Ta Da! - The Text Adventure Design Assistant a Visual Tool for the Development of Adventure Games

Sean Molley
Western Kentucky University

Follow this and additional works at: http://digitalcommons.wku.edu/theses

Part of the Programming Languages and Compilers Commons

Recommended Citation
http://digitalcommons.wku.edu/theses/347

This Thesis is brought to you for free and open access by TopSCHOLAR®. It has been accepted for inclusion in Masters Theses & Specialist Projects by an authorized administrator of TopSCHOLAR®. For more information, please contact topscholar@wku.edu.
TA DA! – THE TEXT ADVENTURE DESIGN ASSISTANT
A VISUAL TOOL FOR THE DEVELOPMENT OF ADVENTURE GAMES

A Thesis
Presented to
the Faculty of the Department of Computer Science
Western Kentucky University
Bowling Green, Kentucky

In Partial Fulfillment
of the Requirement for the Degree
Master of Science

by
M. Sean Molley
December 1997
TA DA! -- THE TEXT ADVENTURE DESIGN ASSISTANT

A VISUAL TOOL FOR THE DEVELOPMENT OF ADVENTURE GAMES

Date Recommended 9/2/1997

Dr. Robert Crawford
Director of Thesis

[Signature]

Director of Graduate Studies

[Signature]

Date
ACKNOWLEDGEMENTS

Although the Text Adventure Design Assistant is primarily a product of my own years of experience in designing games, I have been strongly influenced over the years by many members of the text adventure community. I would like to thank in particular the following individuals and groups: Magnus Olsson, with whom I have had numerous productive discussions on game design and implementation; Gerry Kevin Wilson, whose laments about the difficulty of TADS and Inform led to the original design for this project; Adam Thornton, who urged me to continue with the project and provided many useful suggestions; Michael J. Roberts and Graham Nelson, the developers of TADS and Inform, respectively; and all the many hundreds of people on the Usenet newsgroups rec.arts.int-fiction and rec.games.int-fiction with whom I have agreed, argued, debated, discussed, and analyzed these issues over the past six years. They are a tremendously dynamic and creative group of people.

I would also like to thank the beta testers, who have stuck with me even though the software moved from DOS to Windows 95 and C++ to Delphi 3. Their suggestions and criticisms have made this product far better than it would have been had I worked in a vacuum.

Finally, I would like to thank the original members of Infocom, whose games captivated me for hundreds of hours and whose work inspired me to learn programming in the first place. Infocom is owed a debt by the rest of the computer gaming world which is seldom acknowledged and can never be fully repaid. While Crowther and Woods invented the genre, it was the Implementors at Infocom who defined it. This work is offered in the hope that perhaps someday it might enable someone who would never otherwise have done so to experience the joy which I, like so many before me, have found in the craft of the adventure.
Contents

1 Introduction 1
  1.1 A Brief History of Text Adventures 1
    1.1.1 Adventure 1
    1.1.2 Zork and the Rise of Infocom 2
    1.1.3 The Decline of Text Adventures 3
  1.2 Modern Adventure Games 4
  1.3 Why Write a Text Adventure? 5

2 Foundations 7
  2.1 Special-Purpose Programming Languages 8
    2.1.1 Adventure Game Toolkit 8
    2.1.2 TADS 9
    2.1.3 Inform 10
    2.1.4 Other Languages 12
    2.1.5 ALAN 12
    2.1.6 Hugo 12
    2.1.7 Others 13
  2.2 The Evolution of TA DA! 14

3 TADS 17
  3.1 Overview 17
ABSTRACT

In this paper, I survey past and present tools available to text adventure game authors, and then describe a new product: the Text Adventure Design Assistant (TA DA!), a visual programming system for creating text adventure games. My system consists of two parts: an abstract framework which defines an archetypical game, and a user interface which allows for the construction of games in a visual manner by manipulating the elements of the abstract game to produce a concrete design. The two most popular contemporary programming languages for creating text adventure games, TADS and Inform, are compared and contrasted, and my abstract framework is adapted to both of these languages. The traditional pencil-and-paper design process used by adventure game authors is studied and its application to the development of TA DA! is described. Finally, the implications of TA DA! and similar advances in visual programming are discussed and I predict future trends in the design of both adventure games and other application domains based on this work.
Chapter 1

An Introduction to Adventure Games

My primary concern in writing this paper is to focus on the theoretical and technical aspects of creating a programming-language independent abstraction for a specific application domain (try saying that ten times fast). However, this paper is also about programming computer games. In order to understand what motivated the creation of a tool such as TA DA!, one must first have a background in the area of text adventure games. These games have been around for some 20 years now, and have contributed greatly to a number of areas in computer science in addition to their obvious entertainment value. Advances in natural language parsing and machine-independent compilation (often known as \textit{p-code}) can be traced at least in part to the technology that was developed for adventure games.

1.1 A Brief History of Text Adventures

1.1.1 Adventure

The original adventure game on a computer was called, appropriately enough, Adventure. It was a simulation of a cave, specifically Mammoth Cave in Kentucky, and is sometimes known by its full name “Colossal Cave Adventure.” While it was originally intended to
be used to help with the study of the features of the cave, its developers found that an interactive simulation of a real-world setting had much more potential as a game than it did as a geologic study aid. Its interface was rudimentary, but adequate for the time: the user could communicate his or her desires to the game through the use of a two-word parser which understood sentences of the noun-verb imperative form. So the program could accept commands such as GO NORTH, EAT FOOD, and KILL TROLL. Though the parser was crude, the game itself was not. It was vast and richly detailed, with fabulous treasures, sinister dwarves, and even a sneaky pirate who picked up anything that wasn’t nailed down. It was quite popular with its limited audience (those who had access to a mainframe computer).

1.1.2 Zork and the Rise of Infocom

Not long after word of Adventure’s existence got around to the academic community, a group of college students at MIT got together and played it. They were suitably impressed but concluded that they could do better (as computer science students often do). Their collaborative efforts led to a huge FORTRAN game called Zork which ran on a PDP-11 computer. Zork was designed to be played by connecting to MIT’s machine, and supported multiple users playing at the same time. It proved to be tremendously popular, both because it had more of a plot than Adventure and because the MIT students had substantially improved the parser to the point that it could understand simple adjectives, adverbs, and compound sentences. In fact, so many people played Zork that the students who had created it decided to form a company and produce a version of the game for a variety of platforms, including the just-released IBM Personal Computer. Zork finally made its debut in 1983, in a series of three games which when combined made up most of the original mainframe version.

Zork was a huge success: the trilogy of games would eventually sell over one million copies during the 1980’s. Infocom became an overnight sensation and went on to produce over 35 more text adventure games, each more complex than the ones that preceded it and each one containing a rich story with challenging puzzles and pages upon pages of evocative
Infocom soon found itself with a large core of very loyal fans — many of whom are still writing and playing text adventures even today. People across the nation and around the world took notice. One of the more famous fans of Infocom was television news anchorman Ted Koppel — whose statement "if it's 2 AM, this must be Zork" — was prominently featured on the inside cover of the game.

Because the games were very literary in nature, being composed of long stretches of text punctuated by player commands, they came to be known as "interactive fiction," a name which persists to this day. Text adventure games cast the player into a variety of situations, from swordsman to wizard to an Ensign in the Stellar Patrol. Because text takes up comparatively little space inside a computer's memory, the designers of these games were able to create relatively large worlds, even given the severe hardware limitations of the day. Infocom's first games required 64K of RAM to run; these requirements gradually increased (largely due to increases in the complexity of the parser) to 128K and finally 256K in the latter part of the 1980's. These system requirements, although they were considered massive at the time, are a far cry from today's games, which generally require 4, 8, or even 16 MB of memory just to run the install program!

1.1.3 The Decline of Text Adventures

The 1980's were the "golden age" of text adventures. Infocom was at its height in the middle part of the decade, ruling the computer game world from a lofty perch. Other companies, such as Level Nine in England, also experienced great success with text adventures during this time. (Their games remain very popular in Europe, where many older computers are still to be found). However, computers grew steadily more powerful as the decade progressed, culminating with the release of the EGA (Extended Graphics Adapter) around 1987. This video card was an order of magnitude better than anything that had come before it, with its ability to display 16 colors on the screen simultaneously from a palette of 256 available colors. Up to this point, text adventures had remained the dominant force in gaming, simply because the hardware didn’t support anything else. With the advent of
EGA, followed shortly by the even more powerful VGA (Video Graphics Array) in 1989, game companies could add splashy color graphics to their products. Infocom steadfastly refused to give in to this graphical trend, stating that no picture could replace the power of the imagination. Alas, consumers didn't agree, and flocked to the new games in large numbers. Companies such as Sierra On-Line and Electronic Arts became the dominant force in the game industry; Infocom eventually filed for bankruptcy and was purchased by Mediagenic in late 1989. Today the label is owned by Activision, which continues to produce adventure games under the Infocom name. However, all of the original members of Infocom have long since left the company, and the games being produced under the label today are related to Infocom's text adventures only in that they share the same setting and general theme.

1.2 Modern Adventure Games

Computer games have long been the applications which pushed the limits of existing hardware, especially with regards to processor speed and graphics capabilities. Today's adventure games are no exception. While the complexity of plot and good puzzle design that characterized the games of the 1980's are gone, modern adventure games remain as popular as their predecessors. Perhaps the best example of this is *Myst*, which was released in 1994 and became the "killer app" which caused hundreds of thousands of consumers to purchase CD-ROM drives.

Today’s adventure games retain the essential idea behind the original text adventure games: to cast the player as the protagonist in a story which unfolds as a result of the actions chosen by the player. While the early games expressed themselves through words, much like a novel, today's games use multimedia flash and sizzle — digitized sound and full-motion video. Modern adventure games have more in common with television than with books — they are generally short in length, have limited plots, and little or no replay value. Where text adventure games emphasized plot, story, and puzzle-solving, today's games tend to emphasize visual effects and require little thought on the part of the player.
This “dumbing down” of the genre has actually not been as damaging to sales as one might think, as it has broadened the market because most players are now able to play the games all the way through without becoming stumped by some puzzle and being unable to finish. Adventure games continue to evolve, however, and many consumers are pushing for a return to the story-centric games of the 1980’s. Perhaps some day the text adventure will rise again.

1.3 Why Write a Text Adventure?

Among those who hope that the text adventure will rise again are a group of die-hard fans who played the original games (or have discovered them in the discount racks at software stores, where many of Infocom’s games can still be found today), as well as younger players who have become dissatisfied with the current crop of adventures and seek more challenging and better-written interactive fiction. For these individuals, one of the best ways to experience the power of a text adventure is to write one themselves. Writing an adventure is similar to writing a book in that the author must have some idea of plot, conflict, and characterization, as well as providing a meaningful “path” through the story. However, writing an interactive fiction game also requires the author to be cognizant of puzzle design, player choices, and other issues. Think of a text adventure as a book in which the pages may be read in any order and you see the difficulties which can arise.

So why would any sane human being want to spend hundreds of hours creating a game which will probably never be commercially successful and which might only be seen by a handful of devoted text adventure fans? The fact is that taking a game, any game, from design to completion is a very satisfying process. In addition, text adventures do not require the knowledge of graphics hardware and mathematics that more modern games rely upon, and so are more accessible from a programming standpoint. Additionally, because of their low system requirements, text adventures may be written by anyone with access to virtually any kind of computer, past or present. Finally, many of today’s current adventure games are being prototyped as text adventure games first. In this way the designers hope to recapture
some of the emphasis on story and puzzles that made the original games so popular.

Additionally, text adventures are sometimes used for non-game applications. Several companies have used the paradigm to produce training simulations (much like Crowther and Woods originally intended for their Adventure program). By providing an immediate, first person medium, text adventure games have an immersive quality that is not present in other types of computer training. Because of the variety of responses possible using the written word as opposed to computer graphics (it's much easier to write a paragraph than to draw a picture), text adventures remain the most powerful and flexible medium available to computer game authors. Indeed, text adventures, because of their non-deterministic nature (the next game state is not dependent entirely on the previous game state, but dependent also upon the actions of the player and in many cases random factors), provide a level of complexity and flexibility that is simply not found in today's games. In a sense we have taken one step forward and two steps back — for every advance in the graphics and hardware areas, game design has suffered a corresponding setback as game companies try to produce the splashiest, flashiest product which fits on the largest number of CDs.
Chapter 2

Foundations of Adventure Programming

Once a person has decided to undertake the task of designing and programming a text adventure game, it becomes necessary to develop a whole host of auxiliary code not directly related to the game itself. For example, the parser is the part of the program that accepts and interprets the English-language sentences typed by the player and conveys these actions to the game proper. It is also necessary to develop other routines, such as functions to save and restore the game, which are not a part of the game itself but improve the playability a great deal. How frustrating would it be to the player if it was necessary to start the game from scratch each time the computer was rebooted? Given that many text adventures are very large — often requiring 100 hours or more of playing time to complete — these supplemental routines become necessary, even though they add nothing to the actual game play. For the new adventure game programmer, then, a large amount of time will be spent on developing code which actually has nothing at all to do with the game! And while much of this code can be reused in future games, many would-be game programmers have been so daunted by the task of writing the parser alone that they gave up long before producing anything even resembling a completed game. Despite the fact that text adventures are “simpler” than their graphics-based descendants, they are by no means “simple.”
Fortunately, several individuals have recognized the inherent difficulty in programming text adventure games and have taken steps to make the task easier.

