Honours Year Project Report

Foundations for a Browser-based Application Platform

By

Daryl Seah Boon Leng

Department of Computer Science
School of Computing
National University of Singapore
2007/2008
Honours Year Project Report

Foundations for a Browser-based Application Platform

By

Daryl Seah Boon Leng

Department of Computer Science
School of Computing
National University of Singapore
2007/2008

Project No: H041200
Advisor: Assoc Prof Martin Henz

Deliverables:
Report: 1 Volume
Class System Quick Guide & Reference: 1 Volume
Source Code with Inline Documentation
(in SubVersion Repository)
Abstract

Being ubiquitous and rich in developer technologies such as JavaScript, AJAX, CSS and DHTML, modern web-browsers are rapidly gaining popularity as an ideal application platform. As such, many feature-rich libraries are available that attempt to simplify web-application development. Unfortunately, there is insufficient support for client-side desktop-like applications, where the need for persistency (without a server) and security (without a sandbox) is important. Moreover, the lack of modularisation, verification, and other high-level language features in JavaScript makes it extremely difficult to create a complex yet maintainable application, let alone an entire supporting platform.

In this project, we design and implement the basic architecture for TiddlyCard – a collaborative client-side browser-based application platform. By taking an operating systems perspective and building only upon the core JavaScript language, we attempt to lay the groundwork for a cross-browser platform capable of managing multiple concurrent processes, supporting them with basic I/O, files and persistent storage, and easing their development by introducing high-level language features in the form of an extended JavaScript class system. We also examine the many challenges associated with having the browser as an application platform; including aspects of security, efficiency, inter-operability and coding discipline.

Subject Descriptors:
   D.2.11 Software Architectures
   D.3.3 Language Constructs and Features
   D.4.3 File Systems Management
   H.3 Information Storage and Retrieval
   D.4.6 Security and Protection

Keywords:
   JavaScript, platform architecture, language extensions, file systems, persistent storage, web operating system, browser security

Implementation Software and Hardware:
   JavaScript 1.5, Mozilla FireFox, Microsoft Internet Explorer
Acknowledgements

First and foremost, I would like to express my heart-felt gratitude and thanks to my project supervisor, Dr Martin Henz, for his strong support and guidance throughout the entire project.

My appreciation also goes out to Mr Michael Clark, a long-standing external contributor to the TiddlyCard project, for his valuable advice, design ideas and for providing much needed resources to everyone.

This project would not have been possible without the assistance and support from other developers in the TiddlyCard family of projects; including Mr Lek Hsiang Hui, Mr Michael Lin, Mr Siva Subramanian, Mr Tang Tung Leh, Mr Tran Khoa Nguyen, Ms Wang Shangshang, and Mr Zhou Lin (Richard).

I would especially like to thank Mr Lek Hsiang Hui, Mr Tang Tung Leh and Mr Zhou Lin (Richard) for integrating their projects with mine, helping to locate bugs and their many suggestions for improvements.

For those whose names do not appear here, but who have helped me in one way or another in this project, my gratitude goes out to you as well.
Table of Contents

Title ......................................................................................... i
Abstract ..................................................................................... ii
Acknowledgements .................................................................... iii
Table of Contents ....................................................................... iv

1. Introduction .............................................................................. 1
   1.1. The Rise of the Browser .................................................... 1
   1.2. TiddlyCard – An Application Platform ............................ 2

2. The Big Picture ......................................................................... 3
   2.1. Nature of Browsers as Application Platforms ................ 3
       2.1.1. The JavaScript Language......................................... 3
       2.1.2. Limited Resources ............................................... 3
       2.1.3. Single-Threaded ................................................... 4
       2.1.4. Heterogeneity ...................................................... 5
       2.1.5. Client-Side Security .............................................. 6
   2.2. The Operating System Model ........................................... 7
       2.2.1. Analogy in Practice.................................................. 7
       2.2.2. Going Even Further .............................................. 8
   2.3. An Ambitious Plan ......................................................... 9
   2.4. Development Strategy .................................................... 10

3. A Touch of Class .................................................................... 11
   3.1. JavaScript in the Wild .................................................... 11
       3.1.1. Insufficient Runtime-Safety ...................................... 11
       3.1.2. Inadequate Support for Classes ............................... 12
       3.1.3. No Support for Modularity .................................... 13
       3.1.4. Taming the Beast .................................................. 14
   3.2. A Custom Solution ......................................................... 14
       3.2.1. Design Principles .................................................. 14
       3.2.2. Specific Requirements .......................................... 15
       3.2.3. Motivation for a Custom Solution ........................... 15
   3.3. The Method-Chaining Model ......................................... 16
       3.3.1. Analysis and Proposal ........................................... 17
       3.3.2. The Pros ........................................................... 19
4. The Data Layers: I/O, Files and Persistence

4.1. Asynchronous Design

4.1.1. A Tough Pill to Swallow

4.1.2. Integration into the Platform

4.1.3. Simplifying the Task

4.2. Basic I/O Abstractions

4.2.1. Strings Instead of Byte Arrays

4.2.2. Coarse-Grained Non-Blocking Reads

4.3. A Unified File-System

4.3.1. File Addressing Scheme

4.3.2. Provider Framework

4.3.3. Tree Data-Structure

4.3.4. How It All Works

4.3.5. Implemented Providers

4.4. High-Level Persistent Storage

4.4.1. Design Process

4.4.2. Implementation Overview

4.4.3. Addressing Scheme

4.4.4. Marshalling and Object Serialization
4.4.5. Store and Provider Framework .................................................. 41
4.4.6. Query Handling ...................................................................... 42
4.4.7. Implementation of a File-Store-Provider .................................. 43

5. Completing the Picture .................................................................. 44
  5.1. Applications and Processes ...................................................... 44
  5.2. Inter-Process Communication ............................................... 46
  5.3. Console and Shell Interpreter ............................................... 46
  5.4. Deploying Software Units ...................................................... 49
  5.5. Booting and Shutting-Down ................................................... 50
  5.6. Resolving Security Issues ...................................................... 50
    5.6.1. Context Tracking ............................................................ 51
    5.6.2. Scope Chain Insertion .................................................... 51
    5.6.3. Code Parsing and Transformation ..................................... 53

6. Conclusions .................................................................................. 54
  6.1. Summary of Achievements .................................................... 54
  6.2. Recommendations for Future Work ........................................ 55

References ....................................................................................... vii

Appendix A – TiddlyCard Class System: Quick Guide & Reference ........ ix
1. Introduction

1.1. The Rise of the Browser

In recent years, the humble web-browser has seen a tremendous surge in popularity as a platform where applications can be developed, deployed and used. Such applications are easily distributed (with a modest hyperlink) and are accessible to any user in the world with a modern computer and an Internet connection.

The development of these web-applications was largely driven by the advent of AJAX (Asynchronous JavaScript and XML) and Dynamic HTML. Both of these developer technologies were vital in creating more responsive and dynamic web pages, without the apparent delays associated with retrieving static pages from a server. However, even with AJAX, applications were still mostly dependent on a server for processing and/or storage, making the server the decisive limiting factor in the performance and scalability of an entire system.

As such, developers have been trying to delegate more and more processing to the client in an effort to reduce server load; essentially taking advantage of distributed computing. This demand for client-side processing has motivated browser vendors to provide greater functionality for developers, fuelling the trend even more. Consequently, browsers are now formidable platforms for applications to run in, and they are getting more powerful with each new release.

Recently, another novel use of the browser has emerged in full-blown client-side applications – where access to a server is either optional or not required at all. These applications are given greater access to the user’s system, being free of the sandbox security model, and may even be able to perform local file I/O depending on the browser used. One such application is TiddlyWiki \[^{[17]}\] , a single-file wiki-style personal notebook, written by Jeremy Ruston.
1.2. **TiddlyCard – An Application Platform**

Inspired by the success of TiddlyWiki and borrowing ideas from HyperCard [23], *TiddlyCard* is a collaborative project aimed at developing a browser-based platform for the creation, distribution and running of client-side applications. By leveraging on JavaScript, AJAX, DHTML, CSS and other key browser technologies, its objective would be to fully exploit the rising popularity of client-side browser applications and take the concept to a whole new level by giving developers of these applications the support of a world-class platform.

The first incarnation of TiddlyCard was simply an adaptation of TiddlyWiki itself, and was started by opening a local HTML file statically referencing all the external resources required (i.e. JavaScript files, style-sheets, images, etc…) to load the system. Even though this prototype was visually appealing and functional (from a user’s perspective), it only had a rudimentary architecture and applications were either dependent on the TiddlyWiki backend or had to run without any significant platform support at all, making application development a complicated and laborious process.

It was obvious that the TiddlyCard prototype needed quite a bit more work to transform it into the ‘world-class’ platform envisioned. An entire overhaul of the architecture was required to achieve the many features desired, including extensibility, security, developer-friendliness, and complete cross-browser compatibility. The focus of this honours year project is in establishing the foundations for such an architecture with thorough design and solid implementation.
2. The Big Picture

2.1. Nature of Browsers as Application Platforms

The browser environment poses several challenges and limitations that make it both a unique and hostile place to establish an application platform. Features normally taken for granted in non-browser systems become difficult or even impossible to realize. In this section, we shall examine some characteristic traits common to all browsers and show that a practical platform implementation requires both innovation and adaptation in design in order to survive.

2.1.1. The JavaScript Language

The language of choice in all browsers is JavaScript. Often seen as ‘merely’ a scripting language, JavaScript is in fact a highly dynamic and powerful multi-paradigm programming language with prototype-based inheritance. While this is a joy for programming aficionados and fine for simple programs, developers used to mainstream object-oriented languages like Java, C++ and C# often find it confusing and insufficiently expressive. Its dynamic nature also provides amazing flexibility, but at the expense of runtime control and safety.

Even for JavaScript veterans, developing, debugging and maintaining a complex application with the language alone can be a nightmare. It was apparent that higher-level language features were required for the platform; not just for the developers ultimately using it, but also for the project team creating it. In section 3, we will showcase the language extensions developed for this purpose.

