Sustainability for Digital Humanities research software

Graham Klyne, Oxford e-Research Centre, University of Oxford

7 July 2021

An exploration of sustainability issues in digital humanities research software using the Music Encoding and Linked Data (MELD) framework and applications as an exemplar.

Prepared for the Software Sustainability Institute, SSI phase 3 .

Table of contents

Summary of observations and conclusions

This section contains a précis of key observations and recommendations concerning sustainability arising from our work with Music Encoding and Linked Data (MELD) software, with references to more detailed description and discussion.

Observations

Each observation heading is linked to further discussion in this document.

- Complexity of supporting software environment

  A complex, dynamic supporting environment can bring great benefits to a project in terms of ready-to-use functionality, but may also come with a price to pay when it comes to long-term sustainability. Choose carefully!

- Working with higher level abstractions

  Undocumented abstractions can increase rather than reduce developers' workload.

- Callbacks and Promises

  Asynchronous working ("Don't call me, I'll call you."), and local *vs* global simplification. Judicious use of abstractions (like Promises) can simplify a codebase, but also introduce new impediments to learning it. Choose wisely and, where possible, use existing design patterns.

- Browser vs non-browser code

  For web applications, there is a fundamental choice to make. Browsers provide a rich environment for making responsive, user-facing interfaces, but (like other supporting software environments) using advanced capabilities can affect sustainability.

- Undetected problems in MELD code

  These problems underscore the value of automated tests. Often, the hardest part of fixing a problem is finding it in the first place.

- Hello MELD application series

  Simple sample applications are useful for both helping new developers to use a system, and also to support testing.

- Value of part-time technical expertise

  For projects that don't justify a full-time software developer, having occasional but ongoing access to software engineering expertise can be useful, both for the project itself, and also for capacity building within the project.

- Distributed collaborative working

  Remote collaboration on software development is hard, especially when there are no established anchor points that everyone can build from. Good communication tools can help, but are not a panacea.

Recommendations

Each recommendation summary heading is linked to further discussion in this document.

- Recommendation 1: Technical architecture

  Keep it simple; separate concerns; avoid complex user interfaces if possible

- Recommendation 2: Design for testing

  Testing doesn't cost, it pays.

- Recommendation 3: Continuous integration

  Fail fast: find out quickly when something breaks

- Recommendation 4: Incremental development

  Grow a system in small manageable pieces - it's surprising how quickly they accumulate.

- Recommendation 5: Dealing with technical debt

  Slay ghosts in the code before they slay you.

- Recommendation 6: Keep dependencies up to date

  Don't get left behind when depending on a dynamic supporting ecosystem.

- Recommendation 7: Minimal application examples

  No application is too simple to convey a useful lesson - simple lesson's are often the hardest to learn.

- Recommendation 8: Give some thought to project governance

  Establish norms for team interaction and quality expectations.

1. Introduction

The SSI3 "Software Sustainability for Digital Humanities" activity explores software sustainability, with a focus on testing, using the MELD framework as an exemplar. We aim to provide and document concrete sustainability benefits for the Music Encoding and Linked Data (MELD) project, and suggest how other Digital Humanities (DH) projects might realize similar benefits.

Sustainability is a perennial problem for research software. The funding of research projects means that there is often little or no resource available beyond the lifetime of a project to keep the software outputs running and available. When such outputs continue to be available, it is often by virtue of enthusiasm and dedication of individual researchers, rather than a wider framework in which sustainability is expected and facilitated.

These concerns are particularly acute with many Digital Humanities (DH) projects. Funding for such projects is often very thin, and software is often not the most significant academic output. Yet, increasingly, DH research depends on drawing information from multiple sources, often created by diverse projects, so considerations of sustainability become more important for continued progress of research.

In this activity, through hands-on experience of looking at sustainability of the MELD framework, we aim to identify areas which give rise to sustainability

problems, and steps towards sustainability that can be adopted even by projects operating with shoestring software development resources.

The intended audience for this report includes: digital humanities researchers, research software engineers engaged with digital humanities projects, and digital humanities project funders.

1.1 About this report

This report includes observed phenomena, subjective personal opinions, and widely accepted practices about the development and maintenance of a software system.

> Software sustainability arises at the interface between technical environment within which it is performed, and the personal involvement of software developers who create and sustain the software. As a research software engineer, I feel that it can be useful to highlight subjective issues faced by a working developer. My personal subjective views are presented in the style of this paragraph.

The main body of the report is structured as:

- Discussion of the characteristics of Digital Humanities research software, and how they may affect sustainability of software outputs.
- Case study observations arising from attempts to test Music Encoding and Linked Data software. Some of the issues encountered required some revision of the original plan, and as such provide useful insights into the effect of sustainability issues on software development and maintenance. Some possible mitigations that are specific to the case study are suggested.
- A discussion of the lessons arising from our case study, and how they may apply more widely to other DH research.
- A distillation of recommendations and considerations for software sustainability, particularly in the context of Digital Humanities research.