2.1 Special-Purpose Programming Languages

While it is possible to write a text adventure completely from scratch in a high-level language, for most casual programmers the task is rather difficult and perhaps not worth the effort. However, once a particular piece of support code (such as a parser) has been written, it can be reused in future games, meaning that only an initial investment of time and energy must be made in order to develop routines that will be useful for many games. Hoping to save other people from having to “reinvent the wheel,” a fair number of people have attempted to develop toolkits and even whole languages that are specially designed for programming adventure games. Because of the existence of these languages and kits, there is no reason at all for a new game designer to even consider coding a game from scratch. While this author has written text adventures in several high-level languages from the ground up, that was in the days before these packages were available. TA DA! is a culmination or evolution of the many design tools now available to the would-be game designer. Indeed, TA DA! itself is built upon several existing tools and would not itself be possible had those tools not been created.

2.1.1 Adventure Game Toolkit

The Adventure Game Toolkit, or AGT, is one of the first and perhaps the most popular “utility” created specifically for writing text adventure games. Written by David Malmberg in the late 1980’s, AGT has undergone numerous revisions and was finally released to the public domain in 1995. Not a programming language unto itself, AGT is actually a collection of library routines written in Turbo Pascal which contains most of the support code (such as a parser and game save/load routines) that is needed for a text adventure. The programmer then need only do the game-specific coding for his or her particular adventure, saving much time and effort and concentrating on the game itself. AGT remains popular to this day,
although it has been criticized for the quality of its parser.

In truth, AGT is a fairly robust toolkit, but it has gotten rather a bad reputation in the adventure game community. This reputation is primarily due to the fact that because it was really the first toolkit available, it was widely used and so many hundreds of games have been written using AGT. The natural consequence is that there were far more bad games than there were good ones, which has led to the stereotype that any game created with AGT must be inferior. This label is far from accurate; in fact, Malmberg used to sponsor an annual AGT Programming Contest that drew brilliant entries from around the world. Many of the winners have gone on to some degree of fame and fortune as shareware games.

However, because AGT is a set of library routines, and not a compiler or programming language, its use requires that the designer must know and own Turbo Pascal. Additionally, the author has not actively supported AGT for some years, and so it is rather “out of date” as far as advances in parser and other game technologies go. For purposes of this work, AGT is useful for historical perspective and as an ideological forerunner of TA DA!, but there is no other connection between the two.

2.1.2 TADS

TADS, which stands for Text Adventure Development System, was first released in 1987 by Michael J. Roberts. TADS is a full-featured programming language, with its own (very C-like) syntax and compiler. TADS does not produce executable files but rather compiles the game to a p-code form which is then executed by a run-time interpreter. Most text adventure compilers use p-code to provide a measure of platform independence, and TADS is available on most common platforms.

Until early 1997, TADS was shareware and had a fairly high registration fee. However, the number of registrations had been falling for some years (primarily due to the release of Inform — see below), and so the author decided to release the entire package as freeware in January of 1997. In terms of the quality of games which can be produced, TADS is
an order of magnitude better than anything which came before it, including AGT. At the
time of its first release, TADS provided run-time support for both the IBM PC and Apple
Macintosh platforms, meaning that a programmer could develop a game that would run on
both systems without even needing to recompile the code.

TADS became very popular in the early 1990's, when it was discovered by a group of
people on the Usenet newsgroup rec.arts.int-fiction, which is dedicated to the programming
and design of text adventure games. Quite a few shareware and freeware games (and even
a couple of commercial attempts which were unfortunately largely unsuccessful) have been
produced entirely in TADS in recent years.

TADS has undergone numerous revisions and updates during its lifetime. The most
recent of these came in 1996 when TADS was updated to version 2.1, which was the final
update before the product became freeware in early 1997. Version 2.1 provides support
for a number of powerful features, including a self-modifying parser and dynamic creation
and destruction of objects at run-time. TADS has had a very large impact on the text
adventure community, but its learning curve is very steep. The latest version of TADS has
a syntax and operator set that is very similar to C++, and is about as difficult to learn,
which is off-putting to many would-be game designers. However, for those who persevere
and master the language, TADS is a powerful and robust tool.

2.1.3 Inform

In the early 1990's, a member of the department of mathematics at Oxford University in
England wrote to Mike Roberts (author of TADS) asking whether or not there would ever be
a version of TADS for some of the mainframe computers (such as UNIX workstations, and
the Acorn Archimedes, which is quite prevalent in England). Roberts replied that he saw no
reason to support those platforms, since the work involved in porting the TADS compiler
to those architectures would probably not be justified because it would not generate a
very large number of new registrations. Annoyed that he was unable to play any of the
games being created with TADS and apparently never would be without buying a PC, this
mathematician resolved to write his own adventure game compiler which would run on the Archimedes — and, for that matter, every other machine on the planet. His name was Graham Nelson, and the product he produced — called Inform — has had a tremendous impact on the text adventure game community.

What Nelson did was not only create a new adventure game compiler — he created a compiler which produces files that use an identical file format to that originally used by Infocom in their commercial games. It is a p-code format, much like the one TADS uses, with a major difference: the run-time for the Inform code (called “z-code”, in honor of Zork) is written entirely in standard ANSI C and therefore is portable to any machine in the world with an ANSI-compliant C compiler. More specifically, a game written using Inform can be played on virtually every computer on the planet Earth.

Not only did Nelson produce the compiler and associated library routines, he also went one step further and made the entire package freeware. The result is that Inform has become far more popular than even TADS in just the last three years. It has been featured in magazine articles in Europe and has literally thousands of users, a very high percentage of the modern adventure game community. Inform is perhaps the biggest single development in the text adventure game world since the demise of Infocom at the turn of the decade.

The major problem with Inform is its syntax. Graham Nelson was not a computer scientist; he was a mathematician. He knew nothing about formal language design and so went with what made the most sense to him. The result is that Inform’s syntax is nothing less than hideous compared to more “properly” designed languages such as C++ and even TADS. If TADS has a learning curve that is steep, Inform’s learning curve has no horizontal slope at all. It is very difficult to learn and very difficult to use. However, it is very powerful — in some ways, more powerful than TADS — and it is completely platform-independent. It is also being actively supported by both the author and a large number of other interested people, unlike TADS, and so at the time of this writing Inform is certainly the language of choice for new adventure game authors.
2.1.4 Other Languages

While AGT, TADS, and Inform are probably the three best known adventure game development tools, there have been several other recent efforts which have not met with as much recognition or success.

2.1.5 ALAN

ALAN, which stands for Adventure LANguage, was developed by two professors of computer science in Sweden and is freeware. It is the only adventure game language designed to be used for writing games on the VAX/VMS architecture from Digital Equipment Corp. ALAN's language design is very rough, appearing more like a scripting language than a procedural one. ALAN's biggest advantage is its strong support for non-player characters (NPCs). Non-player characters are the people and creatures that "live" in the game world and with whom the player must interact. Creating "believable" NPCs has long been a problem for text adventure game authors, touching as it does on issues in language recognition, artificial intelligence, and intelligent reasoning and planning. ALAN provides strong support for creating useful and believable NPCs, but its support for most of the other aspects of game design and programming is weak, which is primarily why it has not become popular.

2.1.6 Hugo

Hugo, like ALAN, was created to target a particular platform that its author felt was neglected. In this case, the platform is IBM's OS/2 operating system, which has never achieved much popularity among anyone, but especially among computer game manufacturers. Hugo (named after French author Victor Hugo) is the only adventure game language which is purely object-oriented, in the tradition of SmallTalk. Although it is significant for this reason, Hugo is still undergoing some revisions at the time of this writing and is not yet of the same quality as the previously mentioned products. Additionally, it creates games that run only under OS/2, which greatly limits its audience. Still, Hugo is significant as
the first and only purely object-oriented programming language for adventure games.

2.1.7 Others

David Betz produced a system in the mid 1980's called ADT which was fairly popular but not especially powerful in the games it could produce. In particular it had a very limited parser. Betz updated his language in the early 1990's, adding a more powerful parser and improving some of the other features. He named the new language LADS. While ADT achieved some popularity, LADS has gone almost completely unnoticed in the shadow of TADS and Inform (and rightly so, for it is inferior to both.)

Many other attempts have been made to produce tools and compilers for adventure games. Several such systems exist for the Macintosh, including one older program called World Builder which is still used today. World Builder actually provides support not only for pure text adventures but also for rudimentary graphics within adventures, which makes it unique among adventure game development systems. Mike Roberts started a graphical version of TADS called TADS/G, but never got sufficiently interested in the project to take it past the design stage.

Most of the other tools that have been created are similar to AGT — not compilers or programming languages, but just sets of routines written in an existing high-level language. Several such “kits” have been created in SmallTalk and LISP, and one very interesting tool set was built using Turbo Prolog a few years ago. These sets of routines are useful and can be helpful if the author is already very familiar with the base language. However, game designers are almost always better off using a language such as TADS or Inform which has been specially designed for creating text adventures. Languages designed specifically for writing adventure games almost always offer a feature set which is more intuitive to the game author.
2.2 The Evolution of TA DA!

In the preceding pages, many systems for creating adventure games have been discussed. A large number of people have already considered and addressed the problem of creating systems to allow authors to write text adventure games. Why add to the confusion? What does TA DA! have to offer that makes it unique and significant? The answer lies in the visual paradigm.

The most significant problem with all currently available tools is that all of them require substantial experience with both structured and object-oriented programming. Spending thousands of hours writing code is not a problem for the game designer who is already trained in computer science or who has done significant programming, but not all would-be authors are computer scientists. For the person who has a great game idea, or is a gifted writer, but has never done any programming and has no desire to invest months or years in learning how, the fact that one must be or become a proficient programmer to write games is a major drawback and one which has basically conspired to keep all non-programmers out of game design for many years. Even among those who know how to program, learning TADS or Inform often proves to be more trouble than it is worth. Developing a computer game is a time-consuming activity, even with one of these powerful tools; moreover, since there is almost no chance of financial reward from developing a text adventure, the creation of these games is largely a labor of love.

It was out of a discussion of this very problem that TA DA! was born. Having followed the Usenet newsgroups concerning interactive fiction for some time, it was noticed that the number of complaints and questions about TADS and Inform was very high — perhaps 50 percent or more of the volume of messages on the newsgroups. With this amount of time being spent in fighting the languages, rather than developing the games themselves, how could authors hope to produce quality products? And how could a non-programmer ever hope to master either of these languages when even computer science majors were having this level of difficulty?

TA DA! is an attempt to address the issue of game design from a different perspective.
Rather than developing games as structured programs, TA DA! takes a visual approach to design. Game designers have followed a general set of steps and principles from the very beginning. TA DA! attempts to re-create those steps on the computer, using a graphical user interface to allow the designer to concentrate on what the game should look like and what it is supposed to do rather than on how things are to be accomplished in a particular programming language. Once the game has been designed, the program then generates the code in both TADS and Inform which can then be compiled to produce a finished game. At no time in the process does the user actually write code — he or she only specifies behaviors and designs the layout of the game using the visual interface. By raising the level of abstraction above the structured program level, it is hoped that TA DA! will make adventure game design and creation accessible to anyone who has an idea for a game and is able to formulate that idea into a concrete specification.

Why support two target programming languages? The reasons are many, but the most important reason is that supporting both TADS and Inform required the development of an abstraction which removed issues of language syntax from the specification of an adventure game. The creation of this high-level abstraction makes the tool robust enough that, should a new programming language come along which is better than TADS and Inform, TA DA! can easily be re-configured to work with this new language simply by mapping the new language to the abstraction. The abstraction works because adventure games are a well-defined subset of the set of all applications. They exhibit certain behaviors and features which are consistent no matter what the subject of the game may be. The abstraction itself and the assumptions leading to it are discussed in great detail in the following chapters.

Another good reason to support both TADS and Inform is that no tool is perfect. As it stands, TA DA! can generate code for a wide variety of game designs — but it is unlikely that TA DA! or any other tool will ever be able to generate code for all possible ideas. Because we cannot write a universal program generation tool, the programmer or designer who wants to go beyond the capabilities of any visual design tool will still have to learn some kind of underlying scripting or programming language. Since TADS and Inform are both
very popular, and so different from one another, providing support for both languages both increases the utility of the tool to current developers who have already made an investment in learning one or both of these languages and also provides a choice for new developers who might find one or the other of the two languages easier to use. By not forcing a language upon the user, the utility of the tool is greatly increased in those situations where the game design is not adequately developed within the visual paradigm.