2.1.2. Limited Resources

The browser is really just a small habitat in the vast eco-system that is the user’s computer. As a desktop application, it may be allocated only a subset of the processing and memory resources available. In addition, JavaScript is an interpreted language, meaning executing scripts only get a fraction of the processing power afforded to the browser.
This may seem like a serious drawback of the browser as an application platform. Why would an application want to be caged in the browser when there are so many perks associated with being a first-class desktop application? The reason is portability and simplicity. If properly written, a scripted application would be able to run anywhere a browser exists. Also, if the program is simple, it might be faster and require less effort to simply write a script than a full-blown desktop application.

Even for complicated applications and programs, the limited resources of the browser have not deterred developers. In fact, many scripted libraries are available that demonstrate the true potential of the browser; including implementations of the ciphers AES \(^{[21]}\) and RSA \(^{[24]}\). Even though speed is still an issue, the increasing popularity of the browser platform would help drive the development of ever faster script interpreters. When complete, TiddlyCard might just be the first system to take full advantage of this trend.

2.1.3. Single-Threaded

The browser is an event-driven environment and code attached to an event is usually executed by the user-interface thread. This implies that the environment is effectively single-threaded (at least at the level of code execution) and no other code can run while a script is running. Furthermore, a long-running script may even lock up the browser completely, enticing the user to forcibly terminate the script or occasionally even the browser itself.

These problems have sparked attempts to either emulate or add true multi-threading to JavaScript. For example, some developers have created interpreted languages on top of JavaScript \(^{[14]}\), which are run concurrently using a custom ‘thread’ scheduler. Others use Java’s built-in JavaScript interpreter, Rhino, to add functions linking directly to the Java threading API \(^{[18]}\); which allows JavaScript code (executed using Rhino) to spawn threads and perform various synchronization tasks. The latter solution only works on a server though, since the browser client might not have Java installed or access to its Rhino interpreter. The former is just an emulation – extra baggage that we could do without.
The conclusion was that we had to make do with a single thread. However, a platform running multiple applications on just one thread would most certainly have some disadvantages. For instance, an application blocking for I/O would effectively disable all other applications, and the browser along with it. In essence, developers would have to cope with a system that needed applications to use non-blocking I/O wherever possible and voluntarily allow other applications to run (i.e. cooperative multitasking). Since developers might not have the discipline to write such code, we attempt to gently persuade them to comply by requiring applications to yield upon some strategic API calls. The crux of this approach is discussed in section 4.1.

2.1.4. Heterogeneity

Ironically, a common trait among all browsers is that they have many traits not in common. Each browser may support different technologies, implement different standards, implement the same ‘standards’ differently, and run on various underlying operating systems. Every browser has its own unique features, quirks and bugs. To make things even more complicated, deployments of the same browser may have varying user preferences and pluggable components (like Adobe Flash) may or may not be installed.

The heterogeneous nature of browsers is probably the biggest annoyance in web-development. Developers seeking as large an audience as possible have to spend considerable effort in testing and resolving incompatibilities. Even when focusing on just the two most popular browsers (IE and FireFox), it is still quite a feat to get everything working just right.

A similar situation applies to client-side browser applications, only it is much worse. Compared to web-applications, which can usually rely on (more or less) established standards, client-side applications need to do unconventional things; like reading and writing local files. The API for these specialised functions is very much browser-specific, if available at all. For example, local file I/O in IE is supported through an ActiveX
control \cite{9}, whereas in FireFox it can be done with their XPCOM (Cross Platform Component Object Module) framework \cite{12}.

If each and every application had to deal with all of these browser differences, then the concept of a ‘platform’ would be pointless. Developers would want to write their code once and have it run on any browser as long as TiddlyCard is there to support it. It was clear that a system of abstractions were needed to shield application developers from the complexities of the browser environment.

2.1.5. Client-Side Security

...or the lack thereof, is a serious concern for the browser environment. Normal server-based web-applications have the browser security sandbox keeping them in check, but not client-side applications. These run straight-off the local file-system and typical user configurations can give these programs almost unrestricted system access. Such a scenario may be acceptable for a single-application environment, where suitable browser configuration and a user discerning about security prompts could make an informed choice whether or not to give the application access to these privileged resources.

However, running an application platform is a different story. A user granting the platform access to the local file-system and network also gives these access rights to all the applications running on the platform. Any one of these applications could be malicious and pose significant risk to the user. The problem stems from the fact that within the same page and session, the browser cannot distinguish between a platform and its applications. This (very) coarse-grained access control at the browser-level means the platform itself has to handle security.

At this point, the very nature of our implementation language tosses us an unprecedented challenge: How can the platform restrict access to an application written in JavaScript – a language so dynamic that it is inherently uncontrollable? In section 5.6.3, we will propose a solution to this problem that is beyond the scope of this project, but will be a critical component in ensuring the completeness of the platform.
2.2. The Operating System Model

From the characteristics we have identified and their evaluation, it is evident that a browser platform can be seen as analogous to an operating system. By structuring and designing our platform as such, we would be able to refer to established models for ideas and also have a basis for discussion and comparison with other systems. The diagram below shows how the platform can be layered to match the structure typical of an operating system.

![Diagram of TiddlyCard and the browser layered as an OS.](image)

2.2.1. Analogy in Practice

The main job of an operating system is to manage resources and to provide means for applications access them. Hardware abstractions are made as well and implemented by device drivers, saving applications the need of knowing how to interact with the underlying architectures.

In the context of a browser platform, memory management is not a major concern since the environment supports automatic garbage collection. Instead, scheduling of processes and ensuring security is
important, as this determines how well applications run together and whether or not they can pose a risk to other applications, the system, or even the user. As mentioned earlier, the single-threaded nature of browsers implies that centrally-managed pre-emptive process scheduling would not work since executing code must voluntarily yield to allow other code to run. However, the act of scheduling a process is not entirely useless, as it allows us to track which process is currently running and subsequently determine whether the running application has permission to access certain resources – a feature important for the access-control aspect of security (see section 5.6.1).

As for the issue of browser heterogeneity, the notion of hardware abstractions and device drivers comes in handy. Comprehensive and well-defined abstractions of standard platform services, such as I/O, networking, persistent storage, and user-interface support, would give applications a consistent cross-browser means of accessing them. The aim would be to transfer the responsibility of handling platform differences to specialised driver developers, so application developers can concentrate on more important tasks. This design promotes portability of applications and extensibility of the platform.

2.2.2. Going Even Further

So far we have brought together similar concepts from the browser environment and operating systems, showing how the OS model has helped in grounding our design. We can stretch the model even further though, by introducing features which are not normally associated with a browser platform, but are a staple of modern operating systems.

For example, most operating systems come with a terminal emulator coupled with a command-line shell interpreter. Implementing such a system for our application platform would give it a distinctive OS feel and assist in managing, debugging and inspecting our runtime environment. Features such as pipes, input streams, output/error streams, and a navigatable file-system all naturally follow this concept.
An operating system (or any other decent platform) also normally provides rich utility libraries that are non-essential to the operation of the system, but help to reduce the complexity of application code and attract developers. These libraries may include API to perform cryptography, arbitrarily-large integer math, image processing, data compression, and many other specialised functions. Integrating as many such utilities as possible into our platform API would significantly increase the attractiveness of the system as a development platform.

2.3. An Ambitious Plan

From the operating system model, we have derived many ideas for our system and now we can assemble them into a complete platform architecture and API. The component diagram below, incorporated with these design ideas, gives an overview of the plan for the entire system.

Figure 2: Detailed structure and organisation of the TiddlyCard platform components.
Notice that we have borrowed another OS-related concept in the introduction of a *bootstrap loader*, which is a light-weight piece of platform-specific code capable of kick-starting the entire platform (see section 5.5). Some other components, like the deployment system and security layer, are not within the scope of this project for full-implementation, but are included in the diagram as a target for future completion; these are discussed in section 5.4 and 5.6.

### 2.4. Development Strategy

TiddlyCard is a collaborative project and the combined work of many others plays an integral role in realising the complete system. Regular meetings are held to discuss ideas, report bugs, provide progress updates and determine dependencies among the various sub-projects; some of which have yielded the *networking libraries, user-interface toolkit* and vital components of the *security layer*.

The goal of this particular sub-project is to establish the foundations of the platform by designing and implementing all or most of the critical components necessary for current and future developers to build upon. With this in mind, research and development was concentrated in the following areas, all of which were considered vital for the platform:

- *language extensions* (section 3)
- *I/O and files* (section 4.2 – 4.3)
- *high-level persistent storage* (section 4.4)
- *supporting applications and processes* (section 5.1 – 5.2)
- *a console and shell interpreter* (section 5.3)
- *deployment of software units* (section 5.4)
- *booting and shutting-down the system* (section 5.5)
- *platform security* (section 5.6)

It was decided that a bottom-up implementation strategy would be employed; where the lower components in the diagram (figure 2) are implemented first. However, some of these components had to be developed incrementally as well, since not all of them had definitive requirements, and time was needed to observe how they might evolve.
3. A Touch of Class

Language support is critical for the success of the project as it can expedite development, improve code readability, reduce errors due to programmer mistakes, and ensure consistency by forcing all developers follow the same programming paradigm. In this section, we will explain in greater detail the need for augmenting the JavaScript language and introduce a class system which will be up to the task.

3.1. JavaScript in the Wild

JavaScript can be a nasty beast when used on its own. Several weaknesses of the language have plagued programmers since its introduction and have tarnished its reputation as a language fit for the development of complex applications and libraries. We shall take a look at a few of these problems and examine the various attempts at working-around or fixing them.

3.1.1. Insufficient Runtime-Safety

Being a dynamic scripting language, the syntax and semantics of JavaScript are significantly more relaxed compared to other languages. For example, variables have no declared type (not type-safe) and using or assigning variables without declaring them first is permitted. Also, since functions are first-class values in the language, a duplicate definition of a function simply replaces the older definition.

```javascript
// Declare x twice, missing semi-colon
var x, x
// Assign to Undeclared y
y = x;
// Define Function A
function A() { return 'A'; }
// Re-define Function A
A = function() { return 3; };
```

Figure 3: Perfectly legal JavaScript code, but very likely not the programmer’s intention.

These relaxed rules may be favoured by some programmers, as they would not need to worry about ‘trivial’ issues like un-matched types, missing semi-colons, and so on. However, this lack of code integrity
often has the side-effect of introducing runtime bugs which are extremely hard to trace. Finding the source of one or more such bugs would be an arduous task in an application platform with a huge codebase. Runtime issues like these are easily avoided in other languages with stringent static semantics applied at compile-time. Unfortunately, this is not done in JavaScript.

