details of the activity engine. Nonetheless, it is expected that the engine will need to have a time increment on the order of 15 minutes so that scheduling changes can be made dynamically throughout the day.

Since the details of the activity engine have not yet been determined, special care was taken in the design of the framework to accommodate a breadth of possible implementations. Most importantly, the evolutionary engine and the activity engine are structured to allow them to easily communicate in the event that information needs to flow between the two engines.

8.5 Software Design of the Plug-In Models

For efficiency, each sub-model is written in C++ and compiled as a dynamically linked library (DLL). This DLL is placed into the ILUTE sub-
model directory and will automatically be recognized by ILUTE at program start time. The DLL contains two special tags – one to specify the sub-model type and the other to specify the sub-model name. No user involvement is required to add new sub-models to the system. A sample sub-model DLL is given in Appendix A.

8.5.1 Decision-Making Units

The concept of the decision-making unit began during the early development of the ILUTE model when it was unclear whether events should be processed a person at a time or a household at a time. The term decision-making unit is a generic label applied to the person or persons involved in making a given decision. All of the decision-making units for a given decision must be instantiated from the database in order for the decision to be jointly simulated. Each sub-model is free to choose its own aggregation of decision-makers. Sample aggregation types include individual, household, and joint (e.g. spousal).

8.6 ILUTE User Interface

Most Windows-based applications share a familiar and consistent user interface. The use of a common interface improves the usability of the application because users are not required to relearn common operations. For
example, a Windows user intuitively knows to look for the Open command under the File menu, even if the user has never used the specific application before. Keyboard shortcuts such as control-c for copying text to the clipboard are also common to all conforming Windows applications.

To assist in the development of applications that have a common look and feel, Microsoft developed a set of guidelines to help application developers design Windows user interface objects in a consistent manner. "The Windows Interface: An Application Design Guide" is a book available from Microsoft Press that contains a detailed description of these standards.

The user interface classes within the Microsoft Foundation Class Library (MFC) conform to these guidelines. The use of the MFC library reduces the amount of work required to create an application that conforms to these guidelines.
9 Building the Initial Database: Population Synthesis

This chapter describes the technique employed by the ILUTE team to build the initial database from census and other data sources.

9.1 The Need for Detailed Data

The first step in preparing a long run microsimulation is to create or synthesize a base population which is representative of the actual population. Ideally, this base data would consist of actual information about each of the participants in the study area. Realistically, however, obtaining such detailed disaggregate information is not possible. Instead, researchers must rely on a variety of imperfect data sources (disaggregate but sampled, aggregate, cross-tabulated, survey, etc.) as model inputs. The purpose of population synthesis is to generate a set of individual persons (also called actors or trip-makers) that is consistent with the available aggregate data.

9.2 Population Synthesis Techniques

Most population synthesis procedures use some form of Monte Carlo simulation to draw a "realization" of the disaggregate population from the aggregate data. The most promising technique for the creation of synthetic baseline populations is a two-step iterative proportional fitting (IPF) procedure
outlined by Beckman et. al. [1995]: “This procedure simultaneously estimates the multi-way distributions for each census tract within a Public Use Micro Area (PUMA), such that each distribution satisfies the marginal distributions for the census tract (as defined by aggregate census tables) and has the same overall correlation structure as the Public Use Microdata Sample-based (PUMS) multi-way distribution.”

9.3 Synthesizing Other Objects

While this section has discussed only population synthesis, a similar process is used to generate other entities in the system such as firms, residential buildings, non-residential buildings, vehicles, etc.
10 Further Research

This thesis represents the result of several years of research into the development of a suitable framework for a large-scale dynamic microsimulation model that integrates transportation, land use, and environmental issues.

The result of this research, as described in this thesis, is the architecture for the design of the ILUTE model. The structure of the three engines and the mechanism for adding plug-in sub-models has been developed. The structure of the database has been determined, as have the methods for querying and updating it. An initial user interface has been designed as part of the simulation engine. The mechanism through which the sub-models will set and save parameters through that interface is also known.