1.2 Abbreviations and technical terms used

- DH - Digital Humanities (research)
- MELD - Music Encoding and Linked Data: project and software framework
- SSI - Software Sustainability Institute: see https://www.software.ac.uk/about
- SSI3 - EPSRC-funded phase 3 activity of SSI: see The UK Software Sustainability Institute: Phase 3

1.3 What is software sustainability?

There is much discussion of software sustainability, but it's hard to find a consensus definition of what it means. It has been described thus: "Software sustainability covers a broad range of concepts, related to both environmental sustainability, and the longevity of a codebase." [1].

[1] What Makes Research Software Sustainable? An Interview Study With Research Software Engineers.

> Oddly, I couldn't find any discussion of this on the SSI web site. For the purposes of this activity, "codebase" is taken to include data that may underpin research results. It may well be that longevity of research data is more important than longevity of the code that creates or processes it.

In theory, when a codebase and all its environment dependencies are frozen, the resulting system will keep running indefinitely. In practice, there are many forces that work against this ideal: one of which is the need to apply a never-ending stream of security updates to any system that is accessible from the public Internet (or from a large private intranet). Over time, the underlying system can change to the point that any software running on it also needs to be updated in order to keep running.

Other disruptive forces include hardware failures that necessitate re-installation of the entire system. Even if a complete copy of all the installed software is available, it is possible that there are underlying changes in any replacement hardware that require the operating system to be upgraded, which in turn may be incompatible with the application software. Similar problems can arise when running on virtualized and/or cloud software.

> My experience is that a software system can be kept running for up to 5-10 years without being updated, and sometimes less.

For a longer active deployment life, an active process to keep all software components current is likely needed. In general, it is easier to upgrade in several small increments rather than wait for a forced upgrade and then have to deal with multiple incompatibilities that may interact in subtle ways. Upgrading is much easier when there is an extensive test suite in place. A test suite can help to pinpoint any failures that occur as a result of an upgrade, which is often the biggest problem faced when trying to fix them.

See also: Why software is like a puppy, which contains this quote attributed to Carole Goble: "Tell your funders and PI's that software is like a puppy. Puppies aren't free, you got to feed them, they will get old, and they will die. Funders don't understand software – but they know about puppies."

2. Characteristics of DH research software

There are some characteristics of DH research software that can make its sustainability a different proposition than for, say, commercial or scientific software. This may be in the nature of the task the DH software tries to accomplish, or may be the environment in which it is created and used.

Tasks:

- Digital humanities research is often data intensive (as opposed to computation intensive).
- Humanities research typically draws upon and combines diverse data sources.
- Digital humanities research data are often heterogeneous, describing a variety of subjects with variable degrees of detail.
- Valid primary data, such as historical accounts, may also be contradictory.
- Data to be presented or processed may be non-definitive, needing context and human interpretation in order to understand what they convey.
- Data are not primarily a scholarly end in themselves, but used to support analyses by human researchers.

Environment:

- Shoestring funding. Also, short-term funding (e.g. months rather than years).
- Unclear initial software requirements - the role of software in DH may be to facilitate exploration of data with a view to forming and evaluating scholarly conclusions. Such software may be written *ad hoc* by researchers without any software engineering training.
- Difficulty of obtaining funding for software maintenance that is not directly linked to new research; yet one of the reasons for sustaining software is to allow researchers without software development skills to benefit from existing developments.
- DH scholars and software engineers have diverse perspectives, all of which contribute to achieving useful results. The requirements for long-lived research software need to address both scholarly needs and engineering constraints.
- Research collaborations are commonly spread over multiple institutions, often in different countries.
- Software development is a skill in high demand, and it is common for software developers to leave for better-funded work. This can result in loss of knowledge that is needed to work on the software. And software maintenance can be tedious, uninteresting work; many developers would rather work on new software in which they have a greater sense of ownership. Loss of developers can be mitigated by attention to good engineering practices, but may need technical effort that is not costed in a research proposal.

- Scholars, who may themselves be able to write small programs for a specific purpose, sometimes fail to recognize that "You just can't do things for no money", especially when it comes to writing software to serve multiple users over an extended period of time. The effects of a lack of engineering resources may then be perceived as lack of competence.
- The Verovio design principles note that "digital humanities projects [...] have slow development cycles in comparison with the development of the technology itself".

  Of course, not all DH software is like this, and some scientific software shares some of these characteristics, but on balance it is my experience that these characteristics are more typical of DH software.