3.1.2. Inadequate Support for Classes

JavaScript programmers are given the power to mix-and-match various programming styles; specifically those from the functional, imperative, object-oriented, and prototype-based programming paradigms. Indeed, this gives developers much flexibility in choosing the best combination to solve a particular problem. However, the popularity of the object-oriented paradigm has driven most developers to utilise classes in JavaScript – the support of which is sadly inadequate.

A ‘class’ is really just an extended function definition, where any function can be instantiated with the `new` keyword and a `prototype` property on the function determines the instance members and parent of the class. This is known as prototype-based inheritance.

```javascript
// Define a new class SpecialDate
function SpecialDate() {}

// Inherit from the class Date
var tmp = function(){};
tmp.prototype = Date.prototype;
SpecialDate.prototype = new tmp();

// Override toString method
SpecialDate.prototype.toString = function() {
  return "I’m special!";
};
```

*Figure 4: Defining a class and inheriting from another one using prototypes.*

Notice how complicated and hap-hazard it is just to define the simple class in the example above. The five statements making up the definition have no cohesion at all and unrelated code could easily be slotted in-between the lines.
A consequence of this technique is that class definitions are dynamic – members may be added or deleted and even the inherited class may be changed at any time. Core JavaScript classes like `Object` and `Function` have mutable prototypes as well, so their behaviour may be altered by any executing code. The inability to control whether a class can be re-defined over time makes them unpredictable and there is no guarantee that an object of a class would work as its programmer intended.

In addition, JavaScript provides no native support for declaring and enforcing `public`, `private`, `protected`, `abstract` or `final` class members. An author of a class would have no assurance that a member meant to be `private` is never externally accessible or that an `abstract` method is implemented by an inheriting class. This lack of runtime integrity is just one example of why the language is inherently insecure and this makes JavaScript by itself undesirable for use in an API which demands enforcement of strict implementation rules.

### 3.1.3. No Support for Modularity

The language does not provide any native support for organising logical units of code into namespaces or packages, meaning most JavaScript libraries are simply huge masses of globally defined functions (or prototype-based classes). Although scripts may be segmented into various individual files, the segmentation is only physical and the situation is the same when all the files are loaded. These ‘globally resident’ libraries are frowned upon because there is always the possibility of two libraries interfering with each other by declaring functions or classes with the same name.

In other words, a platform blindly including more and more such libraries into it’s codebase might eventually breakdown simply because of cross-library interference. Naturally, applications on the platform would also suffer from this problem if they do not attempt to logically segregate their code from the code of other applications. It was clear that support and enforcing of namespaces would be extremely important for our application platform if we wanted it to scale well.
3.1.4.  Taming the Beast

Many developers have tried to address these key problems. In fact, the most popular solution is to simply add language extensions to manage defining of classes, verifying their integrity, and organising them into a hierarchy of named objects (for modularity). Unfortunately, no one standard exists – since almost every developer somehow feels the need to design their own class system. Currently in the works though is JavaScript 2.0 \(^2\) (as opposed to 1.5 used in this project), the next version of the language, which will solve all of these problems with proper classes, strict typing, namespaces and many other new features.

3.2.  A Custom Solution

Like all other developers, we felt the need to design our own class system. It had to follow a set of design principles and have certain requirements in order to work well for our client-side application platform. These will be discussed in detail and we will explain the motivation behind a custom solution to achieve these goals.

3.2.1.  Design Principles

*Familiarity:* Developers comfortable in the languages normally used for writing applications should feel at home using our class system.

*Cohesiveness:* A class definition should look like one logical unit.

*Maintainability:* Class definitions should be as readable as possible and be practically self-describing for the sake of maintenance. It should also be easy to edit the code, and bugs should be easily traced and found.

*Convenience:* Difficult tasks should be simplified as much as possible so that code can be short and sweet.

*Unobtrusiveness:* The class system should not, by virtue of its design, interfere with the operation of other independent class systems or JavaScript libraries.

*Compatibility:* The class system should work on any platform (browser or otherwise) that has a compliant JavaScript 1.5 interpreter.
3.2.2. Specific Requirements

The most important requirement was that the class system had to solve or alleviate as many of the weaknesses of JavaScript (outlined in section 3.1) as possible. This includes the ability to verify, modularise, simplify and enhance the use of classes in the language.

More specific requirements were identified based on the design principles, analysis of the overall platform and feedback from discussions with the potential end-users of the class system (other project team members). These requirements were:

- Backwards compatibility with existing code.
- The use of only core JavaScript language features.
- Isolation of class system code in a namespace and avoiding the use of globally defined variables; to avoid conflict.
- Completely avoiding any modification of core JavaScript classes; once again, to avoid conflict.
- Adoption of a naming convention for classes, namespace addresses, and private, public, or protected members; to enforce predictability and allow assumptions to be made at runtime.

Some good-to-have features were also identified, which included:

- Interface and Enumeration constructs in addition to classes.
- Annotations for constructs and their members and a way to query the annotated information at runtime.
- Properties with type-checking and automatic setter/getter method generation.

3.2.3. Motivation for a Custom Solution

Most of the existing class systems were designed specifically for use in web-applications, often touting themselves as a ‘total solution’ to the inadequacies of JavaScript and come with integrated user-interface libraries and other extraneous code. They usually assume the browser environment is exclusively theirs to re-model and change, causing conflicts when other libraries are introduced in the same environment.
We wanted to avoid the reliance on these libraries, which may have complicated licensing agreements, unwanted platform dependencies, and contain code we could not control or did not need. Instead, we chose to learn from them by analysing their strengths and weaknesses in order to create a better solution (seen in the next section).

An alternative to using language extensions was to simply wait for JavaScript 2.0, which would have nearly all the features we need. Unfortunately, progress towards a finalised proposal for version 2.0 has been slow and the specification is only expected to be complete near the end of 2008. Implementation of the standard into the various browsers and waiting for users to migrate would take even longer. This is obviously unacceptable for a project that needs to be completed now!

We also did not want to be dependent on the features of JavaScript 2.0 when we can be assured that using 1.5 would allow our code to run on any browser now and in the future. Nevertheless, it would be prudent of us to consider a class system design that would be able to hide the actual implementation details behind abstractions, allowing us to upgrade the backend to version 2.0 when it becomes available.

3.3. The Method-Chaining Model

The fundamental problem faced when developing the class system was deciding on the technique used to define classes and other constructs. It had to follow the principles outlined in section 3.2.1 and the technique of choice would determine the extensibility of the system and whether it would be easy to realise all the requirements in section 3.2.2.

The breakthrough came when the idea of using method-chaining to define constructs was considered. To explain the concept of method-chaining and its advantages, it would be helpful if we first analysed how two simple classes might be defined traditionally and in the existing class systems of JSClass \cite{4} and Qooxdoo \cite{16}.
3.3.1. Analysis and Proposal

Consider the following code defining two classes, Animal and Cat, written in the traditional JavaScript manner:

```javascript
var Animal = function(name, isCute) {
    this._name = name;
    this._isCute = isCute;
};

Animal.prototype.getName = function() {
    return this._name;
};

Animal.prototype.isCute = function() {
    return this._isCute;
};

var Cat = function() {
    Animal.call(this, "Cat", true);
};

var tmp = function() {};
tmp.prototype = Animal.prototype;
Cat.prototype = new tmp();
```

Figure 5: The class Animal and the class Cat extending Animal in the traditional JavaScript style.

As mentioned in section 3.1.2, this traditional style is not very cohesive. Notice the numerous times we have to repeat `Animal.prototype` for every member defined – refactoring a longer definition could be extremely tedious. We also have to manually make an inheritance link between Animal and Cat. The class system `JSClass` improves on this model by adding some syntactic sugar [4]:

```javascript
Animal = function(name, isCute) {
    this._name = name;
    this._isCute = isCute;
};

Animal.Public = {
    "getName" : function() { return this._name; },
    "isCute" : function() { return this._isCute; }
};

Cat = function() {
    Super("Cat", true);
};

Cat.Exends(Animal);
```

Figure 6: The same two classes written in `JSClass`, with the additional syntactic sugar in red.
Although shorter, the class names “Animal” and “Cat” are still repeated and the cohesiveness of the definition can be further improved. Moreover, JSClass modifies the core JavaScript classes to implement some of the syntactic sugar – something we want to avoid. The JSON (JavaScript Object Notation) approach solves most of these problems. For example, here are the same definitions for the Animal and Cat classes in Qooxdoo\[^{16}\], a UI library with a JSON-based class system:

```javascript
qx.Class.define("Animal", {
    construct : function(name, isCute) {
        this._name = name;
        this._isCute = isCute;
    },
    members : {
        getName : function() { return this._name; },
        isCute : function() { return this._isCute; }
    }
});
qx.Class.define("Cat", {
    extend : Animal,
    construct : function() {
        this.base(arguments, "Cat", true);
    }
});
```

**Figure 7:** The same two classes written in Qooxdoo, where JSON is used to define the class.

Definitions in JSON are concise, cohesive, and easy to batch process, but can be hard to edit by developers due to its ‘rigid’ structure (e.g. preserving the comma between key-value pairs). Also, because JSON is usually hierarchical in nature, you may get confused as to exactly which kind of member you are editing if the nesting is deep. Of course, it is this hierarchical nature which makes JSON so concise.

We can now address the problems of both models (and be innovative) by coming up with our own ‘method-chaining’ model. Instead of executing a single method to process the entire definition (i.e. JSON) or have multiple disjointed statements, we begin our class definition with a single method call followed by a series of chained calls. This approach preserves the context of previous method calls and can be used to implement modifiers (like in Java or C#). In a sense, we incrementally ‘build’ our class definition and finalise it by calling a terminating method.
The code below shows our two classes defined using method-chaining. An initial method call is given the namespace container and name of the class to create, thereby encouraging programmers to properly segregate their code with namespaces. After which a ‘builder’ is returned containing various ‘keyword’ methods that the programmer may call to incrementally define the class. A final call to the END_CLASS() method clearly marks the end of the class definition.

```javascript
tc.lang.Class.define(tc.demo, "Animal")
  .CONSTRUCTOR(function(name, isCute) {
      this._name = name;
      this._isCute = isCute;
    })
  .METHOD("getName", function() { return this._name; })
  .METHOD("isCute", function() { return this._isCute; })
  .END_CLASS();

(tc.demo, "Cat")
  .EXTENDS("Animal")
  .CONSTRUCTOR(function() {
    this.Super("Cat", true);
  })
  .END_CLASS();
```

Figure 8: The classes written in our method-chaining model, with the ‘keyword’ methods in red.