The development process helped the research team reach consensus and overcome many of the formidable challenges associated with designing such an extensible framework; however, several areas still require further work.

In the simulation engine, more work is required to determine the order in which the sub-models are processed. The extent to which sub-models will communicate with other sub-models is yet to be determined. The adequacy of
two temporal scales needs to be assessed and the choice of temporal scales must also be scrutinized.

In the evolutionary engine, a design pattern for market forces has emerged that could use additional study. Matching spouses, houses, and jobs all seem to have something in common with the market forces of supply and demand. The extent to which evolutionary events such as finding a house or finding a spouse is dependent on activity-based interactions also needs additional study. The firm side of the evolutionary engine currently remains largely unexplored.

In the activity engine, a choice needs to be made between discrete event vs. discrete time simulation. The representation of the network side of the activity model also needs to be considered and developed.

This thesis completes the groundwork for the development of an extensible and open dynamic microsimulation framework. The architecture has been created and many difficult design challenges have been overcome. While a number of issues remain, we are considerably closer to achieving our goal of developing ILUTE, a comprehensive microsimulation model of urban land use and transportation.
References


Appendix A

Sample Sub-Model Dynamic Link Library (DLL) Code
// DLLInfo.h: interface for the CDLLInfo class.
// /*---------------------------------------------*/

#if !defined(AFX_DLLINFO_H__F7E68EB3_EC9A_11D0_BBCF_00A024BC6A95_INCLUDED__) 
define AFX_DLLINFO_H__F7E68EB3_EC9A_11D0_BBCF_00A024BC6A95_INCLUDED__

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class CDLLInfo
{
public:
    CString m_submodelType; // e.g. births, deaths
    CString m_submodelName; // e.g. Paul's birth model, etc.
};
#endif // !defined(AFX_DLLINFO_H__F7E68EB3_EC9A_11D0_BBCF_00A024BC6A95_INCLUDED__)
// Deaths.cpp : Defines the initialization routines for the DLL.
// // 
// // THIS_FILE[] = __FILE__;

#if defined __DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

If this DLL is dynamically linked against the MFC DLLs, any functions exported from this DLL which call into MFC must have the AFX_MANAGE_STATE macro added at the very beginning of the function.

extern "C" BOOL PASCAL EXPORT ExportedFunction()
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    // normal function body here
}

It is very important that this macro appear in each function, prior to any calls into MFC. This means that it must appear as the first statement within the function, even before any object variable declarations as their constructors may generate calls into the MFC DLL.

Please see MFC Technical Notes 33 and 58 for additional details.

BEGIN_MESSAGE_MAP(CDeathsApp, CWinApp)
    // NOTE: the ClassWizard will add and remove mapping macros here.
    // DO NOT EDIT what you see in these blocks of generated code!
END_MESSAGE_MAP()

CDeathsApp::CDeathsApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance

59
The one and only CDeathsApp object

CDeathsApp theApp;

BOOL CDeathsApp::InitInstance()
{
    theInfo.m_submodelType = "Deaths";
    theInfo.m_submodelName = "PAS Death Model";
    return CWinApp::InitInstance();
}
// ProcessHousehold.cpp: function executed by Evolutionary Engine once
// per household per year (current time scale for processing deaths)
//
#include "stdafx.h"
#include "ProcessHousehold.h"

CDLLInfo theInfo;

BOOL ProcessHousehold( void *parm )
{
    SendMessage( *(HWND*) parm, LB_ADDSTRING, 0, (LPARAM) "This message printed by the Deaths submodel" );
    return TRUE;
}

CDLLInfo *GetProcessInfo( void )
{
    return &theInfo;
}
// ProcessHousehold.h: exports plug-in interface

#include "DLLInfo.h"

#define DLLExport __declspec( dllexport )

extern CDLLInfo theInfo;

#if __cplusplus
extern "C" {
#endif

    DLLExport BOOL ProcessHousehold( void *parm );
    DLLExport CDLLInfo *GetProcessInfo( void );