Finally, the fact is that people still play these games — on a staggering variety of platforms, from PC's to mainframes to Apple Newton personal digital assistants. There are run-time systems for Java, for laptop computers, and for supercomputers. However, the set of platforms supported by TADS and the set of platforms supported by Inform are not the same. Many people have a run-time for TADS but not for Inform, or vice-versa. Rather than having to choose to only support one segment of the adventure game world or the other, TA DA! frees the designer from this concern by generating code for both languages. In this way the designer is guaranteed that all platforms will be able to run one version or the other of his game without needing to recompile. For a shareware product (which is what most text adventures are these days), compatibility with a wide variety of platforms is essential to maximize the number of potential registrations. The need to maximize exposure is especially acute in the adventure game community, which is not that large to begin with. Neglecting a single platform can mean the elimination of twenty percent or more of the potential audience — never a wise marketing decision.

In summary, it is clear that a tool which speeds up the design process for experienced authors, as well as simplifying the enormous complexity for beginners, would be invaluable to both groups. It would allow the development of new games and new authors while still providing support to the entire text adventure community with its myriad of platforms and operating systems. By abstracting the design process to one which is language-independent, TA DA! seeks to do for text adventure authors what the Z-machine did for Infocom's early games: allow an author to concentrate on writing a good game without having to worry about the dirty details which go on behind the scenes.
Chapter 3

A Look at TADS

This chapter briefly discusses TADS, one of the two existing adventure game programming languages supported by TA DA!. It is not intended to be a tutorial or a reference, but merely to illustrate some features of the language which make it suitable for programming adventure games and which differentiate it from other, more general, programming languages such as C.

Note that the discussion here applies only to the most recent version of TADS — which will most likely be the final version with the release of the system as freeware in early 1997. Earlier versions of the language did not support many of the features of the 2.1 release, such as advanced parser management and dynamic object instantiation.

3.1 Overview

TADS was created to be an adventure game programming system. It was not intended as a general-purpose programming language, and would not function well in that capacity. Nevertheless, TADS provides a high degree of support for both structured and object-oriented programming styles, with a decided emphasis on the latter. TADS has always been object-oriented rather than procedural in nature, although its support for the OO paradigm has certainly gotten stronger over the years. The TADS compiler takes game
source code and produces a p-code format which is then interpreted by a separate run-time system. TADS provides excellent support for its users, with several tutorial games, a good user manual, and an integrated debugger all included in the package. TADS also comes with a robust set of pre-defined classes and objects useful for adventure games, packaged in a “header file” format which will be familiar to C or C++ programmers. Users are able to package their own code into smaller compilation modules and link these modules together to create the finished game.

The compiler and interpreter are distributed together and have been ported to various systems, including MS-DOS, Apple Ilgs and Macintosh, a few flavors of UNIX, and the Atari ST. Both the interpreter and compiler were proprietary until early 1997, when the author of the system, Michael Roberts, released them to the public domain. Had this happened a few years earlier, TADS might control a greater share of the market than it does; as it stands, many people chose to use Inform because that system was (and shall always remain) entirely free. Although TADS has not been ported to a wide variety of platforms, it does support the two largest — PC and Mac — and while the task of porting the compiler and interpreter is very difficult, a game which is compiled on one system may be run on any system to which the run-time has been ported. Thus, a game compiled using TADS on a UNIX workstation can run without changes on a Macintosh, a PC, or (yes) an Atari ST.

It is doubtful that any more ports of TADS will be made, since the author no longer actively supports it as a commercial venture, but a large number of veteran authors use TADS and will continue to do so because it is the language which they are most familiar with (and for its similarity to C, which is a draw for many would-be authors who are already familiar with that language). While it may eventually be replaced by Inform in the long haul, TADS is certainly a very popular language at present; therefore, any tool such as TA DA! which intends to provide support to both new and old authors must provide support for TADS.
3.2 The Language

As one would expect, most of the work of a TADS programmer lies in creating objects for
the player to interact with and then linking those objects together into a complete game
world. TADS provides a number of abstract base classes which taken together represent
a comprehensive framework for adventure games of all sorts. These base classes include
rooms, objects, and characters, as well as the concepts of light, darkness, containers, and
transparency. Furthermore, the standard TADS “ADV.T” and “STD.T” packages define
many common verbs and commands which the parser should recognize. Of course all of
these features are completely extensible, and a game author may choose to replace them
with his or her own creations. Most of the time, however, the standard classes are more
than sufficient, and an author will instead derive new classes which add to rather than
replace the functionality provided by TADS.

In general, TADS programmers use multiple inheritance frequently, far more so than
do even C++ programmers, because many of the abstract classes provided implement only
a single feature (such as transparency or containment). The strong support provided for
multiple inheritance means that a TADS author has a great amount of power and freedom
to create unique and interesting objects, but it also means that a substantial amount of time
must be spent learning the object hierarchy and the base classes provided in the packages.
While a simple object, such as a rock, derives only from class item, a more complex object
such as a trophy case might have to descend from surface, fixeditem, openable (a subclass
of container), and transparentItem.

While TADS is primarily object-oriented in nature, it does provide strong support for
C-style functions, as well as for most of the operators of the C programming language.
TADS provides substantially enhanced string-processing capabilities, however, just as one
would expect from a language whose primary purpose is to present text to the user. TADS
is not strongly typed, and object identifiers are all global in scope. Strings are not usually
associated with a variable; rather, the text is merely placed in double quotes inside the
program itself, and the compiler handles the rest. Double-quoted strings are meant to be
displayed directly to the screen, while single-quoted strings are used to specify keywords (such as nouns, verbs, and adjectives) which the parser must recognize. It is very easy to associate words (such as adjectives) with game objects; they are simply made properties of the object in question and TADS' parser automatically handles the details.

Indeed, the string capabilities bear further discussion, as they are one of the best features of TADS. The language supports a variety of advanced string-related features, including embedding of expressions and function calls within strings which will be evaluated at runtime, different whitespace characters for tab and newline, string concatenation and dynamic string construction, substring matching and searches, easy conversion between string and numeric values, conversion between upper and lower case, and even pre-processing and changing the player's input string at run-time to allow for complete control over player input. TADS also has sophisticated output formatting, including format strings, sexed objects (male, female, and neuter), and supports the full ASCII character set for output as well as text formatting command such as boldface and italics which will be correctly interpreted by the run-time system for the particular platform. For C programmers, who are used to regarding strings as an opportunity for pain and suffering, working with TADS will be a delight.

Specification of variable data types is not explicitly supported, nor is it needed except in terms of class names for inheritance and method calls. The semicolon is used to terminate object declarations, but not individual statements. Functions may define local variables, but all variables defined within an object itself are considered to be properties of that object. Objects may refer to global variables, although by convention TADS programmers usually create a “globals” object which contains all global data.

TADS supports the usual flow-of-control structures, including if-then-else structures, while, do-while, and for loops, and case statements. TADS uses curly braces to delimit blocks of code, just as in C, and blocks may be nested. TADS also provides, break, continue, and goto statements, and has a variety of specialized commands which allow the programmer to invoke standard adventure game user-interface elements (such as a file-open dialog box)
with ease. The comment conventions are the same as C++, with support for both block and end-of-line comments. Finally, TADS has good support for arrays (lists), which may contain objects of multiple types and may be dynamically constructed and modified. TADS even has car and cdr keywords, which behave just like their LiSP counterparts.

In summary, TADS is a robust programming language which is strongly influenced by Pascal, C, and C++. TADS syntax is a mixture of structured and object-oriented programming constructs which might be confusing to the novice programmer. Because its features have changed so much over time, TADS provides a number of compiler directives which allow the use of deprecated features so that older code will still compile properly. Overall, a game programmer who has been using TADS since its creation will probably find the most recent version more confusing than will a programmer who is seeing TADS for the first time. TADS is similar enough to other modern programming languages that mastery of its syntax is possible for both the veteran and novice user.

3.3 Writing a Game Using TADS

To create a game using TADS, the programmer's primary job is to specify the objects and characters which make up the world and how they will interact with each other and with the player. The process of creation should come as no surprise, since the whole purpose of these games is to create a sort of "virtual world" within which the player will strive to accomplish his or her goals. Unlike Inform, TADS programmers generally do not have to resort to writing a large number of specialized non-object-oriented functions in order to accomplish the goal of creating believable, interesting, dynamic objects. Objects created using TADS' "pre-packaged" class hierarchy tend to be very robust and require little work on the part of the user. For example, consider the following code snippet:

```
mine: room
  sdesc = "Gold Mine"
```
ldesc = "You're standing in an abandoned gold mine. Ominous creaks come from the support timbers which surround you on all sides, and the air is thick with white dust from the thousands of tons of rock."

north = mineEntrance

nugget: item
  sdesc = "gold nugget"
  ldesc = "Your eyes grow large with avarice as you regard this fist-sized chunk of solid ore."
  noun = 'nugget'
  adjective = 'gold' 'golden'
  location = mine

Notice the variety of properties that can be specified for objects. The programmer can provide several levels of detail for describing the rooms, items, and characters that the player will encounter, and is able to specify the nouns and adjectives that will be used to refer to them. There is no limit on the number of words that the parser can recognize, so the author is able to provide as many synonyms as possible for each verb and noun. Providing as many synonyms and alternate phrases as possible helps to eliminate one of the most common pitfalls of adventure game design: the dreaded "guess the word" puzzle, wherein the player knows what she wants to do but is unable to find the "magic word" that matches the one chosen by the author for a particular object or action. TADS strives to prevent player frustration by allowing authors to provide a rich list of words to go along with every object. The parser has excellent support for pronouns and prepositions as well, helping players to construct commands which are very natural and intuitive.
TADS also allows the author a substantial amount of control over the "travel" verbs, which in text adventures generally consist of the compass directions plus up, down, in, and out. In the example above, the programmer has indicated that by traveling north from the mine, the player will reach another room which represents the entrance to the mine shaft. Presumably that room will have its south property set to point back to this room, so that the player will be able to travel freely in both directions, but this need not always be the case. Perhaps a cave-in will occur at some point, which will block that passageway and force the player to find another way to get in or out of the mine. All of an object's properties, including its keywords (nouns and adjectives), may be modified dynamically by the game to reflect changes in the game world.

TADS' object model is very powerful and provides programmers with a great deal of flexibility. TADS has implemented objects in a simple manner that is easy to use and understand. And, since the "gold nugget" item of our example inherits from the pre-defined class item, it already has most of the characteristics the player will expect of a game object — it may be picked up, dropped, thrown, carried, given away, and so on. Of course, the programmer might want to modify the default behaviors of some of these verbs, and TADS fully supports modification or replacement of its default behaviors. For example, giving the gold nugget to another character might cause that character to become the player's friend.

Unfortunately, it is this very process of overriding default behaviors and providing new ones that causes most of the problems for people who are trying to learn TADS (especially non-programmers). Because of the complexities of the English language, TADS models several different types of object interactions, each of which has its own syntax and each of which can be individually changed or left alone by the programmer. When done for a single object, this task may not seem so onerous. But considering that a typical adventure game of moderate length will have anywhere from 100 to 200 different objects for the player to interact with, providing unique and interesting behaviors for every object can quickly become a major headache, especially for the novice programmer or the programmer who is not used to the object-oriented paradigm. Managing the interactions between the
player and objects and between objects and other objects occupies almost all of the TADS programmer's time.

Let us suppose that we wanted to change our gold nugget so that when the player picks it up a cave-in occurs which seals the mine shaft and traps the player inside the mine. We might add the following code to the object definition for the gold nugget:

```plaintext
doTake( actor ) =
{
"As you lift the gold nugget, an ear-splitting CRACK comes from the nearest support timber. You stagger towards the exit, but the ground trembles and shakes beneath your feet, throwing you to your knees as a huge slab of rock breaks from the ceiling above you to crash across the passageway, cutting you off from the light of day. You cower, thinking that all is at an end, but the sudden cave-in stops as quickly as it started, leaving you whole, unharmed, and utterly trapped beneath the surface of the earth."

mine.north := nil;
pass doTake;
}
```

In this code, the player's action of taking the gold nugget has caused three events: the display of the text describing the cave-in, the removal of the northern exit from the mine, and finally a call to the parent class's `doTake` method so that the gold nugget is actually picked up (we have to give the player something for her trouble, after all!). Now, if the player should try to exit the mine by going north through the former passageway, she will
receive a message telling her that she cannot go that way. Clearly she will have to seek another exit, and thus we have introduced another plot element into the game. Perhaps we will introduce an added level of tension by making the air supply in the mine slowly run out, causing the player to suffocate if she does not quickly find an exit.

Notice that it was possible to refer to, and even change, the properties of the mine room object from inside a method of the gold nugget item. A key consideration when writing TADS programs is that all objects in TADS are global in nature; there is no support for information hiding, private members, and the like. Providing such support would be detrimental to the programmer anyway, because the number of object interactions in an adventure game is vast, and many objects might have effects upon one another that are best handled within the object causing the change, rather than trying to implement some kind of complicated object security routines that would only bog down the entire system.

In the second example, we modified the doTake method of our gold nugget to cause something other than the default behavior to occur. The method name is constructed using a standard format that cannot be changed by the programmer without throwing out the entire set of TADS base classes (generally not a desirable thing to do, although some advanced TADS programmers have in fact released their own complete replacement packages for the classes provided with TADS).