### 3.3.2. The Pros

Notice how the capitalised ‘keyword’ methods help to distinguish one member from another and increases readability to the point of being practically self-describing. The abstract nature of each method call means the implementation details are hidden, allowing for upgrades and fixes in the background without affecting existing class definitions. Verification can be performed as well, catching potential mistakes early.

Another advantage is that programmers are given the freedom to re-order the members in (almost) any way they please within the bounds of `define` and `END_CLASS()`, without having to worry about messing up the syntax, making it both a flexible yet cohesive technique. Most importantly, the model allows us to effortlessly extend the definition syntax by adding more methods on the builder, easily scaling up as new features are required.
3.3.3. **The Cons**

Despite all these advantages, the most apparent flaw is in loading time. Calling methods repeatedly to define a class, with each call possibly performing verification, can be a slow process. However, our experience has shown us that loading time is really quite negligible and the development of ever faster JavaScript engines and CPUs would probably make this concern irrelevant.

Another minor flaw is the bulkiness of the definitions. JSON-based models are used by many class systems (Qooxdoo just being one of them) because the definition code is very concise, making them suitable for web-applications since these would benefit from smaller download footprints. However, we felt that readability and maintainability would be much more desirable traits than shorter more cryptic code in our client-side application platform. In our system, code would be downloaded at most once and stored locally, so forcing ourselves to use a more ‘anorexic’ solution would not make sense.

Finally, the method-chaining model helps to promote runtime safety, but is ultimately unable to enforce it. Due to the dynamic and insecure nature of JavaScript 1.5, a class may still be modified after the definition has ended and true access-control for private and protected members is not possible. However, this problem is inherent in all other models as well and not just ours. To solve it, we would either need to wait for JavaScript 2.0 or come up with a very radical solution. One such possible solution is discussed in section 5.6.3.

3.4. **Implemented Features**

Utilising our extensible method-chaining model, we have developed a powerful class system over the course of this project. Borrowing ideas from popular application programming languages like Java and C#, developers feel comfortable with the system and can start using it without having in-depth knowledge of JavaScript. Feedback from users has been positive and most now cannot live without it.
This section showcases some of the more notable features of the system. A more comprehensive reference can be found in appendix A.

3.4.1. Namespaces
Hierarchical namespaces have been implemented as object containers, which are instances of tc.lang.Namespace. They are ‘built’ before use, either manually by the programmer or automatically by the bootstrap loader. The namespace addresses are required to conform to a specific format, ensuring they can always be properly accessed by name.

```plaintext
(tc.lang.Namespace.build(this, "sg.com.myspace"));
(tc.lang.Class.define(sg.com.myspace, "MyClass")
  // ...
  .END_CLASS();
```

**Figure 9:** Manually declaring and using a namespace container.

3.4.2. Classes and Interfaces
Besides vanilla classes, the class system also supports the creation of abstract, static and final classes, following semantics similar to Java and C#. Interfaces sporting multiple-inheritance are also supported. These features allow programmers to be as expressive in JavaScript as they would be in other object-oriented languages. The code snippet below shows how interfaces are declared and implemented in a finalised class.

```plaintext
(tc.lang.Interface.define(tc.demo, "Staff").END_INTERFACE();
(tc.lang.Interface.define(tc.demo, "Student").END_INTERFACE();
(tc.lang.Interface.define(tc.demo, "TeachingStudent")
  .EXTENDS("Staff", "Student")
  .METHOD("teach")
  .METHOD("learn")
  .END_INTERFACE();
(tc.lang.Class.defineFinal(tc.demo, "Bob")
  .IMPLEMENTS("TeachingStudent")
  .METHOD("teach", function() { /* teach something */ })
  .METHOD("learn", function() { /* learn something */ })
  .END_INTERFACE();
```

**Figure 10:** Declaring interfaces and implementing them in a finalised (non-inheritable) class.
3.4.3. Enumerations

Enumerated types with symbolic names and integer values are also available in the class system. These enumerations allow programmers to create lists of related items or constants in a readable convenient fashion. Bitwise enumerations are supported as well, where the automatically generated values are bitwise-mutually-exclusive.

```plaintext
tc.lang.Enum.define(tc.demo, "Languages")
  .ADD("English")
  .ADD("Mandarin")
  .ADD("Malay")
  .ADD("Tamil")
  .END_ENUM();

tc.lang.Enum.defineBitwise(tc.demo, "CoffeeWith")
  .ADD("Ice")
  .ADD("Milk")
  .ADD("Sugar")
  .ADD("Nothing", 0)     // <-- explicitly giving a value
  .END_ENUM();
```

Figure 11: Declaring normal and bitwise enumerations.

3.4.4. Nested Constructs

Collectively, classes, interfaces and enumerations are known as constructs. Constructs may be nested within classes and interfaces as static members, allowing for better organisation of certain compositional or aggregational relationships.

```plaintext
tc.lang.Class.define(tc.demo, "HashTable")
  .CLASS("Bucket")
  // ...
  .END_CLASS()
  .END_CLASS();
```

Figure 12: Nesting of one class inside another.

3.4.5. Methods, Fields and Constants

In figure 8, we have seen how methods can be declared by specifying its name and function body. Fields and constants can also be declared in a similar fashion by specifying a name and a value. Verification is performed when these members are added, ensuring that the programmer does not make trivial mistakes – like declaring two members with the same name.
3.4.6. Typed Properties

Properties can be seen as fields on steroids. Besides a name, they are also given a type (e.g. “string”, “integer”, etc...) and setter/getter methods are automatically generated for the property. These methods perform type-checking and conversion for the programmer, providing both convenience and runtime safety. Functionality was inspired by properties in C#, and similar control over setters and getters is given.

```
.tc.lang.Class.define(tc.demo, "Alice")
  // Immutable string property, Name
  .READONLY().PROPERTY("Name", "string", "Alice Tan")

  // Mutable string property, NRIC, with custom setter
  .PROPERTY("NRIC", "string")
    .SET(function(value, oldValue) {
      // further verify format of value
      return value;
    })
  .END_CLASS();
```

Figure 13: A read-only property and another with a custom setter defined.

3.4.7. Abstract, Final, Instance and Static Members

Members of constructs can be declared as **abstract** or **final**; the class system will verify that **final** members are never overridden and that all **abstract** methods are implemented in a non-abstract class. **Static** members can also be declared; these reside within the scope of the construct itself rather than in an instance of it.

3.4.8. Public, Protected and Private Access Modifiers

Members can also be declared as **public**, **protected** or **private**. However, as mentioned earlier, true access-control is currently not possible; instead, we use naming conventions and depend on the discipline of the programmer to follow them. For example, the names of all **private** members are prefixed with double-underscores (“__”), denoting their restricted access-level. Similarly, **protected** members are prefixed with a single underscore (“_”) and **public** members have no underscores prefixed. These allow programmers to easily determine the access-level of members by name and (hopefully) adhere to the access rules.
3.4.9. Annotations

One of the most powerful features of the class system is annotations. Any construct or member can be annotated with arbitrary data and this can be retrieved at runtime using reflection (see section 3.4.13).

```plaintext
tc.lang.Class.define(tc.demo, "Service")
  // The class itself annotated with an author
  .ANNOTE("@author", "Bob").THIS()

  // Deprecated static method with an informative message
  .DEPRECATED("Please use start() instead.")
  .STATIC_METHOD("run", function() { /* ... */ })
  .STATIC_METHOD("start", function() { /* ... */ })
ENDORCLASS();
```

Figure 14: Generic annotations and deprecation being used.

Some special annotations exist as well. Annotating a method as `DEPRECATED` will instruct the system to discreetly inform the user when the method is invoked. This allows developers to deprecate API and be assured that users will be informed of the change. Another special annotation is `TRANSIENT`; applying this to a member allows the system to know that the member should not be serialized (see section 4.4.4).

3.4.10. Member Groups

To facilitate applying modifiers (i.e. static, final, abstract, private, etc...) on many members, hierarchical groups can be created. These allow members to be logically organised by the programmer and makes the definitions more compact. If a group is given an optional name, the system will check that the pairing of `GROUP` and `END_GROUP` is correct.

```plaintext
tc.lang.Class.define(tc.demo, "Numbers")
  .STATIC().GROUP("num_grp")
  .CONSTANT("ONE", 1)
  .CONSTANT("TWO", 2)
  .CONSTANT("THREE", 3)
  .CONSTANT("FOUR", 4)
  .END_GROUP("num_grp")
  .END_CLASS();
```

Figure 15: A group of static constants in a class of Numbers.
3.4.11. Importing of Static Members

Many class systems allow importing or ‘mixing-in’ code from other sources when defining a class. Our system supports importing of static methods and constants into the definition of a class or interface. Importing many members (with names matching a regular expression) at a time is also supported.

3.4.12. Debugging With Method Tracing

Support for code debugging through the use of method tracing has also been implemented. By annotating a method with \texttt{TRACE}, the class system will track when the method is invoked, the values of the arguments given, when the method has completed, and whether it has returned a value or thrown an exception. This information can be retrieved and analysed by the programmer, allowing him/her to more easily find the source of a bug.

\begin{verbatim}
  tc.lang.Class.defineStatic(tc.util, "BuggyMath")
  .TRACE()
  .METHOD("gcd", function(m, n) {
    if (m < n)        return this.gcd(n, m);
    else if (n == 0)  return m;
    else              return this.gcd(n + 1, m % n);
 })
  .END_CLASS();
\end{verbatim}

\textbf{Figure 16:} Tracing a buggy method with a single call to TRACE.

3.4.13. Runtime Reflection

Like Java and C#, our class system has an API to query the meta-data of the constructs defined and their members at runtime. Combined with the dynamic capabilities of JavaScript, this effectively allows developers to employ \textit{reflective programming} – a very powerful technique.