#if __cplusplus
}
#endif
// DLLInfo.h: interface for the CDLLInfo class.
//

#include <afx.h>
#endif
#endif

class CDLLInfo
{
public:
    CString m_submodelType; // e.g. births, deaths
    CString m_submodelName; // e.g. Paul's birth model, etc.
};

#endif // !defined(AFX_DLLINFO_H__F7E68EB3_EC9A_11D0_BBCF_00A024BC6A95_INCLUDED_)

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#include <afx.h>

class CDLLInfo
{
public:
    CString m_submodelType; // e.g. births, deaths
    CString m_submodelName; // e.g. Paul's birth model, etc.
};

#endif // !defined(AFX_DLLINFO_H__F7E68EB3_EC9A_11D0_BBCF_00A024BC6A95_INCLUDED_)
Appendix B

Sample Items from the Initial Data Dictionary
<table>
<thead>
<tr>
<th>Class:</th>
<th>C++ Name:</th>
<th>Database Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>CPerson</td>
<td>PersonXXXXXXX</td>
</tr>
</tbody>
</table>

**Class Description:**
Describes the properties of a person

**Responsibilities:**
die
get married
get divorced
find a job
give birth
leave current job
leave current household
find a household
purchase a vehicle
move to a new residence

age
person ID *(Primary Key)*
household ID
marital status
spouse ID
date of birth
job ID

**Collaborators:**
Household, Job, Vehicle
<table>
<thead>
<tr>
<th>Class:</th>
<th>C++ Name:</th>
<th>Database Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Household</td>
<td>CHousehold</td>
<td>HouseholdXXXX</td>
</tr>
</tbody>
</table>

**Class Description:**
Describes the properties of a household

**Responsibilities:**
- household ID *(Primary Key)*
- household type (family, non-family, single person)
- dwelling unit ID
- number of cars
- household income
- date of formation
- present utility
- equity
- active in housing market flag

**Collaborators:**
- Dwelling Unit, Person
<table>
<thead>
<tr>
<th>Class: Activity</th>
<th>C++ Name: CActivity</th>
<th>Database Name: ActivityXXXX</th>
</tr>
</thead>
</table>

**Class Description:**
Describes an activity

**Responsibilities:**
- activity ID (*Primary Key*)
- person ID
- activity type (work, school, shopping, other)
- priority (measure of importance)
- start time
- end time
- flexibility of start time
- flexibility of end time
- reschedulability
- interruptability
- dependencies

**Collaborators:**
- Person
<table>
<thead>
<tr>
<th>Class:</th>
<th>C++ Name:</th>
<th>Database Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dwelling Unit</td>
<td>CDwellingUnit</td>
<td>DwellingUnitXXXX</td>
</tr>
</tbody>
</table>

**Class Description:**
Describes a dwelling unit (a place where a household lives)

**Responsibilities:**
dwelling unit ID *(Primary Key)*
location (census tract)
tenure (rent, own)
age (in years)
size (area of floor space)
parking (number of spaces)
current market price
dwelling unit type (single detached, duplex, etc.)
active in housing market flag
neighbourhood ID

**Collaborators:**
Neighbourhood
<table>
<thead>
<tr>
<th>Class:</th>
<th>C++ Name:</th>
<th>Database Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vehicle</td>
<td>CVehicle</td>
<td>VehicleXXXXX</td>
</tr>
</tbody>
</table>

**Class Description:**
Describes a vehicle

**Responsibilities:**
- vehicle ID (*Primary Key*)
- vintage (year of manufacture)
- vehicle type
- state of repair
- capacity
- emissions info
- current value
- insurance class

**Collaborators:**
<table>
<thead>
<tr>
<th>Class:</th>
<th>C++ Name:</th>
<th>Database Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firm</td>
<td>CFirm</td>
<td>FirmXXXX</td>
</tr>
</tbody>
</table>