The doTake method tells what to do in the event that the gold nugget item is the direct object of the “Take” verb, so it is called doTake. If we wanted to specify what should happen when the gold nugget was the indirect object of a command involving the “Hit” verb (for example, the player might type HIT WALL WITH ROCK), then we would override the method ioHit. Additionally, TADS provides “verification” methods which allow an object to define whether or not it is even a legal target of the verb, before any action is taken. These methods have the prefix ver prepended, so a method to verify whether or not an object could be the indirect object of the “Shake” verb would be written verioShake. As you can see, method naming can become quite lengthy for large games, with their dozens of verbs and hundreds of objects.
These examples demonstrate that while it is very easy to create complex objects and to specify interactions between those objects in TADS, it can be a tedious and confusing process, especially for a person who has never programmed before. While the idea of overriding a method and then calling the parent's original method will at least be familiar to a C++ programmer, it could be overwhelming to a novice. And, since there is no other mechanism for providing "life" to objects, rooms, and characters, the user is forced to delve into the depths of the parser and the provided base classes to determine which methods must be overridden, and under what circumstances, and how best to go about it. Indirect objects present a whole other set of issues involving verification, the order in which methods are checked, and cascading effects which have not even been touched upon in this brief discussion.

One powerful aspect of TADS not mentioned previously is that it uses a virtual memory model in its runtime system, which means that the game files may be larger than the amount of memory that the computer has available. This ability to swap parts of the game in and out of memory as needed means that authors are not constrained by any machine limitations, either on the compiler end or on the interpreter end. Games written using TADS have absolutely no size limitations whatsoever, which was certainly not true of the old Infocom text adventures, most of which had to fit into 48K of memory.

Hopefully it is now apparent that while TADS is certainly a powerful language, and one which provides a substantial amount of support to the adventure game author, the presence of a good visual tool which would allow the author to automate much of this design process without having to memorize complex layers of objects and method naming schemes would speed up the design process and make it far more palatable to the novice. Let us now look at Inform, the other language supported by TA DA!, and see if it shares any of these characteristics of power combined with obfuscation which make the visual paradigm all the more appealing.
Chapter 4

A Look at Inform

This chapter contains a brief look at Inform, the second adventure game programming language which is supported by TA DA!. Just as with the preceding chapter on TADS, this chapter is not intended as either a tutorial or reference manual for the language, but rather is intended only to illustrate the fundamentals of the language and demonstrate some of the difficulties which make using a visual tool desirable.

The discussion here is based on the 6.04 release of the Inform compiler, and the 6/2 release of the library files. These are the most recent releases at the time of this writing, but the author — and a large number of devoted Inform users on the Internet — continue to provide updates, improvements, and suggestions regarding Inform on a constant basis. Compared to TADS, which was primarily the work of a single person, Inform has evolved in the years since its introduction in April 1993 thanks not only to Nelson but also to many devoted users who have contributed greatly to its development. For the most current release of Inform, as well as a number of publications relating to the language, the reader is encouraged to visit the official Inform site on the World Wide Web, which is located at

http://www.c1.cam.ac.uk/users/gdr11/inform

Where needed, this chapter will also point out some of the primary differences between TADS and Inform. The two languages come from very different design styles and philoso-
phies, and games written in one language are not easily converted to the other by hand. Having read both the material on TADS and this material on Inform, it will hopefully become apparent that a single tool to generate code for both languages would be both useful and appreciated by the adventure game community, and furthermore that such a tool would have to use some kind of higher-level abstraction to eliminate the major differences between the two programming languages. Designing games in a language-independent fashion is precisely what TA DA! attempts to do, of course.

4.1 Overview

Inform is another system that was developed for creating adventure games. It was created by Graham Nelson, who did not especially care for TADS. In addition, Nelson wanted to see if he could recreate the original Inform story file format and develop a system whose games would fit that format. By so doing, he reasoned, the games created by the system would be playable on any of the computers that Infocom originally supported, and indeed this is the case as both Inform and the games it creates may be ported to and used on any computer with an ANSI C compiler. Inform has enjoyed a rapid rise in popularity since its introduction, and the author has regularly provided new releases which have improved the power of both the compiler and the accompanying library files. Inform is also completely free, although authors of games using Inform are free to market their games as they see fit.

As with TADS, Inform consists of a core language compiler and associated header files that provide the actual classes and objects that define a base common to all adventure games. Unlike TADS, however, Inform is actually designed to be useful as a general-purpose programming language, and most of the game-specific material is contained in the library files. In fact, several authors have written non-game-related programs using Inform just to demonstrate these features of the language. However, Inform is really not suitable for this sort of work on a large scale, since it is an interpreted language rather than a compiled language. The Inform compiler takes source code and produces a p-code that is identical in format to the original Z-machine used by Infocom in their commercial products. (Infocom
published the specs for this format in an article in the IEEE Computer magazine). Inform also provides support for an “extended” Z-machine that allows for larger and more complex games than were possible using the original Infocom system.

Unlike TADS, Inform does not come with its own interpreter and run-time system. Instead, there are a large number freely available public domain interpreters that have been written by people over the years to play the old Infocom games. Since the Z-machine standard is entirely machine-independent, consisting of a set of bytecodes (much like a compiled Java program), Inform games may be played on any machine with a Z-code interpreter. Several interpreters have been written using strict ANSI C, so that just like the compiler itself the games can be played on virtually any platform. Interpreters have even been written that allow Inform games to be played on the Apple Newton personal digital assistant, and there is little doubt that as new platforms and hardware systems evolve, interpreters will be ported to these machines as well, meaning that a game written using Inform is likely to be playable for the foreseeable future of computing.

Inform also does not currently have a debugger. Graham Nelson has been working on an integrated debugger, which he currently calls “Infix”, but it is not finished at the time of this writing. Because TADS has a debugger, but Inform does not, adding in-line support to TA DA! for connecting to and using the debuggers was not possible. The possibility of providing a connection to the TADS debugger and merely leaving a “stub” for the future inclusion of support for Infix was considered, but discarded. My rationale for this decision was that if TA DA! is properly written, there should be no need for syntax-level debugging at all, because one of the tool’s primary goals is to generate bug-free code. The lack of a syntax-level debugger is a problem for some Inform programmers due to the complexities of the language.

By and large, Inform has become the system of choice for new adventure game authors, primarily because it is being actively supported and has a more vocal set of supporters than does TADS. Numerous tutorial games and utility libraries have been written for Inform, and new games are released on a fairly regular basis. Graham Nelson himself has written
two of the most popular modern text adventures, “Curses” and “Jigsaw”, both of which experienced a number of downloads which would easily have made them as popular as any of the games written by Infocom during its heyday had they been released during that time period when text adventures were commercially viable.

There is no doubt that once its nuances have been mastered, the Inform programmer is capable of creating games with every bit as much detail and power as the great text adventures of old. However, in terms of the parser and the size of the games which can be created, Inform does fall short of the mark set by TADS. Nevertheless, Inform’s parser is far more open than is that of TADS, which leaves open the possibility that individual programmers could improve it to the point where it met or exceeded the standards of TADS. Inform provides support for dynamic instantiation of objects, but the programmer must specify in advance what the maximum number of instances of a particular object will be. “Recycling” of used objects is possible, but Inform does not have the open-ended ability to create things on the fly which TADS provides.

Additionally, even with the new expanded Z8 implementation of the Z-machine (never done by Infocom, but it is a logical extension of their work), Inform games are limited to a fixed maximum size. While the Z3 story file (the standard used by Infocom) is very limited in scope, Inform by default creates story files using Infocom’s “Advanced” file format, called Z5. The Z5 format allows game file sizes of up to 256K. The Z8 format allows games of up to 512K in size, generally adequate for even the most gargantuan game. Authors with truly ambitious projects, however, might run afoul of Inform’s memory restrictions. A game requiring more than 512K would force its author to use TADS, because at present 512K is the absolute upper bound for the Inform compiler and all existing interpreters. Other than the total memory limitation, however, Inform provides authors with all of the power that they need to create games of arbitrary complexity and size. In practice, very few games ever exceed even 256K, so Inform’s size limits are not a serious drawback to the language.
4.2 The Language

Inform was designed to support the programming of adventure games, but it follows much of the design philosophy of general-purpose programming languages as well. Where TADS uses a hybrid of Pascal and C operators, Inform tends to be very C-like in its operators and function invocations, even to the point of requiring the appending of parentheses to a function call with no arguments. However, Inform diverges from C in many notable ways, most obviously the use of square brackets instead of curly braces to delimit routines, but going back to curly braces for blocks of code. Inform also provides support for some object-oriented programming techniques, though not nearly so many as TADS. Indeed, the author describes it as being both procedural and object-oriented, but it is much more the former in reality.

Inform's core language is quite small, consisting only of 28 statements and a variety of operators. The statements provide support for procedural programming constructs, such as \texttt{if..else}, \texttt{for}, \texttt{do}, and \texttt{while} loops, \texttt{break} and \texttt{continue} statements, a \texttt{jump} statement, a \texttt{switch} statement, full support for recursion, and supplies defined constants to represent boolean values for true and false. Inform does not explicitly type variables, supporting short integers, strings, and characters with implicit type conversions between the three as needed. While variables must be declared (unlike TADS, where they are simply created as needed), they do not have to be typed at declaration time. Inform is case-insensitive and limits identifiers to 32 characters in length.

Inform provides a wide variety of operators, including the standard arithmetic operators, modulo, and unary increment and decrement operators, as well as a rich supply of relational operators, including six special relational operators which apply only to objects and are used to determine attributes, properties, and inheritance of objects. Unlike TADS, which supports a large number of embedded string operations, Inform is somewhat more limited in its ability to construct strings on the fly and in fact provides an explicit \texttt{print} keyword for displaying strings to the screen (although its use for displaying text is optional at certain times). Inform does support expressions within strings, as well as embedded
special characters to represent whitespace.

Inform supports both global and local variables, although the number of globals is limited by the compiler. Inform provides statements which allow the program to exit gracefully, to exit less than gracefully, to format text output, to print both ASCII and other special characters, and (unlike TADS) supports special “text boxes” into which important text, such as quotations or clues, may be placed, and supports the ability to break the screen into multiple “windows”, which may display different things, such as a map of what the player has explored thus far in the game. So, while it seems that in some ways Inform is more limited than TADS, it uses what it has to good effect and provides some surprisingly powerful and unique features which are very useful in adventure games. Inform also supports switching between a monospace and regular-style fonts, as well as bold-face and underlining.

Inform provides support for arrays, although they are not able to be dynamically sized as they are in TADS. Inform also uses the C view of strings as arrays of characters. Inform takes the view of object hierarchies which is most convenient to adventure games: that is, objects have parents, children, and siblings. This arrangement is very useful for expressing concepts such as containment. The object tree itself is always directly accessible by the player, which is something that is not possible in TADS. Indeed, most of the lower-level library functions provided by Inform for traveling, picking objects up, and the like are basically nothing more than direct manipulations of the object tree. Programming in Inform requires the author to “get his hands dirty” and manipulate fundamental properties much more so than in TADS.

Inform provides a robust set of commands for manipulating objects and changing their parents, children, and siblings at run-time. Inform also allows for the creation of classes. Classes may define both public and private members (information hiding is not supported in TADS). The newest version of Inform supports multiple inheritance, which was not originally a part of the language but was added due to user demand. Inform permits objects to have what it calls properties, which may be numeric values, string values, and the like; and attributes, which are true-false in nature and are used to define the characteristics of
an object. Some of these attributes are defined by the library files, while others may be created by the programmer. For example, a door that is to be closed and locked would need to have the open attribute removed from it and the locked attribute given to it. While properties may not be dynamically added and removed, attributes may.

Because the parser is so openly specified and flexible, Inform games written in languages other than English have started to appear. Creating non-English games would be nearly impossible to do in TADS without substantially rewriting the TADS source code itself or making extensive use of the preparse functions supplied by TADS to map the foreign-language input to some kind of phrasing that would match the English subject-verb conventions. In other words, for games which are to be written in a language other than English, Inform is about the only choice. Fortunately its support for this sort of thing is very robust, including codes for the accent and special characters of a great many languages, and it is easy to add new codes and characters if a font is available which contains those characters.

In conclusion, Inform is a powerful programming language which, like TADS, was designed to support the writing of adventure games. However, Inform attacks the problem from a more generic and procedural standpoint, whereas TADS approaches it from an object-oriented and game-specific design philosophy. Both languages utilize large library files which contain the class definitions and base objects which are then modified by the user to create games. Inform's objects are very much like C structures, while TADS' objects tend to be more like C++ classes.