For example, all the methods and properties on a class can be iterated at runtime, along with their annotations and modifiers. These can then be compiled into an online help system, fed into a documentation generator, or even converted into an interactive form (an actual sub-project in TiddlyCard!). The possibilities for the use of reflection are endless and these barely scratch the surface of what can be done.
3.5. Comparison with Other Systems

The table below summarises how our class system compares with other popular/notable systems available as well as with the latest JavaScript 2.0 draft proposal. In terms of features, we have a pretty good edge over other language extensions. Our closest competitor is Qooxdoo, which makes use of JSON-based definitions instead of method-chaining.

<table>
<thead>
<tr>
<th>Features and Capabilities</th>
<th>TiddlyCard</th>
<th>Dojo</th>
<th>JSClass</th>
<th>Qooxdoo</th>
<th>TIBCO GI</th>
<th>MooTools</th>
<th>JavaScript 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constructs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Classes</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Static Classes</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Abstract Classes</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Final Classes</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Singleton Classes</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Inheritance</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Multiple-Inheritance</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Constructors</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Interfaces</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Inheritance</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Multiple-Inheritance</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Mix-Ins</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Nested Constructs</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><strong>Members</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Overloading</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Fields</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Constants</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Properties</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Fine-Grained Setter/Getter Control</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Grouping</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Importing/Borrowing</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Abstract Members</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Final Members</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Static Members</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Access Control (public, protected, private)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><strong>Verification</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Dynamic</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Naming Convention Enforcement</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Namespaces</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Strong Typing</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Annotations</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Deprecation Support</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Reflection and Introspection</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Method Execution Tracing</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Dependency Declaration and Resolution</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

● Full support  ○ Partial or emulated support
4. The Data Layers: I/O, Files and Persistence

Being a client-side platform modelled after an operating system, TiddlyCard has to provide services like I/O and local file access to applications. High-level database-like storage is also required, since most applications would want a convenient way to persist arbitrary data instead of dealing directly with the intricacies of a local file system.

Therefore, an entire collection of APIs would be needed to facilitate the storage, retrieval and transportation of data. They would have to provide low-level to high-level functionality and follow the driver/provider model where possible to ensure portability and extensibility. This section discusses the various libraries created for this purpose as well as an important design principle to ensure efficiency and responsiveness in applications.

4.1. Asynchronous Design

Unlike conventional application programming on an (actual) OS, which assigns each application at least one dedicated thread of execution, applications written in JavaScript all share one single browser thread. As such, only one branch of code from a particular application is ever executed at any time. This has some advantages and disadvantages.

One advantage is that you do not need to worry about synchronization issues inherently associated with other (multi-threaded) platforms. If your code is executing, you can safely assume that no other concurrently executing code will interfere with it. This is the reason why the core JavaScript language omits the typical synchronization primitives (e.g. semaphores, monitors, locks, etc...) present in most other programming languages.

However, this doesn’t mean writing applications for a JavaScript platform is any simpler! On the contrary, sharing only one thread has a major downside for developers who want to author efficient and responsive applications in JavaScript – the necessity for writing asynchronous code.
4.1.1. A Tough Pill to Swallow

Essentially, asynchronous applications would have to voluntarily allow other applications to run by first scheduling the next sequence of code to execute at a later time or after some event, and finally relinquishing its hold on the execution thread. By performing these two steps at strategic points in code, the overall time taken to perform an operation increases, but the user is given the illusion of a significantly faster system since other applications are given a chance to execute. This approach is similar to the continuation passing style used in functional programming languages.

Unfortunately, this solution is a rather inconvenient one. Developers must have the discipline to write code in this manner, and most are not used to the style of passing around continuation functions. To make things worse, such code can be mind-bending to write and ensuring completeness and correctness becomes harder as the chain of calls becomes more complex. For this reason, many developers choose the easy way out by writing synchronous code wherever possible. Ironically, some applications claiming that they use AJAX (Asynchronous JavaScript and XML) actually use the synchronous option provided by the API – effectively defeating the purpose of the technology.

Despite the complexity involved, many developers agree that asynchronous design is crucial for writing responsive browser applications that may perform long computational tasks or I/O. Such a design becomes even more important in our platform where multiple applications need to share the browser environment.

4.1.2. Integration into the Platform

An interesting property of the browser is that with just one thread and no synchronization, it is basically impossible to use an API in a synchronous manner if it was designed to be asynchronous. In other words, if an API call performs its task at a later time (not in synchrony with the current thread of execution) then the caller cannot perform a blocking wait, since this would prevent the API from completing the task.
Instead, the caller must relinquish its hold on the execution thread and wait for the API to ‘call back’ with the results or confirmation of the task. This property is useful, as it allows us to ‘force’ developers to write asynchronous code when using our API.

However, writing the entire application platform in an asynchronous manner would not be realistic to both the development team and the end-users of the system. So to wean developers off the convenience of synchronous programming, we decided to apply this property where it matters most – in the APIs intrinsically dependent on I/O. We would want developers to learn how to use non-blocking I/O by making it mandatory in our system and this alone should help in ensuring that the platform stays responsive for most applications.

A further advantage of having an asynchronous API for I/O and is that we can utilise any backend (be it synchronous or asynchronous) available on a browser to implement our drivers. As mentioned in section 2.1.4, the browser environment is heterogeneous and we cannot expect all browsers to have a synchronous interface for specialised functions like accessing local files and performing I/O on them. With an asynchronous API, our system caters to browsers with only asynchronous interfaces as well, ensuring a more portable system.

4.1.3. Simplifying the Task

All the APIs discussed in the next few sections were designed to be almost entirely asynchronous, while still maintaining a certain level of convenience for the programmer. We also tried to be consistent in how our API calls are made, particularly in terms what the caller should expect after making a request and how the system responds when calling back. This should ensure developers do not get too confused and would allow them to adapt more quickly to this style of programming.

It might be interesting to note that there are other projects (outside of TiddlyCard) that have tried to simplify asynchronous programming in JavaScript by augmenting the language with syntactic sugar. This could be an avenue for future development in the project (see section 6.2).
4.2. Basic I/O Abstractions

The pseudo class diagram below shows the various components implemented in the basic I/O API. There are some pre-defined exceptions given, along with interfaces for readers and writers. Various reader/writer implementations are provided as well; including a pipe.

The design of our I/O abstractions was inspired very much by Java’s java.io package, which is well organised and makes good use of inheritance and other object-oriented techniques. With the class system in place, modelling and implementation of the I/O classes was fairly straightforward. However, not all ideas could be transferred directly from Java into the browser environment. Two significant design choices pertaining to this will be discussed in the next two sub-sections.

4.2.1. Strings Instead of Byte Arrays

Fundamentally, I/O deals with the reading and writing of byte or character data, so a decision had to be made on how to represent sequences of such values in our API. Unfortunately, the only primitive values available in JavaScript are number, string and boolean. Using an array of single character strings or numbers (a 64-bit floating point
number to be precise) to represent a sequence of bytes/characters would not be very efficient memory wise.

Instead, we choose to have the API accept and return data simply as strings. Even though they are immutable, JavaScript is fairly effective at string manipulation, and the user should be able to splice and concatenate data for I/O quite easily. Strings are also the most compact way to store both bytes and Unicode characters.

### 4.2.2. Coarse-Grained Non-Blocking Reads

When reading from a stream using the typical libraries in Java or C, the call to `read()` usually blocks until sufficient data is available or End-Of-Stream (EOS) is reached. As mentioned in section 4.1.2, this had to be changed to a non-blocking call since the underlying implementation could be asynchronous (i.e. the reading code needs to relinquish the execution thread so that more data can be obtained).

However, we found that the complexity in forcing every single `read()` to be asynchronous was tremendous, since each invocation of `read()` meant post-processing code had to be placed in a continuation. This would be too much to handle for developers, even if it did make reading code more browser-efficient. As a compromise, it was decided that an asynchronous `waitForData()` call would be added to allow for coarse-grained non-blocking reading. Calls to `read()` would be synchronous until the read buffer runs out of data, after which the code needs to invoke `waitForData()` to asynchronously buffer more. This makes reading code significantly simpler and still allows for non-blocking I/O.

|-------------|---------------------|-----------------------|
| header = in.read(); seg1 = in.read(); seg2 = in.read(); | in.read(function(h){ header = h; in.read(function(s1){ seg1 = s1; in.read(function(s2){ seg2 = s2; })}); }); | // Buffer till EOS in.waitForData{
| | | function() {
| | | header = in.read();
| | | seg1 = in.read();
| | | seg2 = in.read();
| | | }); |

**Figure 18:** Sync., fine-grained async. and coarse-grained async. reading – simplified comparison.
4.3. A Unified File-System

After modelling the basic I/O classes, we could then concentrate on developing the API for reading and writing local files. However, we wanted to take it one step further. By borrowing the ideas of *mounting* and *symbolic-linking* from UNIX, we developed a flexible unified file-system on which any minor file-system can be mounted as long as appropriate drivers are available; this includes the local file-system.

This gives us much freedom in organising our own logical hierarchy of directories for the platform, *independent* of the underlying operating system. Applications can assume that our file-system follows a standard “TiddlyCard” structure and need not worry about OS-specifics since the platform would be responsible for managing these differences. The following sections will discuss the design issues leading up to the completed file-system and briefly mention the implementations of three different file-system providers (i.e. drivers).

4.3.1. File Addressing Scheme

Before we could design our file abstractions and API, we first had to decide on how files are addressed in the system. After some research, it was determined that a URI (Uniform Resource Identifier) was the best cross-platform method of referencing a file, since any browser running on any operating system can reference a local file simply by specifying the “file://” schema followed by a URI path.

However, the URI specification states that the path components of a URI may only contain printable US-ASCII characters \[^3\]. This implies that any white-space, control or Unicode characters must be escaped with *percent-encoding* in order to be strictly URI compliant (as shown above).