**Class Description:**
Describes a firm

**Responsibilities:**
- firm ID (*Primary Key*)
- number of employees
- type
- number of parking spaces
- parking cost
- building ID

**Collaborators:**
Building
<table>
<thead>
<tr>
<th>Class:</th>
<th>C++ Name:</th>
<th>Database Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building</td>
<td>CBuilding</td>
<td>BuildingXXXX</td>
</tr>
</tbody>
</table>

**Class Description:**
Describes a building

**Responsibilities:**
- building ID (Primary Key)
- location (census tract)
- tenure (rent, own)
- age (in years)
- size (area of floor space)
- parking (number of spaces)
- current market price
- building type (retail, office, manufacturing)
- neighbourhood ID

**Collaborators:**
Neighbourhood
<table>
<thead>
<tr>
<th>Class:</th>
<th>C++ Name:</th>
<th>Database Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neighbourhood</td>
<td>CNeighbourhood</td>
<td>NeighbourhoodXXXX</td>
</tr>
</tbody>
</table>

**Class Description:**
Describes a neighbourhood

**Responsibilities:**
- neighbourhood ID (*Primary Key*)
- location
- crime rate

**Collaborators:**
<table>
<thead>
<tr>
<th>Class:</th>
<th>C++ Name:</th>
<th>Database Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job</td>
<td>CJob</td>
<td>JobXXXX</td>
</tr>
</tbody>
</table>

**Class Description:**
Describes a job

**Responsibilities:**
- job ID (*Primary Key*)
- firm ID
- occupation
- hours
- benefits
- salary
- start time
- end time
- flexible time
- days of week (e.g. Monday to Friday)

**Collaborators:**
- Firm
<table>
<thead>
<tr>
<th>Class:</th>
<th>C++ Name:</th>
<th>Database Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SchoolSchedule</td>
<td>CSchoolSchedule</td>
<td>SchoolScheduleXXXX</td>
</tr>
</tbody>
</table>

**Class Description:**
Describes a person’s school schedule

**Responsibilities:**
- school schedule ID *(Primary Key)*
- school ID

**Collaborators:**
- School
<table>
<thead>
<tr>
<th>Class:</th>
<th>C++ Name:</th>
<th>Database Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>School</td>
<td>CSchool</td>
<td>SchoolXXXX</td>
</tr>
</tbody>
</table>

**Class Description:**
Describes a school

**Responsibilities:**
- school ID (*Primary Key*)
- grades
- number of students
- number of staff
- Building ID

**Collaborators:**
- Building
<table>
<thead>
<tr>
<th>Class:</th>
<th>C++ Name:</th>
<th>Database Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Land Use</td>
<td>CLandUse</td>
<td>LandUseXXXX</td>
</tr>
</tbody>
</table>

**Class Description:**
Describes a land use (building stock – supply side)

**Responsibilities:**
- land use ID *(Primary Key)*
- type (commercial, retail, high-rise office, industrial, etc.)
- location
- size

**Collaborators:**
### Class Description:
Describes a transit route

### Responsibilities:
- transit route ID (*Primary Key*)
- route number
- set of transit nodes and links
- headways

### Collaborators:
- Node, Link
<table>
<thead>
<tr>
<th>Class:</th>
<th>C++ Name:</th>
<th>Database Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link</td>
<td>CLink</td>
<td>LinkXXXX</td>
</tr>
</tbody>
</table>

**Class Description:**
Describes a link in a road or transit network

**Responsibilities:**
- link ID (*Primary Key*)
- capacity
- length
- type
- free-flow speed
- volume delay function
- travel time
- valid modes

**Collaborators:**
<table>
<thead>
<tr>
<th>Class:</th>
<th>C++ Name:</th>
<th>Database Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trip</td>
<td>CTrip</td>
<td>TripXXXX</td>
</tr>
</tbody>
</table>

**Class Description:**
Describes a trip

**Responsibilities:**
- trip ID (*Primary Key*)
- origin node
- destination node
- activity
- purpose
- mode
- auto occupancy
- desired arrival time

**Collaborators:**