4.3 Writing a Game Using Inform

Just as with TADS, the Inform programmer must define all of the objects, rooms, and characters which make up the game world. As would be expected, the syntax for doing so is quite different in each of the two languages. Let's construct the same example of a gold mine with a nugget on the ground which we created in the previous chapter, this time using Inform:
Object Mine "Gold Mine"

with description

"You’re standing in an abandoned gold mine. Ominous creaks
come from the support timbers which surround you on all sides,
and the air is thick with heavy dust from the thousands of
tons of rock. ",

n_to mineEntrance,

has light;

Object -> nugget "gold nugget"

with name "gold" "golden" "nugget"

initial

"There’s a huge gold nugget sitting on the ground sparkling
in the dim light. ",

description

"Your eyes grow large with avarice as you regard this
fist-sized chunk of solid ore. ";

The most obvious difference between this code and the corresponding TADS code that we saw in the previous chapter is that while in TADS we provided only one description of the object (under the ldesc property), Inform allows us to provide two: the initial and description properties. The first is used only when the object is seen by the player, before it has been taken or manipulated. The second is what the player sees when he or she actually makes a more detailed examination of the object itself. The ability to provide an initial description is so useful and so popular among game authors that TA DA! provides a workaround which adds this feature to TADS, even though it is not supported by default. Generally I have avoided tinkering with the internal workings of either TADS or Inform,
but in those cases where one language provides an extremely powerful and popular feature that the other language does not provide, I have done so to ensure that games created using TA DA! will behave the same no matter what language is the ultimate target.

Also, notice that instead of explicitly stating that the gold nugget is located in the mine, Inform uses the arrow (→) to say that the object being defined is a child object of the object which was just defined. Another way to indicate the relationship between the nugget and the mine would have been to explicitly specify the nugget's parent at the end of its object declaration, as follows:

Object nugget "gold nugget" Mine

Defining parent-child relationships between objects is a good example of a characteristic of Inform that is frustrating to many programmers: there are generally numerous ways to accomplish the same task, some of which are confusing and some of which are not. As with C, the readability of an Inform program can be very high or very low, depending on how the programmer chose to structure the code and which operators were used.

Now, let us add the code which causes a cave-in to occur when the nugget is taken. Inform does not subscribe to the same model that TADS does when it comes to object interactions. Where TADS provides the opportunity for the programmer to write both "validation" and "action" methods for each object as direct and (where appropriate) indirect object, Inform deals only with the direct objects and allows for three possible stages of intervention: Before, which serves much the same purpose as the TADS ver methods, allowing your object to prevent the action from occurring; During, which is when the action itself is actually being carried out; and After, which allows your object to react to what just happened, and possibly even alter the outcome. All of these are checked for an object before any results are returned back to the player, so Inform objects have just as much power as TADS objects to catch, alter, and block events from occurring and to cause changes in other objects.

Of course, it would be difficult for the parser to function without allowing some method of checking indirect object behaviors as well as direct object behaviors, and so Inform
provides a mechanism for doing this as well, though a system of "imaginary actions" which are sent to the indirect object at the same time as the "real actions" are being sent to the direct object. The question of which of the two takes precedence in the event that both want to respond to the message can be tricky and is a source of some confusion to Inform programmers. Adding new indirect object methods is supported, but requires an extra level of work from the Inform programmer which is not required of the TADS programmer. Indeed, Graham Nelson has several "exercises" in the Inform Designer's Manual which suggest that adding a TADS-style validation stage in action processing would be a desirable trait. Fortunately, TA DA! abstracts the process of defining actions to a level where it is still possible to generate appropriate code for both direct and indirect objects for both languages.

Although it is more convoluted than the TADS mechanism for responding to actions, the Inform scheme actually allows objects to react more than once to the same action, something which is not possible in TADS. Although the ability to respond multiple times to the same event may seem like a trivial matter, sometimes it is desirable to have objects that print multiple "status" messages as they are used. For example, a crank being turned might want to print a message that the crank starts to turn, followed by a message that it gets stuck halfway there. Once whatever is blocking the crank is removed, it would then print the same message about starting to turn, but a new message saying that it turns all of the way and a secret door opens (or whatever else the author desires). All these behaviors can be encoded in the same object, using before, during, and after routines for the crank object. Inform's message passing scheme is very non-intuitive, however, and causes a great many problems to people trying to learn the language (and sometimes even to experienced programmers). TA DA! attempts to take the confusion out by allowing the user to specify object behaviors in a simple and direct fashion and then maps the result to the correct Inform routine.

At any rate, we wish to have the nugget respond to being taken by triggering a catastrophic cave-in. Since we do want the nugget to actually be taken, we'll place the code for
this into an after block, thus causing it to be delayed until after the object has actually been taken. We modify the gold nugget by giving it a new routine:

```plaintext
after
[; Take: { Mine.n_to = NULL;
    "As you lift the gold nugget, an ear-splitting CRACK
    comes from the nearest support timber. You stagger
    towards the exit...";
 }
],
```

(The text of the cave-in has been abbreviated to save space since it is the same as the previous chapter). Notice that in order to cut off the exit to the mine entrance, it was only necessary to set its north direction property to NULL, which is a pre-defined Inform constant. Also notice that unlike in TADS, where each verb has its own set of methods, Inform lumps all of the actions together inside a single routine. Thus, if we wanted to also have the gold nugget do something special when it was thrown, we would place the code inside the after routine we have just written instead of writing a separate method. Also, it is unnecessary to pass control back to the inherited Take method of the library files, because we are performing this action in an after block. Placing the code after the action processing is complete means that the taking has already happened and we are merely responding to it.

Just as TADS allows us to provide objects which are treated as “parts” of a larger object, Inform also permits the creation of special kinds of objects which are considered to be “attached” to something else and move with it even though they are not explicitly mentioned. (The concept of attachment is different from the behavior of containers, although some of the ideas are the same). We could create several objects of class scenery which would represent things that the player could look at, but not touch or manipulate.

The creation of descriptive but non-functional objects would be useful in providing a greater level of detail for the room description. The player could examine the support
timbers or the surrounding rocks and even smell the dust. Perhaps in one of these “sub-object” descriptions we could embed a vital clue which gives the player a hint about how to escape from the mine after the cave-in. For example, smelling the dust after the rocks have collapsed might give the player a whiff of fresh air, which could be traced to a previously unseen crevice along one wall. Climbing through the crevice would lead the player to another set of rooms, at the end of which was an exit to the surface. In such ways is a truly interesting game born.

Although the examples in this chapter and the previous chapter are simple in nature, they give a description of the fundamental ways in which authors can cause simple, or even complex, actions to occur in response to what the player does. While TADS and Inform have different views of the world and very different syntax, they both provide the same fundamental constructs. By picking out the key elements of these constructs and abstracting them to a more generic level, TA DA! attempts to allow the user to design something in a fashion which can then be translated directly into both languages without losing any of the functionality or power of the original design. In the next chapter we shall define an abstraction for adventure games which allows language-independent design to take place.

Readers who wish to know more about TADS or Inform, or to download either system or play sample games created with these systems, are referred to the wonderful Interactive Fiction Archive, which is maintained at the time of this writing by Volker Blasius at the following FTP site:

ftp.gmd.de/if-archive

This site contains a wealth of tutorials, sample code and games, and also has an archive of the discussions on the various Usenet newsgroups where Inform, TADS, and other systems are discussed on a regular basis. Anyone wishing to know more about text adventures, past and present, is encouraged to visit this site and sample its many offerings.
Chapter 5

The Implementation

This chapter contains a detailed look at TA DA! itself, including a complete specification of the abstraction which I call a “meta-grammar” and which is used both for the design and for the code generation. We look at TA DA! and its unique position as both a generic compiler front-end as well as a multi-targeting compiler unto itself. We conclude with a discussion of the human-computer interface issues which were addressed when designing the user interface and how the interface reflects the general adventure game design process that authors have been using since the beginning of the genre.

5.1 An Abstract Game?

In order to achieve the stated objective of allowing the user to design the game without giving thought to the underlying code that will be generated, it became necessary to design a system whereby the game itself could be expressed in an abstract form. If it were possible to “pull out” the elements that are common to all games, and somehow generalize them into a sort of “ideal game”, then the design tool’s task would be to facilitate the development of games according to that abstraction or ideal without forcing the designer to worry about how the abstraction will “come down” to the level of the code. Providing this ideal representation is exactly what TA DA! attempts to do with its “meta-grammar” or “meta-language”.

39
5.2 Overview

In a traditional compiler, the *grammar* which defines a language is expressed as a series of production rules. Each rule defines terminals and non-terminals, and the non-terminals are the subject of later production rules which define them as well. Eventually all non-terminals are replaced by terminals, and the lexical analyzer and parser then work together to tokenize the input stream and match the tokens to the production rules, thus verifying that the input comprises a valid program in the language. TA DA!, although it does have a back-end similar to that of a standard compiler, does not have this independence of grammar from front-end. Instead, the user interface and the production rules themselves are sort of tied up together, so that it becomes impossible to create an "illegal" game. Although the user has complete control over the design, because of the tools and components provided and the legal ways in which they may be linked together, TA DA! does not produce games which will not compile in the target languages.

TA DA! manages to produce clean code for every possible design because the game design itself is expressed through the user interface, which allows only valid steps to be taken at any point in the design. For example, if the user is coding a room, only information about that room will be able to be entered through the interface. An attempt to suddenly start editing an object without first closing the room edit session will not be allowed. The user must finish editing the room (or close the edit box, which implies that the edit is finished) before he or she can move on to something else. Because of the way the TA DA! code generator is structured, this user interface "enforcement" ensures that only legal programs will eventually be created. Providing complete and accurate contextual information to the user at all times avoids one of the major hassles often associated with learning to use a new language — cryptic and unhelpful error messages which the novice programmer must pore over in attempting to determine why the piece of code that he or she just spent two hours painstakingly crafting is refusing to compile.

Once the design has been created and approved by the user, then the code generator takes over. Throughout the design process, the program has been building a large amount
of data expressed in an abstract form which represents the game design. Using templates in
the target languages, code is then generated and customized to fit the specifications defined
by the user during the design process. The abstract form used by TA DA! is similar in some
ways to the abstract syntax tree used by many compilers as an intermediate form prior to
machine code generation.

So TA DA! is like a very complex front-end to a compiler. Or, more appropriately, it is
like a very complex front-end to a set of parallel compilers, since the code generation process
for the target languages is simultaneous and independent. On the other hand, because of
the abstractions involved and the structuring of the design within the program itself, TA
DA! is also like a complete compiler unto itself. It goes through many of the same steps as a
compiler and produces output which is then fed to the “real” compilers so the compilation
process may be repeated. This two-step architecture allows for a high level of abstraction
in the visual tool, freeing the game designer to concentrate on the game itself rather than
on the details.

5.3 The Object Model

The fundamental viewpoint of TA DA! is that the elements which make up an adventure
game are best expressed in terms of objects. (In this case, the term “object” refers to a
collection of encapsulated properties and methods, rather than to physical objects such as
one might find in the real world.) TA DA! supports properties and methods without ever
explicitly referring to them as such. Furthermore, the other aspects of the object model are
supported as well:

1. State of an object — the values of an object’s properties make up its state. TA DA!
supports both static and dynamic changes to an object’s state. Changes may be made
by the designer at compile time, or at runtime through the mechanisms provided by
the object’s methods.
2. Persistence — Although TADS supports dynamic creation and destruction of objects, Inform does not, and has no way to “fake” it. Since this feature of TADS is almost never used in practice, it is not supported in TA DA! either. Therefore all objects are persistent and permanent. They may, however, be removed from play by making them inaccessible to the player; this is functionally equivalent to dynamically destroying them.

3. Encapsulation — The design of a game using TA DA! is a very encapsulated process. When the user is editing a room, for example, only properties and methods relating to that room are accessible to be worked on. When the user wants to link one or more objects together, she may easily do so, but each object is consistent unto itself.

4. Inheritance — While TA DA! provides certain “base” object components that can be used to create complete games, it is very unlikely that the user would not want to define some new components or modify the existing ones. Therefore TA DA!, like its two target languages, provides strong support for inheritance and subclassing.

Furthermore, there are some standard object operations that are also supported by this model. Concepts such as containment, association, and use are all well-defined by the abstraction and implemented via the user interface of TA DA!

5.4 The Abstraction

The basic components of the TA DA! abstract game can be summarized in four parts. Each of these parts has many sub-parts which are distinct, but an overview of each of the parts is as follows:

5.4.1 Blocks

A block is a fundamental unit: an object. Blocks exist to represent rooms, items, characters, and other physical things that exist within the game world. Blocks also exist to cover non-corporeal aspects such as conversational topics for characters, the compass directions in
which the player can move, and exits from rooms. There are blocks to represent gameplay-related items which are part of the design but which are not obvious to the player as well, such as commands and extensions to the standard parser (allowing for more verbs to be recognized, for example, requires one block per verb to be added). Blocks are the most basic and most important aspect of the abstract game. Everything will be expressed as a block in one fashion or another.

5.4.2 Attributes

An attribute does not exist on its own in the abstraction, but rather as a part of a block. Blocks can have many different attributes, depending on what the block represents. For example, a block which represents a room has attributes for the name, description, any special flags which apply to the room (for example, whether it is in darkness or underwater), and so forth. Objects have attributes that tell which room they are located in, among other things. Attributes make each block distinct from every other block of the same fundamental type (although every block does have a unique identifier).

5.4.3 Triggers

A trigger is a method that enables an object to respond to actions directed at it. For example, a bomb could have a trigger that says that when it is thrown, it explodes. A trigger can be activated only when the object is a direct object, only when the object is activated as an indirect object (for example, HIT BALL WITH BAT would check for triggers on the ball as a direct object and on the bat as an indirect object), or both. Triggers are the primary methods that are used to interact with objects.