<table>
<thead>
<tr>
<th>Raw Path:</th>
<th>Strict-URI Percent-Encoded Path:</th>
</tr>
</thead>
<tbody>
<tr>
<td>/C:/My Documents/こんにちは 100%.txt</td>
<td>/C:/My%20Documents/%E3%81%93%E3%82%93%E3%81%AB %E3%81%A1%E3%81%AF%20100%25.txt</td>
</tr>
</tbody>
</table>

*Figure 19:* A raw path containing spaces and Unicode characters, encoded as a strict-URI.
This encoding makes the resulting path quite unreadable, especially with regards to non-English characters. To solve this problem and make the paths easier for humans to interpret, we decided to use a relaxed-form of the URI specification so that only characters that might affect the semantics of the path will be escaped. These characters include ‘%’ and ‘/’, both of which have a special meaning and cannot be left un-encoded.

<table>
<thead>
<tr>
<th>Raw Path:</th>
<th>Relaxed-URI Percent-Encoded Path:</th>
</tr>
</thead>
<tbody>
<tr>
<td>/C:/My Documents/こんにちは100%.txt</td>
<td>/C:/My Documents/こんにちは100%25.txt</td>
</tr>
</tbody>
</table>

**Figure 20:** The more readable relaxed-URI version of the path, with only ‘%’ escaped.

We also adopted the convention of reserving the filenames “.” and “..” to refer to the current and parent directory respectively, since these have the same semantics in practically all operating systems.

### 4.3.2. Provider Framework

Next, we had to design the framework for file-system providers to follow, which would consist of a set of abstract interfaces. Through these well-defined interfaces, we would have a unified means of querying, reading and writing files cross all file-system implementations. Once again, the class system comes to the rescue by allowing us to define abstract classes for providers to extend, as shown below.

**Figure 21:** Simplified class diagram of the framework file-system providers need to follow.
Specifications for the File, FileReader and FileWriter classes were derived by studying the equivalent classes in Java and by analysing and unifying the file I/O API provided in both Internet Explorer and FireFox. This was important as it allowed us to effectively create a cross-browser interface for file I/O that would be rich enough to express the functionality of the native APIs in both these browsers – a compelling reason to use our platform when it comes to local file access.

4.3.3. Tree Data-Structure

The key issue in developing our unified file-system was representing it as a data-structure that could efficiently store and access paths to mount points and symbolic-links. As such, considerable effort was spent in crafting an ordered tree data-structure (most similar to a prefix tree), where each node represents a path to either a mount point or a symbolic-link redirection (never both), and every child of a node \( x \) must have \( x_{\text{path}} \) as its path prefix.

![Figure 22: A possible structuring of the file-system tree.](image)

In order to access the data-structure reliably, every path queried must be normalised by un-escaping each path component, eliminating the “.” and “..” components, and finally re-escaping the path using our predictable relaxed-URI encoder. This process ensures that two equivalent paths are always converted to the same canonical form and will therefore lead to the same node in the tree. For example, the paths “/local/C:/My Doc/Pictures/../” and “/local/./C:/My%20Doc” are both normalised to “/local/C:/My Doc”.

34
4.3.4. How It All Works

Bringing it all together, the file-system works as follows: An application requests to either query, read or write a set of files by passing their absolute non-canonical paths to our API. The system then normalizes the paths and resolves the provider for each of them. After which the requests are translated and forwarded to the appropriate providers, which in turn supplies the implementations of File, FileReader or FileWriter satisfying the request. These are wrapped with a façade (to ensure the application only sees the coherent unified file-system) before being asynchronously given to the requesting application for use.

![Figure 23: A request to open the directory "/user/tmp" being fulfilled; based on figure 22.](image)

4.3.5. Implemented Providers

The unified file-system is in actuality a path resolution and translation system on which other minor file-systems reside. In other words, it is useless without implemented providers. For this project, three providers following the framework discussed in section 4.3.2 were implemented.

Two of these providers are for mounting the local file-system under IE and FireFox. At the very least, this allows applications to perform local file I/O on the two most popular browsers – support for other browsers may be added later on. The third provider allows us to mount the equivalent of a virtual-disk – a file-system represented entirely in memory with JavaScript objects. This implementation is extremely useful as we can use it to emulate a local file-system in the absence of native browser drivers. As an added bonus, it also allows us to mount and explore any JavaScript object as if it were a file-system – a feature handy for debugging (see section 5.3).
4.4. **High-Level Persistent Storage**

In the past, support for file I/O would have been sufficient to keep application developers happy, but now we require a great deal more. Applications today are more advanced and often need to efficiently store and retrieve substantial amounts of complex data units – a task too specialised for developers to handle themselves. This job is usually delegated to a platform API for accessing a high-level storage device (e.g. a database) for persistence; examples include *Hibernate* and *JDBC* in Java, *ODBC* in Windows, and *Google Gears* in the browser. It would be essential for our platform to have such an API as well.

4.4.1. **Design Process**

Since strong support for objects was available (with the class system), we decided that a storage system similar to an *object database* could be implemented. Unfortunately, we had to solve the problem of retrieving an object *without* its object id. The solution came naturally when we combined the notion of an object database with concepts from modern web-applications. Meta-data in the form of tags and other queryable information could be attached to the object when it is stored, and can later be specified in a query to retrieve it. Tagging an object allows it to be classified into many different categories, with each tag basically being a group of related objects. From a relational database perspective, it would be the equivalent of a tuple residing in more than one table.

However, we wanted the system to be able to store data *other than* objects as well. For example, applications might want to save textual data like HTML documents or maybe even binary data like JPEG or GIF images. To realise this feature, we simply augmented the attached meta-data to include a *content-type* and *encoding* field compliant with MIME (Multipurpose Internet Mail Extensions) [5] – an established hierarchical standard for describing content. This would allow the API to distinguish one type of stored ‘object’ from another and more effectively determine how to load and save it. In effect, an actual JavaScript *object* stored (in contrast to textual or binary documents) would simply be another type of content.
With some research into various other browser-based platforms, it was also found that most of them refer to their persistent repository simply as a ‘store’. This term was adopted in our design, except that we could not just have a single store. Many different storage implementations could exist so we needed a model where stores are supplied by providers and applications may choose which provider they want. Although, given the many types of content (i.e. objects, documents, data fragments, etc…) that our system needs to support, it would not be wise for us to force these store providers to be responsible for recognising and knowing how to handle every type. As such, it was also decided that a separate marshalling layer would be required to convert the various content to/from a common storage format. This would make the task of implementing a provider much simpler.

4.4.2. Implementation Overview

![Diagram](image)

Figure 24: Overview of how an application would store and retrieve objects.
The diagram on the previous page illustrates how an application would go about storing and retrieving objects with our implemented system. The various stages are described as follows:

1. The application partially fills out a meta-data ‘form’ and attaches it to the object to be saved before passing it to the marshalling layer. This layer figures out how to ‘package’ the object for storage and might also add more meta-data describing the object or marshalling process used (e.g. “bear was compressed”).

2. After the object has been marshalled, the system then sends the packaged object (with meta-data) to a provider for storage. If successful, a ‘receipt’ is given to the application, confirming the meta-data given and stating exactly where the object was stored.

3. Optionally, the application may marshal the object and fill out the meta-data ‘form’ entirely on its own and send it directly for storage. This gives applications more flexibility if they prefer to handle the marshalling themselves.

4. To retrieve the object, the application provides its ‘description’ to the system by literally quoting some or all the information given in the storage receipt. The system will then locate the provider handling the object and forward the request for retrieval. If the application lost the receipt, it can query the system with a partial description and browse all matching objects for the one it wants.

5. Once again, the application may optionally take the marshalled object from storage as it is and handle the unpacking on its own.

6. Otherwise, it should request the marshalling layer to perform the extraction of the original object. Before extraction, the layer will examine the meta-data attached to determine the best way to unpack it (e.g. “bear was compressed, inflation needed”).

7. When done, the original object is given to the application along with the meta-data from storage (as a reference).

Now that we have seen how the system works in general, the next few sections will describe some notable aspects of it in greater detail.
4.4.3. Addressing Scheme

As with the file-system, our high-level storage system required an addressing scheme for locating stored objects. Once again, we based our design on the URI format \[3\] due to its attractiveness as a universally-supported addressing scheme. Basically, the address had to uniquely identify a provider, a store from a provider, an object within a store, and possibly a particular revision of an object (if version controlled storage is supported). By enforcing that the identifier for each component is URI path compliant, we could prefix each component with a ‘/’ and end up with an address quite similar to a file-system path. We chose to use ‘?’ to prefix the revision component instead of ‘/’ though, since seemed more suited for the query portion of the URI.

\[
\langle\text{provider_id}\rangle[\langle\text{store_id}\rangle[\langle\text{object_id}\rangle[\langle\text{revision_id}\rangle]]]
\]

**BNF:**

```
address := provider 
provider := / provider_id | / store_id provider_id store 
store := / store_id | / object_id store_id object 
object := / object_id | / object_id revision 
revision := ? revision_id 
provider_id := uri_path_component 
store_id := uri_path_component 
object_id := uri_path_component 
revision_id := uri_query_component 
```

*Figure 25:* Format of a storage address in compact notation and in BNF.

4.4.4. Marshalling and Object Serialization

As shown earlier, marshalling is the process by which an object (or any data for that matter) is converted to the storage format recognised by the system, while unmarshalling is the process of reconstructing the original object (or data) from the system storage format. In our actual implementation, these two procedures were named *saving* and *loading* respectively because ‘marshalling’ was considered to be a little too long. Similarly, the acts of storing and retrieving the marshalled object (step 2 and 6 of figure 24) were simply named *put* and *get* respectively.
Since many types of data had to be marshalled, an entire framework for assisting in saving/loading was created. This framework allows developers to add new handlers for various types of content with each handler (un)marshalling in a different way. For example, a handler for “text/plain” content was implemented which could marshal a text string into various encodings, including UTF-8 [25] and Base64 [5]. The idea would be to allow applications to choose the best handler for their data.