5.4.4 Traps

A trap allows an object to catch messages that were not intended for that object. Traps are often associated with characters or other barriers. For example, an ogre who is guarding a treasure chest would have a trap that said that any attempt to manipulate the chest would
be intercepted by the ogre. The player did not intend to interact with the ogre, but the ogre responded to the player's action because of the trap. If the player wants to examine the treasure chest, then the ogre will have to be dealt with in some fashion. Traps are one way in which puzzles may be constructed in adventure games.

5.4.5 Conflicts Between Triggers and Traps

Certain kinds of events that occur regularly in adventure games fall under more than one of these categories or do not clearly fall under any category. For example, an exit is a block which is contained within a room. It represents, as one would expect, a path that leaves that room and goes to another room. Sometimes the designer wishes to have a message printed when the player goes through a particular exit. Should this be a trap or a trigger? The player did not intend to interact with the exit itself, merely to pass through it, so it could be represented as a trap. On the other hand, the message is triggered by the player's action, so this situation could also be represented by a trigger. In the case where an item is ambiguous, the system uses the least "active" methodology. In this case, a trigger is less obtrusive than a trap because it requires fewer messages to be examined and passed on. Therefore special events which occur when the player attempts to use or uses an exit are modeled as triggers rather than traps.

Furthermore, certain events might cause both a trigger and a trap to be activated. In the example with the ogre and the treasure chest, the player might have given the command \texttt{OPEN CHEST}. Suppose that the chest has a trigger on it that says it should explode when opened. The ogre has a trap that says to intercept any attempt to manipulate the chest. If both of these are allowed to occur, the ogre would block the player from opening the chest and then the chest would explode, which is not realistic since the player didn't actually get to touch the chest. Therefore when an event activates multiple traps or triggers, the traps are processed first and any "surviving" messages are then passed on to the appropriate triggers. The basic meaning is that, as the name implies, a trap can literally "catch" a message and prevent it from reaching its original destination. This behavior is consistent
with the behavior players expect of adventure games, and the system works quite well in practice.

5.4.6 Hooks

A hook is the method of communication between objects. It allows two objects to be linked so that they can pass specific messages between one another. For example, a lever in one room is supposed to raise a gate in an entirely different room when the lever is raised. These two objects are created independently of each other and then a hook is created to link them so that when the lever is acted upon (raised), a message is also sent to the gate that it should ascend. Hooks can operate either unidirectionally (as in the case of the lever and gate) or bidirectionally (in which case changes in the state of either object can affect the other object’s state as well.) Hooks are one of the ways things are made to “happen” in response to the player’s actions. A hook is much like a trigger that is activated in response to another trigger on a different object.

5.4.7 Fuses

A fuse is a turn-based timer which can be activated by a trigger or hook. It basically acts as a delayed trigger which “goes off” after a specified number of turns have elapsed. (A turn in interactive fiction refers to a single command entered by the user and the game’s response to that command.) For example, the designer of a spy game might want to set the game in a building that contains a bomb. The player’s job is to find and defuse the bomb. At the beginning of the game, the designer would activate a fuse, and after the specified number of turns had elapsed, if the player had not found and disarmed the bomb then it would activate a trigger which would cause the building to explode and most likely kill the player. Fuses can be either long or short in duration. Shorter fuses are actually more common than longer fuses, and are used in almost every adventure game. Fuses are used to provide believability to objects that should “run out” or be used up after a specified period of time.
5.4.8 Daemons

A *daemon* in text adventure design is very much like its namesake in operating systems design: a background process that does certain work on a periodic basis. In the case of interactive fiction, daemons are functions which are called every turn. Daemons are used to set things in motion which should always affect the player. The most common daemons are used to cause the player to become progressively more hungry and thirsty as the game progresses. If she eats and drinks, then the daemons are reset, and the process of becoming hungry and thirsty starts again. Daemons are also used to provide functions to certain kinds of non-player characters (NPCs), so that those characters can take actions in the game world even when the player is not in the same room as the NPC. Daemons and fuses have much in common, but are separate entities in the game abstraction because they are used in very different ways and at different times by game authors.

5.5 Implementing the Abstraction

The most critical part of creating a game using TA DA!, then, is not the formulation of the abstraction but rather the translation of this abstraction into actual game code. The translation is accomplished through a mechanism that has much in common with both templates (used by C++) and components (used by Java). This approach guarantees both correctness of the generated code and complete compliance with the design, as well as providing for extensibility — both of the languages supported by the system, as well as for the system itself and the kinds of games which may be designed and the operations which may be supported.

Although I have spoken at great length about TADS and Inform in this document, TA DA! itself does not contain any code that is based on anything to do with either of those two languages. They are merely the two languages which have been chosen for this project because they are the two most commonly used adventure programming languages and because their dissimilarity illustrates the power of the approach taken by TA DA!
towards abstracting the game and generating code from a single design. In fact it is possible to add support for virtually any language to TA DA! by following the process outlined below.

Using the complete specification for the abstract “meta-grammar” created for TA DA!, I was able to define a set of language templates for both TADS and Inform. Each template defines a generic implementation in that language for the block or other feature specified. For example, one of the most fundamental block types in adventure games is that of the room. The basic room block in TA DA! has only two attributes — name and description. The generic template for this block in TADS, then, looks as follows (some of the features of the template have been removed to simplify the discussion):

```
TEMPLATE BLOCK ROOM
ID : room
    sdesc = QUOTED_STRING
    ldesc = QUOTED_STRING
;
```

The items in all capital letters are part of the template definition syntax; the items in mixed case are actual code in the target language. In this case `sdesc` is TADS syntax for “short description,” or the name of the room which will be printed on the status line at the top of the screen when the player is in that room. Likewise, `ldesc` is TADS syntax for “long description,” the lengthy message that is printed when the player enters the room and looks around. Rooms can have other attributes, of course, but the ones shown here are mandatory. A discussion of other attributes and of inheritance or derived classes is deferred for now to concentrate on the template-code relation itself.

So if the user has created a room using TA DA! (and there must be at least one room in order to have a valid game, or else there will be no location for the action to take place), then this template will be filled in at code generation time with the appropriate values as defined by the user. For example, suppose that the user was creating a cave. The room might be named “cave,” with the short description “A Dark Cave” and the long description “You are standing at the entrance to a deep, dark cave which seems to have been dug by
some huge creature. The cave stretches away below you." The template would then wind up looking as follows when code was generated in TADS:

cave : room
   sdesc = "A Dark Cave"
   ldesc = "You are standing at the entrance to a deep, dark cave
   which seems to have been dug out by some huge creature. The cave stretches away below you."
;

Note that the semicolon at the end is part of TADS syntax to signify the end of an object (in this case, a room) and is not part of the template definition itself. It is part of the code. This same template could also be written for Inform. It would look as follows:

TEMPLATE BLOCK ROOM
Object ID QUOTED_STRING
   with description
      QUOTED_STRING,
   has light;

Again, the semicolon at the end is part of the language syntax, as is the comma after the description. The code generation module intelligently matches the parameters provided by the user in the design stage with the items needed by the templates. ID and QUOTED_STRING are special markers used by the templates to identify key parts which must be filled in with user data. This template, when complete (using the cave example above) would appear as follows:

Object cave "A Dark Cave"
   with description
      "You are standing at the entrance to a deep, dark cave which seems to have been dug out by some huge"
creature. The cave stretches away below you."

The mapping from design to both languages is obvious and intuitive in many cases (as it is here), but in other cases it is not so easy. We shall examine some of these cases and their solutions in later sections. For now it is important to notice the key aspects of the system. By using templates, good code is always generated, free of user errors (assuming the templates have been validated to begin with). The user is able to enter the description, name, and other information in any order and free of concerns about how or where they will appear in the final code.

The system is very flexible. Should the specifications of one or both languages ever change, only the affected templates need to be rewritten. Should a better way of doing something be developed, the code in the templates can be revised or replaced without affecting the design itself. Should it become desirable for other languages to be added to TA DA!'s targets, adding support for them requires no changes in the program code itself. All that must be done is that a set of templates for the language must be written and then TA DA! told about the existence of the new target and its template files through the configuration and import utilities provided with the software.

The templates themselves are ASCII files and may be created using any text editor. Writing a “template editor” package would be a useful addition to the software. One drawback is that the system does not have a mechanism for adding new kinds of templates or for importing user-defined templates. Both of these would prove extremely valuable extensions to the system, much like the ability in Delphi to create “custom components” which then become a registered part of the system and may be freely exchanged between users. Developing a medium of exchange for adventure game components would certainly be a major enhancement to the development environment.

Overall, TA DA! is a very robust system that takes a different approach to the visual design paradigm. It is not at all like most contemporary visual tools, which are restricted to user interface design instead of code creation. Since the user interface for a text adventure
is standardized and very simple, TA DA! does not have to worry about this aspect of game design, meaning that the user doesn't have to worry about it either. The author is free to concentrate on the content of the game, rather than on its presentation. Let us now examine how an author goes about creating that content using TA DA! and compare it to the traditional adventure game design process.

5.6 The Design Process

The basic premise underlying TA DA!'s implementation is that designing a text adventure game is not like writing either a structured or an object-oriented program. The process requires significant thought and time; one cannot just sit down (even with a dedicated adventure game programming language such as TADS or Inform) and begin writing code. The user interface of TA DA! is modeled upon the pencil-and-paper design process that most adventure game authors, including this one, go through when creating a game (no matter what the implementation language). This process has basically four stages: designing the overall story, drawing the map, adding the objects and characters, and defining the interactions between objects, characters, the player, and the game world. Let us examine each of these four stages and see to what extent each is modeled and supported by TA DA!

5.6.1 The Plot

Interactive fiction games are more than just a series of player commands and computer responses; ideally, they tell a story, wherein the player becomes the protagonist and works his or her way through all of the plot elements created by the author to reach some kind of a resolution. Most adventure games have at least one "good" ending, in which the player successfully overcomes all of the obstacles and achieves victory, and numerous "bad" endings, which end with the player's goals unfulfilled and possibly even with the death of the player's game avatar. Other adventure games do not draw such clear lines, and provide instead a series of experiences and interactions which are not goal-directed at all. The player "wins" by simply seeing everything that the author has created.
For plot-driven games, most of the same genres that apply to paper fiction works are represented. There have been high fantasy, science fiction, western, horror, even modern “real-world” games. Some games cast the player as the hero or heroine; others cast the player as the “bad guy” who actively works to achieve nefarious ends. In any event, the style of writing, kinds of rooms, objects, characters, and puzzles, and the overall tone and pacing of the game are all dependent on the author’s choice of setting and plot. Unfortunately, this area is one in which TA DA! is completely unable to help the novice game author, because it is only able to help with the design of the actual workings of the game program.

There are many resources for new game authors available on the World Wide Web and the Internet at large. It is true that most people do not decide simply to write adventure games at random, however; generally a person plays a game (or many games) and then has an idea for a game of her own. It is the desire to see this game become reality which motivates people to want to learn to use these adventure game programming systems, and the reality that these systems do not allow them to jump right and make their vision of the perfect game come to life immediately often causes them to give up before ever really getting started. Hopefully, TA DA! will allow some of these would-be authors to get working games started quickly and overcome that initial block. For those who come without an idea, however, the only resources TA DA! provides are the sample game and a tutorial on writing adventures which will hopefully spur some creative thought. Since most would-be authors are inspired by games that they have already played, or by a book, play, or movie they have seen, coming up with the story line is generally not a problem.

5.6.2 The Map

The fundamental unit of an adventure game is the object, but the fundamental unit of interaction with the player is the room. Rooms are where everything takes place. A room might represent a space of any size, from the smallest closet to a forest hundreds of miles wide. Scale is an important element of adventure games; many games take place over large areas of a world, and so some of the rooms will be very small in scale, representing individual
buildings and even chambers within those buildings, while other rooms will represent large tracts of land. Indeed, it is not uncommon for a game which has two cities connected by a road to represent each of the cities by tens, even hundreds of rooms, while the road connecting them — which is much larger and longer than any city street — is represented by only a few rooms. The reason for compressing the representation of the much larger highway, and expanding the representation of the two cities, is simply to keep the player interested by her surroundings. The player will not have much fun if she has to wade through five thousand identical rooms which say You are on the King's Highway. The road stretches east and west from here. just because the author wanted a consistent scale.

Adventure game designers are limited in the ways that they can connect rooms, because most parsers handle traveling in terms of the eight compass directions (north, northeast, east, and so on) plus up, down, in, and out. Thus, the technique of designing a game's map by using a rectangular grid of rooms came into vogue, started by Infocom. (The map for the original Adventure was actually based on the real map of Mammoth Cave, Kentucky, and did not use the compass directions. Instead, players would provide the name of the place they wanted to go — for example, a player could type go river and be taken to the river. This method of travel is generally confusing to players because it becomes difficult to form a spatial picture of the game world and is not used in modern games).

On this rectangular grid rooms are represented using boxes. Boxes are connected to one another using straight lines to represent exits between rooms. As we have seen, a connection between two rooms actually requires two pieces of code, one for each end of the connection in each of the two rooms. TA DA! handles this seamlessly, meaning that the user only has to worry about where to place the connections. Rooms are laid out with north at the top of the page. Representing up and down is more problematic, and is usually accomplished by having several pieces of paper, one for each “level” or “floor” of the map. For example, in a palace with three floors and a dungeon, four pieces of paper would be used, and the transitions between pages are labeled in the appropriate places. In cases where there is only
one room on a higher level (for example, a tree house in the middle of the front yard) the exit is simply marked with the letters u and d on the map to indicate that it is an ascending or descending exit rather than a compass exit.

Through this process, it is possible to completely specify the map for the entire game. Scale is not a problem using this method, since the author will provide a sense of the size of each room through the descriptions that he or she writes for the player to see. Two rooms identical in size on the map might be wildly different in game scale; but since all rooms will ultimately be the same to the compiler, it is more convenient to have them all be the same size on the map as well. Representing larger rooms by larger boxes generally does not provide any more information to the designer and only forces her to draw larger maps.

Each room is generally given a name. These names serve to identify the room in the mind of the author (and perhaps the player as well). In TADS, the property sdesc of a room stands for “short description,” which is generally just the name of the room. Inform has a similar method for providing a short, three or four word summary of what a room represents. The rooms’ names are generally written inside the boxes on the map, thus providing a picture of the entire game world and all of the ways to travel from one place to another at a glance.

This overview of the map also helps suggest to the author some places where important events might occur and which exits and entrances might be initially denied to the player until key actions have been taken or puzzles have been solved. By making certain parts of the map inaccessible at the beginning of the story, the author is able to create the sense that the world is unfolding before the player through her actions and explorations. Some areas might not become accessible until the very end of the game — for example, if the plot of the game is that the player is a thief trying to break into the Baron’s treasure room, then it stands to reason that getting into the treasure room will be very difficult (or the game has no challenge); thus this room might remain inaccessible until the very end of the game, when the player triumphantly makes off with a bulging sack of gold coins.

Just as normal exits are represented on the map with solid lines, then, the author has a
device to represent areas which are inaccessible or whose exits may come and go. Dashed or dotted lines are used to mark exits which only open after a certain action has been completed or a particular puzzle has been solved. Locked doors are also represented in this fashion, since until the key has been found and the door unlocked, the exit cannot be used. Often these dashed lines on the map are an indication of some kind of puzzle or problem that the player must solve before entrance may be gained. For example, there might be a bridge between two rooms representing the two sides of a bottomless chasm. A troll on the bridge demands payment before the player can cross. Until the player has found some way to satisfy the troll, the troll will not permit the player to get to the other side of the chasm. Once satisfied, the troll allows the player free passage in both directions — or perhaps it demands payment each time the bridge is to be used, creating an entirely different obstacle whereby the player must be sure to complete all of the tasks on one side before going to the other side since she has only enough money to cross the bridge once.

Another type of obstacle that can be indicated on the map is the one-way exit — an exit which the player can go through from one direction but is closed to the return trip. For example, the metal detector at an airport is a one-way exit. The player can go through (if no metal objects are being carried) but once through cannot go back for security reasons. One-way exits are shown on the map via solid lines with an arrowhead. These kinds of exits are very useful to the author because they provide a way to “force” or “trap” a player into a particular set of rooms where something important is about to occur or where some goal is yet unfinished. Other times one-way exits are used to model mazes, a common adventure game theme. The original Adventure had a maze, but it was Zork’s maze that made the idea famous in fantasy adventure games, with its description You are in a maze of twisty little passages, all alike. Mazes provide an entirely different sort of obstacle to the player with their switch-backs, loops, and one-way exits.

Clearly, then, the map is an integral part of the game. Indeed, the hint books for the original Infocom games were popular because they included maps of the game world along with the hints and clues. Making a map is a useful skill for any adventure game player to
learn because it allows the player to quickly get back to any place he or she has visited, and it shows areas where some exploration might be in order. "Draw a map" was one of the hints which every game player knew by heart, along with "don't be afraid to try strange things" and "take everything that isn't nailed down."

Because drawing the map does so much for the game author and contributes to the game in so many ways, it is the central part of the TA DA! design process. The majority of the screen is taken up by a map of the game, which the author can rotate, pan, and zoom. Rooms are added to the map by clicking on a button, and then clicking on the map to place the new room. Double-clicking on a room brings up an editing screen where the author can edit the room's name, description, and key attributes (for example, the presence of light, meaning that the player doesn't need a light source to see what is in the room).

A word about the room descriptions is in order. When a player enters a room for the first time, she is greeted with a description of the room. The description of the gold mine in our examples from the TADS and Inform chapters is an example of a small room description. Good room descriptions provide atmosphere, involve several of the senses, and in general try to create as vivid a depiction of the scene as possible. Writing room descriptions is an art, not a science, and it is here that many adventure games fail. Engaging the player in a text adventure requires the ability to write descriptions that enable the player to achieve the "suspension of disbelief" that in turn causes her to forget that she is playing a computer game and instead transports her to realms of the author's creation. While TA DA! provides an excellent editor and has a good tutorial on creating rooms, it cannot help with this most creative aspect of room design. Designing good rooms is difficult, and is a perfect example of the ways in which game design can be enhanced and assisted, but perhaps never fully automated. Nevertheless, by relieving the author of the burden of writing the program code, she is freed to concentrate on making the descriptions of rooms (and objects, characters, and events) as engaging as possible.

To place an exit between rooms, the player need only click on the first room and then click on the second, and TA DA! draws the exit between them. By then clicking on the
line representing the connection, the author is able to edit the properties of the exit itself (for example, whether or not it is a one-way exit or if there is a door present). Doors are represented on the map, as are one-way, special, and looping exits (exits which return to the same room from which they left). The only restriction is that there cannot be more than one exit leaving a room in a particular direction (this would be impossible for the programming languages to handle). So a fork in the road must be represented as “running to the northeast and southeast,” for example, rather than “splitting to the east.”

Rooms may be dragged around the map at will, and the program automatically updates the exits as rooms are moved to keep the map as “clean” as possible. Different maps may be used to represent multiple levels, or up and down markers may be placed on the main map. The map window automatically splits to show both levels when an up-down connection is being edited. Implementing a three-dimensional, rotatable map would be a very desirable addition to the project.

In summary, TA DA! provides the author with complete control over the game map. The goal is to make working on the game design on the computer indistinguishable from the pencil-and-paper design. To this end, beta testers (both experienced and new game authors) were solicited to provide feedback on the user interface and suggestions for improvement of the design process. As many of their comments and ideas as possible were incorporated into the final product. Many more of their comments and ideas are on the list of additions and revisions for the next version, which goes to show that the evolution of TA DA! is not yet complete. However, all agreed that the current system provides an easy-to-use and intuitive mechanism for quickly creating the map of an adventure game. And with TA DA!’s auto-code-generation capabilities, the skeleton of a game – which otherwise might have taken hours or days to program in one language by hand and then tediously ported to the other language – can be created in both TADS and Inform in minutes.
5.6.3 The Objects

Having laid out the map and decided what events will take place where, the author must now create the physical world with which the player will interact. It is not enough to have rooms to walk around in if there is nothing for the player to do. The gold nugget in our example from the previous chapters is an object (albeit a very simple one). Most adventure games have anywhere from 200 to 2,000 different objects for the player to interact with. These range in complexity from decorative (a mosaic on the wall) to functional (a mosaic on the wall with a secret panel), from immovable (a gigantic statue of a pagan deity) to the portable (the ruby in that statue’s eye), and from the vast (a Sherman tank) to the microscopic (a computer virus). In short, while it is the rooms that the player will experience, it is the objects that the player will use.

Indeed, many adventure games — including some of the classics from Infocom, as well as the original Adventure — are based on a plot no more complicated that the acquisition of a number of different objects (the Twenty Treasures of Zork ranged from a solid bar of platinum to an ancient Egyptian sarcophagus to Neptune’s trident. The player won the game when all the treasures had been found and were safely tucked away in his trophy case at home). Creating interesting as well as useful objects is a challenge to every game author, and TA DA! attempts to make this process as painless as possible.

A new object is created by clicking on a button. The author must then give the object a name, and specify all of the words (nouns and adjectives) the player will be able to use to refer to the object. The object is usually given a description, and its starting location is specified if it is an independent object. If the object is a decoration or part of another object (a dial on a television, for example) then its relationship to its parent object is easily specified by choosing the parent object from a drop-down list of rooms or objects. Thanks to the built-in libraries of both TADS and Inform, it is not necessary for either the author or TA DA! to specify the basic behaviors that are common to all objects. Basic and complex attributes (such as whether or not the object is a light source, a container, transparent, may be dynamically duplicated, etc) are specified by choosing them from a
list and filling in relevant details (such as the maximum number which may be present for dynamic objects). When an object is placed into a room, an indication of its presence appears on the map (what this indication is may be specified by the user, ranging from its name written on the map by the room, to an icon of the object appearing inside the box, to a list of objects beside the map window, to nothing at all for experienced designers).

Once the basic attributes and properties of an object have been specified, the author must then define the methods, or abilities, that the object has. How will it respond to player actions? What other objects does it affect? Creating interesting, complex, and powerful object behaviors is easily done, because of the system of traps, triggers, and hooks that forms the basis for the abstract adventure game model described earlier in this chapter. The author is able to concisely define the conditions under which an object should respond to player actions and then define the effects of those actions. Default actions which are provided for all objects may be left intact, replaced with user-defined actions, or removed entirely. The gold nugget in our example caused a change in the state of two rooms and a message when it was picked up, and is an example of a trigger. Because the method of specifying object behaviors is done through the abstraction, it is independent of any programming language. These behaviors are stored along with the object and are then mapped to TADS and Inform using the template library.

In addition to immediate reactions to player input, some objects need to be dynamic, or “alive” — they need to respond to the passage of time. Daemons may be started, stopped, or modified by objects to cause them to perform various actions or print out special messages, or fuses and timers may be set in response to player input. For example, suppose that we wanted to create an object that represented a bomb. If the player presses the big red button on the bomb, a fuse is started which in five moves causes the bomb to explode, destroying everything in the room (including the player, if she happens to be present). A message is also dispatched to all of the rooms surrounding the explosion room, indicating that a loud explosion was just heard nearby, letting the player know that it is now safe to return and see what havoc the bomb caused. Creating a bomb, or another complex object with an
attached timer, would be easily accomplished using a trigger and a fuse in TA DA!.

By allowing the author to create objects in stages (first working out the basic details, then specifying attributes, then creating behaviors) TA DA! permits the author to work on the game a little bit at a time without having to jump back and forth between sections of program code which might be spread across multiple files (TA DA! automatically segments the code that it generates into separate files for rooms, objects, and characters unless told to do otherwise by the user). The author can create all of the objects in the game, give them names and descriptions, then go back and work on each one to produce more complicated interactions. Objects can affect one another, the player or other characters, the rooms, and so forth. Everything is specified using the abstraction so that there is no language-dependent design being done.

In summary, then, although TA DA! cannot help an author decide what objects should be present or how to make them interesting and exciting, it provides an easy interface and powerful creation capabilities which should enable all but the most advanced programmers to recreate anything which they could have coded by hand using the design tool without ever having to touch the underlying code. For those authors who do wish to do so, of course, there is an edit window for adding user code or modifying TA DA!'s generated code, but it is one of the major goals of the project to make using that window unnecessary.

5.6.4 The Characters

Few text adventures are successful without memorable NPCs (Non-Player Characters). Their design is an art unto itself, which is why TA DA! has a special set of tools dedicated to creating interesting and useful characters.

Characters are a special kind of game object: they walk, they talk, they jump through flaming hoops. Characters can be as simple as a librarian who hushes the player every time he tries to speak, or as complex as a robot who follows the player around, offering helpful advice and trying to assist the player with everything she tries to do (and generally making a mess of it). Characters can have conversations with each other and with the player, and
they can choose to respond to or ignore orders given to them by the player or by other characters. They can pick up and manipulate objects just as the player can, and sometimes will have vital objects which the player must somehow beg, borrow, steal, or trade from them in order to solve various puzzles in the games.

In general, the process for creating a character is similar to that for creating an object or a room; first a “blank” character is generated using a button, which prompts the author to fill in name, key words, description, and the like. Just like objects, characters can respond to player actions, so there is the same set of tools for creating traps, triggers, hooks, timers, and fuses as for objects. For very simple characters, such as the librarian mentioned above, a simple set of messages and reactions will be sufficient. However, for more complicated characters who have lives of their own outside of what the player does or with whom the player may converse, there are two special tools intended to assist authors with the process of designing truly unique and interesting NPCs.

Specifying the “life” of a character may be done in a variety of ways. The character can follow the player around, stay in one place, or travel around independent of what the player decides to do (either randomly or following a pattern specified by the author). Characters may react to objects that they see or that the player shows or gives them. Characters may also have a set of actions that they perform independent of the player. Because of the way both TADS and Inform’s object models are constructed, the messages which are passed to objects when they are manipulated do not actually have any special information about who is doing the manipulation. Both systems are sufficiently generic that it is quite possible for objects to be manipulated by NPCs in exactly the same fashion as by the player. Therefore, TA DA! allows the author to build “command lists” that an NPC can execute.