Of course, we cannot forget that our storage system was initially designed to be a repository for objects. We needed a special class of handlers that know how to save and load objects from our class system as well as all the objects and values native to JavaScript. In other words, we needed handlers that could perform object serialization and deserialization. As such, we added yet another framework to the storage system for installing pairs of object serializers and deserializers. Having a pluggable framework for both content handlers and the SerDes (contraction of SerializerDeserializer) sub-type gives our storage system unrivalled extensibility.
Since object serialization is a key part of convenient persistence in most object-oriented languages, we immediately set out to write a SerDes based on Java’s serialization model. In summary, a binary serializer (and deserializer) was written that can handle any object from our class system and all native JavaScript objects/values, detect circular object references, ignore any object members marked as *transient*, and invoke pre-serialization/post-deserialization routines on the objects. Unlike Java though, an object need not implement a *Serializable* interface to be accepted for serialization. This binary SerDes is one of the most powerful features of the platform, since it gives us the ability to convert almost any runtime JavaScript object into a flat string of bytes and back. The framework for SerDes pairs was also designed to be de-coupled from the primary storage API, meaning they can be used to serialize objects for transportation over the network or for writing formatted binary files as well.

![Figure 27: Illustration of the SerDes framework de-coupled from the storage API.](image)

### 4.4.5. Store and Provider Framework

The marshalling layer is important, but the business-end of the storage system is still a store from a provider. Some requirements of a store include *getting, putting, querying* and *removing* of marshalled objects. Providers have to allow *creating, opening* and *destroying* stores. They also have to be able to describe their capabilities to the system, so that applications may select the best provider to use. Like to the file-system provider framework, we chose to define abstract classes for providers to implement, which would clearly specify the interface methods that both a provider and its stores have to provide. On top of these classes, we added a bitwise enumeration of store/provider attributes, allowing them to describe capabilities by simply OR-ing attributes together.
4.4.6. Query Handling

In the framework above, we see that stores must support execution of queries to retrieve objects. A query would analogous to an SQL statement, specifying constraints on the meta-data of objects desired. This raises the question of how a query should be constructed – what language and format should it be in? In any case, it had to be as expressive as possible, extensible, and yet easy for both applications to use and stores to process. After some trials, we found that *method-chaining*, the technique used in the class system, could also be applied to query construction and satisfy these requirements. An SQL-like string-based query was also considered, but this would have required every provider to implement a parser for the query language; possibly over-complicating the implementation of providers. The figure below shows how a query would be constructed in our implementation.

```javascript
var query = new tc.storage.Query()
    .withName("My Favourite Bear")
    .withTags("teddy", "bear")
    .withoutTags("roosevelt")
    .withContentType("application/x-teddy");
```

We can also specify various ‘actions’ to be taken on the objects matching the query, other than just retrieving them. For example, we can write the equivalent of an SQL DELETE statement by specifying a secondary action “Remove” or perform a COUNT aggregation function by changing the primary action to “Count”, as shown in figure 30.
As more constraints are added, the fields of the query object are simply updated to reflect them. When passed to a store for execution, providers can then simply read off the query object and search for matching results. Providers implementing stores using an actual database backend can also easily convert the information in the query object to the query language of the database (e.g. SQL, XQuery, etc...) and essentially perform a fast native lookup to get the results. While not nearly as expressive as SQL right now, this construction technique is easily extended and more features can always be added later.

4.4.7. Implementation of a File-Store-Provider

Once the infrastructure for the storage system was established, we could then focus our efforts on writing a provider to produce stores; without which the entire framework would be useless. To make full use of the already working file-system and file I/O API, we decided to implement a provider that would use files as a means of persisting data.

The file-store-provider implemented has a fairly simple design: A store (and its contents) is represented by two files. When a store is first opened, the provider caches the contents of the latest file into memory and uses this cache for getting/putting as the store is used at runtime. Whenever instructed to flush the cache, the provider alternates writing between the two files – ensuring that a backup is always available in case a write is interrupted. This implementation is fast (due to caching), error resilient, and fault tolerant. Its drawbacks are that it is not optimised for storing large amounts of data since everything is cached in memory, and query performance suffers since no indexing is performed. These faults are acceptable though, since the file-store-provider was intended as simply a reference implementation for future providers to follow and not a production-quality storage provider.
5. Completing the Picture

At this point, we have only seen the class system and the data layers, sub-systems of TiddlyCard meant to provide applications with a high-level programming language, I/O, files and persistent storage capabilities. Now we will introduce the minor sub-systems implemented and other concepts from our plan in section 2.3, which would give us a complete and working application platform. First and foremost, we shall examine the framework for directly supporting and managing application and processes.

5.1. Applications and Processes

![Diagram of Application and Process Framework]

Figure 31: Overview of the application and process framework.

The illustration above shows the main components of the application/process framework and how they are related. The numbered relationships and their associated components are described as follows:

1. Applications are defined by sub-classing from an abstract `Application` class, and are executed simply by instantiation (i.e. the constructor is the ‘main’ function). When executed, the application instance is assigned a `process` with a unique identifier, which represents the running application. Both the application instance and its process are tightly coupled until the process dies.
2. The running application (interchangeably, ‘process’) may schedule asynchronous tasks to be run. In this sense, the scheduler involved is not a process scheduler, but rather simply a task scheduler, where a ‘task’ is code to be executed at some later time in the context of the process scheduling it.

   Basically, processes tell the system when and where they would like to continue execution (permitting other processes to run in-between), instead of the system allocating each process execution time. Multitasking has to be done in this cooperative manner since the limitations of the browser (i.e. single-threaded, non-pre-emptible, etc…) does not permit true process scheduling.

3. Some facilities are available for Inter-Process Communication (IPC), which were inspired by the UNIX programming environment. Although applications could use global variables in JavaScript for communication, this method would eventually cause problems when many applications fight over the global real-estate. To discourage this practice, some formal means of IPC had to be implemented; these are discussed in section 5.2.

4. One of the problems identified with the high-level storage API was that after an application has created a store (to persist objects), it needs to be able to retrieve the same store in subsequent sessions by providing its address; meaning the address itself had to be stored somewhere else. This and other issues made the storage system rather inconvenient to use, so we decided to directly integrate support for persistent storage into the application framework using the storage API as a backend.

   This allows stores to be created easily by applications, with the system automatically tracking and loading them the next time the application is started. Support for saving user settings and other minor application data has also been added (i.e. similar to the Windows registry, but easier to use), which makes the job of adding persistence to applications much simpler.
5.2. **Inter-Process Communication**

Support for IPC in our platform was inspired by UNIX, which has pipes, signals, memory-mapped files and network sockets among other communication features. Given the complexity of the latter two, we decided to just implement *pipes* and a variation of *process signalling*.

Pipes pretty much come for free since an implementation is already part of our basic I/O abstractions. They can be set-up so that one process treats a pipe as its output and the other as its input, allowing string data to ‘flow’ from one process to another. Note that system-wide *stdout* and *stderr* streams are also available for applications to use; these are essentially pipes connected to the current console (see section 5.3).

In our system, processes may signal one another with any string (accompanied by an optional set of arguments), in contrast to UNIX where only integer signals are possible. A signal simply invokes a specially named event-handling method on the application class, notifying the application if it has implemented the method to respond to the signal. Some system-defined signals include “show”, “hide”, “close” and “kill”; the last being raised when the process is terminated.

5.3. **Console and Shell Interpreter**

Thanks to the efforts of *Michael Clark*, a long-standing external contributor to the TiddlyCard project, the prototype of our platform had a console window where applications could print messages for debugging and/or logging purposes. This console was also integrated with a simple command interpreter, which allowed us to navigate and examine the ‘global object space’ of the browser like a file-system and run programmer-defined commands. As the TiddlyCard platform slowly came together, the console (which we found to be exceptionally useful during the course of the project) began to show signs of age and increasing incompatibility. We knew that eventually something had to be done to bring the entire console system up-to-date.
With the completion of the class system, unified file-system, and application framework, we decided to upgrade the console to give it a whole new look and implement a proper shell interpreter utilising all our established architecture. By doing this, we hoped to demonstrate the power of our platform API and give developers a feature-rich console system to play around with.

The screenshot below shows the new (Turtle) shell interpreter running in the upgraded console window and accepting commands to explore the unified file-system. Part of the boot process (see section 5.5) for starting the TiddlyCard platform is also displayed. With all these features, our platform is given the distinctive feel of a typical operating system.

As we have pointed out, the console was worth upgrading because it has been an indispensable tool for TiddlyCard developers in debugging and managing our libraries and applications. Method traces (see section 3.4.12) and deprecation warnings are printed to the console, allowing us to easily locate the source of a bug and be notified of outdated API at runtime. Additional commands can also be added by developers to launch or configure applications and platform libraries.
Figure 33: The method tracing example from section 3.4.12 (with added deprecation) being tested.

Below is a quick summary of the many features and interpreter commands implemented in the upgraded console system.

Features:
- Saving and browsing of command history.
- Stdin/Stdout/Stderr streams for applications to use.
- Colours and output formatting using ANSI escape codes [22].
- Mounting of the local file-system on "/local/root".
- Symbolic linking of "/local/home" to the startup directory.
- Mounting of the JavaScript global object on "/global".
- Compatible with IE, FireFox and Safari for Windows.

Interpreter Commands (inspired by UNIX):
- Listing of all available commands with descriptions (help).
- Standard file-system functions (cd, pwd, ls, rm, mkdir, cat).
- Managing of mount points and symbolic links (mount, link).
- Listing and signalling of running processes (ps, signal).
- Starting and exiting the shell interpreter (turtle, exit).
- Manipulating the JavaScript environment (turtle, exit).
- Runtime benchmarking of code (bench).
5.4. Deploying Software Units

Every modular platform needs to have a system for managing, distributing and installing units of software (e.g. applications, updates, drivers, etc...) as developers would want to be able to deploy their software quickly and easily to end-users. In fact, the success of the platform itself can depend very much on the efficiency of its deployment mechanisms. After all, if developers find it a hassle to distribute their software and target users have trouble finding and installing them then hardly anybody would want to use the platform.

With this in mind, deployment has been a key topic in many of our project discussions. However, a good deployment system could not be created without first establishing a solid foundation for our platform – the goal of this very project. As such, the job of creating a full-fledged deployment system had to be left to the next generation of TiddlyCard developers. We cannot neglect to mention though, that out of necessity, a rudimentary system for deployment was implemented.

In the early stages of the project, developers were loading their applications/libraries into the system by specifying the required resources file-by-file, which made it very troublesome to maintain. To ease our suffering, a system was created whereby groups of related resources (i.e. JavaScript files, style-sheets, etc...) could be composed into ‘plug-ins’, each of which could then be loaded into the system as a whole, rather than in pieces like before. Each plug-in is defined by a descriptor file that specifies a list of resources (which may include other plug-ins), their loading order, and other information.