NPC commands can be triggered by any of a variety of things: the passage of time, player actions, the presence of other characters or certain objects, or even just randomly generated numbers. A clever author can create characters who appear to be traveling around the game world pursuing their own goals, which may or may not be complementary to the goals which are being pursued by the player. Imagine the player’s chagrin when she
takes too long to get into the Baron’s treasure room and discovers that an NPC has been there first and cleaned out all of the gold! Now the plot of the game changes entirely as the player must track down and find this character to steal back the gold. Or, assuming she gets to the treasure room before the NPC thief does, once the player steals the gold a character representing the town sheriff can begin to track the player. Depending on how good she has been at hiding her trail, the sheriff might catch her and throw her in the dungeon, or she might escape to enjoy her ill-gotten gains.

In addition to the actions and responses that a character might have, many NPCs are also placed into a game with the ability to have conversations with the player. These NPCs might have valuable clues that the player needs to know, such as a secret password, or they might just be there to provide atmosphere, giving the player more information about the game world, or they might even be there to provide comic relief, such as a court jester who tells jokes and funny stories when asked by the player. For the creation of NPC conversations, there are two systems available. One is that of directed dialogue, and the other is that of dialogue trees.

Directed dialogue is the “traditional” method of player conversations with NPCs. The player must initiate the conversation, by asking or telling the NPC about a particular topic. For example, suppose that the player needs a ride to an island, so he approaches the dock. When talking to the local fisherman, the player might type: **ASK FISHERMAN ABOUT BOAT.** If the boat is something in the fisherman’s set of topics, then he will launch into a discourse about what a faithful boat it’s been and how he sure wishes someone would help him plug that leak in the bottom, because then he’d give that person a ride anywhere they wanted to go. The player might get a clue from this response and next type **ASK FISHERMAN ABOUT LEAK,** which would cause the fisherman to show the hole in the bottom of the boat to the player, who just happens to be carrying a cork, and a partnership is born. This form of conversation is really nothing more than a specialized form of character reaction to player actions (the verb in this case being **ask** or **tell**) but it is important enough that **TA DA!** provides a special mechanism for creating these kinds of conversations so that the author
can concentrate on making the character's responses interesting and consistent.

Dialogue trees require the author to put a bit more thought into the characterizations of both the NPCs and the player's character itself. In many games, there is no real personification applied to the player's character; the absence of giving the player a specific character is intended to allow the player to picture herself as actually being inside the story. Some games, however, do choose to cast the player as a specific — usually important — person in the game world (a princess, or a great knight, or the evil vizier), and such a game must provide some characterization for the player's character so that the player will understand and relate to her role. The difference between the former method and the latter method is the difference between second person ("You do this") and first or third person ("I do this" or "Sean does that").

Dialogue trees enable the author to provide a detailed level of characterization for both NPCs and the player's character, where necessary. The game author builds a dialogue tree by providing a series of conversational "gambits," or choices, one of which must be chosen by the player. To continue the example of the fisherman, instead of asking the fisherman about a particular thing (such as his boat), the player would merely initiate a conversation by typing TALK TO FISHERMAN. The dialogue tree would then give the player a set of choices as to how to open the conversation, for example:

1. "Catch many fish today?"
2. "That's a mighty fine boat you have there!"
3. "You sure are ugly!"

The player would then choose one of the three responses, each of which would elicit a different response from the fisherman, followed by more choices for the player.

Of course, some decisions by the player cause the conversation to take decidedly different turns (for example, choosing number 3 when talking to the fisherman might cause him to become angry and end the conversation, or even attack the player if he was in a bad enough mood. In either case, the player is unlikely to learn about the leak and thus will be unable to help the fisherman and get a ride in the boat). Because each choice by the player leads
to a different response by the NPC, with different succeeding choices afterwards, the overall possible paths for the conversation form a tree. TA DA! provides explicit support for this method of characterization, allowing the author to build and edit dialogue trees from within the system and then generating all of the code necessary to support them. (Although TADS and Inform do not themselves have built-in support for dialogue trees, extensions to both languages have been written which permit their use. Because dialogue trees are so common in modern adventure games, I felt that including the ability to create dialogue trees in the product would enhance it substantially).

In summary, creating interactive and unique NPCs is an important part of most adventure games. Some games have no characters at all, while others have many, perhaps dozens. While TA DA! cannot help the author with the process of coming up with good dialogue or interesting characters, making them "come to life" is a process which the system makes very easy, allowing the author to concentrate on the characters themselves. The time that the author doesn't have to spend digging around in the code and porting between languages may be spent instead writing more and larger snippets of dialogue to expand the vocabularies of the NPCs and make them all the more realistic and believable.

5.6.5 The End?

Finally, having examined the system and the languages that it is based upon, we must ask the question of whether or not TA DA! is successful. Of course, the program is still evolving, and has the potential to continue to do so because of its template-based design. Ultimately, the "success" or "failure" of the system will depend on whether or not it actually enables anyone who would not otherwise have been able to do so to create adventure games. On that front, TA DA! has already succeeded, because there are several games already in development using the system. These games are being created by people who have never been exposed to computer programming using a structured language such as TADS or Inform.

Another measure for the program might ask how games created using the system com-
pare to games created without using the system, by programming directly in the target languages. One point that should be mentioned is that, to date, no game has been released which is available in both TADS and Inform format. Authors are currently forced to choose one system over the other; game players must maintain both if they want to be able to play all of the new releases. TA DA! and other projects like it can help to reduce this need in the future. There is no doubt that a skilled programmer using either TADS or Inform can create games of greater complexity than any programmer could create using TA DA! alone. The system is simply not flexible enough to cover all possible situations. One necessary peril of abstraction is that you give up some of the power which lies in the details. However, even for a skilled programmer, TA DA! can still provide benefits in the form of reduced development time for the foundation of the game and a clean integrated development environment. Since the system allows users to directly edit the generated code, the "power user" is free to take the code generated by TA DA! and modify it to her heart's content.

Finally, the question arises of whether or not TA DA! truly abstracts all of the features of both programming languages. The answer is that it does not. For example, Inform features a number of powerful commands allowing the programmer to section the screen into multiple windows, which may be used to display different aspects of the game at the same time (the player's inventory, a room description, and perhaps a map of the area constructed using ASCII characters). TADS has nothing that even approximates this feature. Therefore, in order for a game created using TA DA! to be dual-compiled on both systems, I was forced to completely ignore this feature of Inform. A programmer who wanted to make use of these commands would not be able to use TA DA! at all because its generated code is not designed to be compatible with them. Is this a failure of the system? My contention is that it is not, because the goal was not to abstract two specific programming languages and provide a mapping for all of their features. The goal was to abstract a particular application — in this case, the computer text adventure game — and to see if this abstraction was strong enough that two disjoint languages could both be used to satisfy its assumptions and conditions. In this regard, one can clearly see that TA DA! succeeds admirably.
In the end, the issue of whether or not TA DA! accurately reproduces the design process is moot. The system provides a whole new way to create games — sitting at the computer specifying objectives and behaviors instead of tinkering with programming constructs. The system is not finished evolving, but it is my hope that it will be possible to someday completely remove the programming altogether and allow an author to simply create games straight from her imagination and onto the computer screen. While this goal is still far in the distance, TA DA! is at least a step in the right direction. Its initial release to the text adventure game design community is certainly not “the end” for TA DA!; it is merely the beginning.
Chapter 6

Conclusions and Future Research

TA DA! is new, and so it has not yet found its place in the adventure game community. Whether or not it will gain acceptance only time will tell.

TA DA! is currently available for downloading from the author’s World Wide Web page, located at

http://www.cs.wku.edu/~mollems/TADA

This archive contains the Windows 95/Windows NT version of the program, as well as a tutorial on adventure game writing and a sample game created using TA DA!. The Web site will be updated with the latest version of the system as changes are made, as well as providing areas for user feedback and suggested changes and improvements to the system. TA DA! is available free of charge, although the copyright on the program, tutorial, and sample game is retained by M. Sean Molley.

Although TADS and Inform are currently the two “big boys” of the adventure game authoring community, there is no reason to assume that they will always be dominant. As new languages are developed, or as new versions of existing languages make the older mappings obsolete, it will be necessary to update TA DA! to reflect these changes. Hopefully, the system is sufficiently easy to use in the sense that anyone can write games with it; however, the template system and underlying code is complicated and most likely beyond
the capabilities of novice or non-programmers to change on their own. So, to this extent, there is still work to be done in the area of making the system completely usable by non-programmers.

Additionally, I would like to allow expert users greater ability to edit the generated code and add their own code seamlessly. This addition would also require adding the capability to link from TA DA! to a particular language's integrated debugger, since the code written by users cannot be guaranteed to be bug-free like the code generated by the system. TADS currently has a debugger, while Inform does not. The creation of a language-independent adventure game debugger would be a substantial undertaking and is certainly beyond the scope of this project, but it might make an interesting research project in the future.

Perhaps the most significant aspect of this project is not its attempt to simplify the game design process for non-programmers, but rather the work done on simplifying a particular application domain (in this case adventure games) to a single abstraction which is completely language-independent and able to be implemented in a visual manner. The visual design paradigm is certainly popular at the moment, with tools such as Delphi and Visual Basic dominating the market, and it is certainly easier for new programmers and people who have never programmed before to grasp a visual metaphor rather than a textual one. It is perhaps ironic that TA DA! implements a visual method of creating textual games, but the abstraction which underlies the system could be applied to some kinds of graphical adventure games as well with only a few modifications.

If a similar abstraction could be found for other application spaces, then tools like TA DA! could be developed for those kinds of applications that would enable non-programmers to build them simply by specifying behaviors and making design decisions. If enough of these abstractions and high-level specifications could be created, it is not too much of a stretch to postulate a "super-abstraction" that would sit on top of all of the meta-grammars and provide an overarching specification for any program within any of the domains. Such an abstraction would enable the development of a visual tool that would allow a non-programmer to write nearly any application that could be imagined simply by specifying
the design and behaviors of the program. This would be the design equivalent of a Turing Machine — any program which could be envisioned by the user could be created using the tool. Clearly, TA DA! is only the tiniest of steps in this direction, but it does point out some of the difficulties that will arise.

Indeed, TA DA!'s template-based system will most likely prove inadequate for the world of programs at large, because of the nearly infinite variability that is found therein. Nevertheless, the model presented here would be appropriate to many application domains. The so-called “experts” and “wizards” that are present in most modern integrated development environments are nothing more than glorified templates that are filled in according to user specifications and answers to simple questions about the desired behavior of the application. Database application builders have long used this type of approach in providing users with the ability to construct complex systems merely by specifying the types of data to be stored and the rules governing data movement and transformations. The template model could be used to design a system for creating expert systems, or fuzzy logic systems, where the templates represented rules instead of language constructs. The model could be applied to build a general-purpose physical simulation system, where the user would specify the particular physics and properties of the system to be simulated and the templates would represent those fundamental properties of interactions between objects in the world. The number of areas that are sufficiently narrow in scope yet deep in their number of applications is certainly large enough to warrant further investigations into these types of visual development tools, especially as computing moves out of the hands of the computer professionals and into the hands of the world at large.

A major area of research that could be explored would be extending TA DA! to add support to the system for more types of game designs, especially “modern” adventure games with graphics and sound. Evolving the tool to support these kinds of games would not only introduce more templates and layers to the meta-grammar to deal with their concepts but would also make TA DA! potentially viable as a commercial product. Furthermore, continued development of the user interface will allow for tighter integration between the
design tool and the underlying programming languages, resulting in more efficient code generation. Finally, the development of a new programming language for the creation of these modern adventure games, based on the abstraction instead of the other way around, could eliminate some of the problems and compromises that arise as a result of mapping between both TADS and Inform and provide the tightest coupling possible between the visual design tool and the compiler underlying it. Of course, building a new and specialized programming language to work directly with the design tool would get away from one of the original goals of TA DA!: to provide support to programmers no matter what their language of choice might be. However, such a language would allow for the creation of advanced games that are not supported by any language at present.

In conclusion, TA DA! in and of itself is not going to have a major impact on the computer gaming world. However, the ideas behind it are powerful and are at the cutting edge of compiler theory. Indeed, TA DA! straddles the line between being a compiler and being some kind of computational jigsaw puzzle. With pieces that are sufficiently abstract, yet suitably powerful, TA DA! has the potential to be a jigsaw puzzle from which one can assemble any adventure game. Bringing the creation of adventure games to the mass of non-programmers is a step in the direction of bringing the creation of all programs to that level. It heralds a future where computers will be told what to do and not how to do it, where users will specify outcomes instead of behaviors, and where the best programmers will not be those with the greatest ability to manipulate symbols on a keyboard, but rather those with the vision to define their ideas in terms anyone can understand and appreciate. Only then can we truly unleash the power of the human imagination to create new worlds limited only by our dreams.
Bibliography