This system basically gives us a way of binding together a group of related resources with meta-data; in other words, a way to define a unit of software. Indeed, we already have a simple (albeit inadequate) deployment system on our hands. To polish it off, automatic resolving of dependencies among units would be required, along with a dynamic loading system and one or more techniques for packaging and distribution; these we leave for future TiddlyCard projects to accomplish by taking full advantage of the foundations we have put in place.
5.5. Booting and Shutting-Down

Any operating system needs a boot procedure to get going and a shutdown procedure to save user data and perform clean-up tasks. Since we chose to follow the OS model for our platform, these steps naturally had to be included in our implementation.

Booting the system was the hardest part, since we had to determine the smallest subset of the platform required for everything else to function. These efforts culminated in the creation of a bootstrap loader which would emulate key platform functions until they can be replaced by loading the actual libraries. On boot initiation, the loader will read the boot sequence specified in the HTML file and load each resource into the system. When done, the application and process framework (i.e. the system core) is notified to continue a high-level start-up of the system, after which any start-up scripts are executed and applications may run.

A shutdown is initiated either by the user navigating away from the page or in code (by notifying the system core). This triggers any shut-down scripts to execute and all processes to be killed, which in turn informs applications to save their data and perform clean-up. The system then waits for all scheduled tasks to complete before notifying the data layers to commit any out-standing writes. This final step ensures that all data is saved and signals the end of a proper shutdown.

5.6. Resolving Security Issues

In section 2.1.5 and 3.1.1, we explained how the browser environment might not be very suitable for implementing an application platform due to its insecure nature. Despite this conclusion, security was still a major concern when developing TiddlyCard, and many strategies were devised for protecting the user from malicious applications, ensuring applications cannot interfere with each other or the platform, and securing JavaScript code in general. However, due to their complexity, there was insufficient time to fully implement any of these measures. This section explains these techniques, in order of increasing completeness, and how they should theoretically help to fully secure the system.
5.6.1. **Context Tracking**

One aspect of security in most platforms would be to prevent unauthorised calls to restricted code by determining which application process is responsible for a call. For example, easily abused API like local file I/O and networking could have an ACL (Access-Control-List), where only certain processes are granted the right to use them. Tracking the execution context (i.e. identifying which process is currently executing) is necessary for applying this form of access control.

Context-tracking is made simpler in JavaScript since, by virtue of its design, there is only ever one thread of execution and interleaved execution from concurrently executing code is practically impossible. With these attributes, we can track the currently executing process just by updating a variable when an application starts running and whenever a task is executed or scheduled for the application. Security would depend on reliably maintaining this state.

This is only a *weakly secure* solution though, since it depends on the assumption that applications are unable to access the variable tracking the context and have no other means of performing the functions of the restricted API. Unfortunately, the dynamic and uncontrollable nature of JavaScript means there are infinitely many ways for applications to manipulate the system, breaking this assumption. For example, they can either find a reference to the variable storing the context (changing it to elevate privileges), or even circumvent our platform API entirely and directly interact with the API in the browser. To close these loopholes, we need much stronger measures (see next two sections).

5.6.2. **Scope Chain Insertion**

In platforms with native support for protecting privileged code (e.g. SecurityManager in Java, Kernel mode in CPUs, etc...), enforcing security is much easier since *passive* techniques can be employed; where system code is structured to prevent malicious applications from harming the system or each other. JavaScript does not have such support, so security techniques which are passive (context-tracking
included) are limited as far as enforcement goes. The next logical step is to take an active approach to security, where application code is modified in order to assert control over what it can do or access.

Scope chain insertion is one of the simplest forms of active protection. This technique involves wrapping application code with an additional enclosing scope, which would block access to certain global variables or functions. Take the following un-modified application code for example:

```javascript
var myscope = { "ActiveXObject" : null, "myscope" : null }

with (myscope) {
    tc.lang.Class.define(tc.example, "DeadlyApp")
        .EXTENDS(tc.core.Application)
        .CONSTRUCTOR(function() {
            new ActiveXObject("Scripting.FileSystemObject")
                .GetFile("C:\\Windows\\regedit.exe").Delete();
        })
        .END_CLASS();
}
```

**Figure 34:** Malicious application which deletes a file through the IE ActiveX API \(^9\) when executed.

The above code is proof that serious damage can be done by a malicious application if it has direct access to the browser APIs. We can prevent this by enclosing the code in a new scope that overrides access to "ActiveXObject" before loading it into the system.

```javascript
var myscope = { "ActiveXObject" : null, "myscope" : null }

with (myscope) {
    tc.lang.Class.define(tc.example, "DeadlyApp")
        .EXTENDS(tc.core.Application)
        .CONSTRUCTOR(function() {
            new ActiveXObject("Scripting.FileSystemObject")
                .GetFile("C:\\Windows\\regedit.exe").Delete();
        })
        .END_CLASS();
}
```

**Figure 35:** The same application nullified by the addition of a new scope.

The `with` block in the modified code overrides the global "ActiveXObject" class with a `null` value, effectively preventing the application from using it. The scope object inserted (`myscope`) blocks itself as well, preventing the application from undoing the blockade. This technique is possible in JavaScript because the language allows dynamic evaluation of code.
There are numerous inadequacies associated with scope chain insertion though. For one thing, applications might anticipate the scope insertion and use inverted braces (\{...\}) to avoid being trapped in the new scope. Furthermore, this solution fails completely if the application finds other ways of obtaining a reference to the JavaScript global object; which is actually a fairly easy thing to do! Many other issues are present, but these two already show us that a more radical solution is needed.

5.6.3. Code Parsing and Transformation

Simple active protection by blindly modifying application code (as in the case of scope chain insertion) is similar to locking up a criminal without first checking if he/she has a gun – a method you do not want to put your faith in. Unfortunately, locating a ‘gun’ in an application is not as simple as it sounds. JavaScript is an extremely flexible language and code can be obfuscated in infinitely many ways, allowing malicious applications to hide their intentions until the very last nanosecond.

A complete solution must work at the language-level and constantly monitor the application for errant behaviour. This can only be done if we perform low-level JavaScript parsing and apply modifications which will transform the code into a version that dynamically checks and prevents runtime security violations. With this technique, it should be possible to enforce any security policy we want, including guarding of private and protected variables; provided that an appropriate set of code transformations are defined to add the necessary runtime checks. For example, we can prevent access to the global “ActiveXObject” class (in figure 34) by applying a transformation rule which will check and deny access to this class whenever a variable is used in code. Needless to say, these checks would have a big impact on application performance, so it might be a good idea to only use this technique as a means of quarantining untrusted applications until they are deemed safe.

While extreme, this proposal is the only one that can possibly resolve all the security issues present in JavaScript 1.x (without waiting for version 2.0), and considerable progress towards a working implementation of it has been made in another TiddlyCard project (H041240 by Zhou Lin).
6. Conclusions

It has been a huge challenge working on the architecture of an application platform as ambitious as TiddlyCard; skills and knowledge from areas as diverse as software engineering, web-technologies, programming languages, operating systems, database design and system security all had to be brought together and applied. The characteristics and limitations of the browser environment posed many hurdles as well, requiring both systematic analysis and innovation in problem solving to overcome. We shall now conclude this project by summarising the achievements made and providing recommendations to further advance the evolution of TiddlyCard.

6.1. Summary of Achievements

In section 2.3, we laid out a grand plan for TiddlyCard, with the goal of this project being to establish the foundations of the platform by designing and implementing the critical components necessary to give current and future developers a dependable framework to build upon. We have most certainly met this goal, with the following list of achievements in both implemented components and research:

- **Class-system for JavaScript** with namespaces, classes, interfaces, enumerations, annotations, reflection and many other features.
- **Asynchronous design** of all the data layers for efficiency.
- **I/O abstractions** similar to Java.
- **Unified file-system** with mount points, symbolic links, a provider model for adding new file-system drivers, and three different driver implementations.
- **Database-like high-level persistent storage** with generic object serialization, a marshalling framework, a provider model for adding new storage implementations, query handling, and a storage provider implementation based on files.
- **Support for applications and processes** with a well-defined framework, task scheduling, integration with persistent storage, and methods for inter-process communication.
• **Console and shell interpreter** with command history, access to stdin/stdout/stderr streams, output formatting (using ANSI escape codes [22]), and many implemented interpreter commands.

• **Defining of software units (plug-ins)** for loading and deployment.

• **Procedures for booting and shutting-down the system.**

• **Techniques for securing the platform.**

### 6.2. Recommendations for Future Work

The foundations we have laid would definitely give future developers a head-start in our application platform; there are flaws though, and much more can be done to improve the architecture. Here are just some ideas for future developers of TiddlyCard to consider:

**Security Layer** – This component is currently non-existent and malicious applications could easily pose a threat to the system. Using the proposal in section 5.6.3 and pioneering work into a JavaScript parser and transformer (a TiddlyCard HYP by Zhou Lin), a complete set of transformation rules could be defined for securing any application and implemented into a working security layer for the platform.

**Deployment System** – Even though we have a way for defining software units (plug-ins) in the system, a complete deployment system for packaging and distributing these units is still missing. This should be implemented, as it is essential for the success of the platform.

**Platform Reflection and Driver Loading** – There is currently no API support in our system to reflect on the capabilities of the underlying ‘hardware’. Such an API would be extremely useful to have, since driver libraries are often very platform specific and this information is needed to know whether or not a driver can or should be loaded.

**Simplifying Asynchronicity** – By using the same code transformer created for security, syntactic sugar could be added to JavaScript, which can then be translated to proper asynchronous code before being loaded. This would make it much easier for developers to write asynchronous applications. The same technique is employed in *Narrative JavaScript* [11] and *Clorox* [13], both of which are projects un-related to TiddlyCard.
References


http://www.ietf.org/rfc/rfc2396.txt

http://sourceforge.net/projects/jsclass

http://www.ietf.org/rfc/rfc2045.txt


http://www.neilmix.com/narrativejs/doc/

http://developer.mozilla.org/en/docs/XPCOM

http://clorox.csail.mit.edu/


Appendix A

TiddlyCard Class System:

Quick Guide & Reference