A particular consequence of the environment in which DH software is created is that it may be unrealistic to fully employ software engineering best practices. Some functionality may be required quickly to support a particular scholarly output, with no immediate requirement or additional resource to ensure sustainability of the software. Yet even in these circumstances, some regard for software engineering best practices may be rapidly repaid in time saved in debugging and enhancing a software tool, especially if it is to have any life outside the project for which it was created.

Smaller research software projects in all disciplines (and many DH research projects are small) are usually quite different from the commercial software development environments in which many software engineering practices have evolved, in that they do not have a permanent staff that is using an established set of tools and development environment. Researchers may spend a small part of their overall time doing software development, or research software engineers may be switching between projects that require different developer tools. Many software engineering practices implicitly assume there is an established development environment, with a range of advanced supporting tools to mitigate many of the problems that are noted later in this report, and that developers are all familiar with these tools. The overhead of establishing and managing some of these tools may well prove to be too much for a small research project. While this report suggests a number of areas where additional tools may help, it does not assume a large, homogeneous, maintained software development environment that is used by all developers; it focuses more on recommendations that may be applicable to a fragmented and heterogeneous development environment.

Because of the value in DH of capturing context and combining a wide range of data sources, it might be argued that sustainability is even more important for DH software than it is for, say, scientific software. Once a computed scientific result has been established, there may be limited value in revisiting it later. But humanities research tends (by its nature?) to gain value through the accumulation of understanding and perspectives over time, and as such, access to and integration with previously accumulated data can greatly increase the value accrued by investing in software sustainability, or at least sustainable access to generated data.

The foregoing points to a related sustainability issue: software vs data. Sustainability is often focused on software programs, and less on the data they manipulate - and such is the focus of this report. Yet it is often the case that the real value in both scientific and humanities software-based research is in the data. The scientific community (notably life sciences, astrophysics) have responded by creating a number of domain-specific data repositories to facilitate data sharing. There are similar initiatives for humanities, but diversity and heterogeneity of subjects and data mean that central repositories with broad utility are challenging to create. With regard to data sustainability, for which many issues mirror those around software sustainability, data management has been a live topic in the academic community for some time. There are many projects dealing with research data management, and related issues of replicability and reproducibility of research results (e.g. see resources from DCC, JISC, OCLC, etc.). Research Data Management and Research Software Sustainability are entwined issues, as data may be unusable without the right software, and vice versa.

Following a trail blazed by the bioinformatics community, humanities researchers have started looking to linked data and knowledge graph technologies to create sustainable interoperable data (e.g., Linked pasts network, Linked art, Mapping manuscript Migrations, etc). But even when using standard data models and formats, divergent or inconsistent use of vocabularies (ontologies) can create barriers to re-use. And maintenance (updating and correction) of substantial datasets can require significant resource.

3. Case study: Music Encoding and Linked Data

3.1 MELD background

Music notation expresses performance instructions in a way commonly understood by musicians, but is limited to encoding static, a priori knowledge. Music Encoding and Linked Data (MELD)https://meld.web.ox.ac.uk/) is a basis for communicating static and dynamic information between musicians, musicologists and others. MELD is essentially a framework for distributed real-time annotation of digitally encoded music notation (including, but not limited to, musical scores). MELD users and software agents create annotations of semantically distinguishable music concepts and relationships. These are associated with musical structure specified by the Music Encoding Initiative (MEI) schema. The use of standard Linked Data RDF and Web Annotations allows incorporation of further knowledge from non-musical sources.

The MELD framework isn't entirely typical of the DH software characterization outlined above. It was conceived from early in its lifetime as a framework that could be used in support of a range of music research applications. The initial development was conducted and funded under the aegis of a large multi-centre multi-year engineering project, which was only later re-targeted for use in DH

projects. Yet it does still suffer from some of the environmental characteristics described: collaboration with other researchers would give rise to requirements that had to be addressed very rapidly without time to consider sustainability issues. And while the MELD framework itself benefited from the substantial efforts of a large research project, the specific applications under review have been created within the more limited resourcing of humanities projects.

The MELD framework has a number of components, two of which are:

1. `meld-clients-core`: support code for MELD client (browser) applications based on the Javascript React framework. This is a powerful, popular and widely used framework, created by Facebook, for implementing Web browser applications. But with great power comes considerable complexity, and becoming conversant with React can take considerable effort, even for someone already familiar with Javascript. Further, React makes heavy use of modern Javascript capabilities and patterns that are a considerable departure from traditional programming models.

2. `meld-web-services`: a set of web services to support MELD client applications. Historically, these were custom code to support sessions and annotations, but much of this is being replaced by off-the-shelf Linked Data Platform (LDP) servers (e.g. GOLD, and more recently using various deployments of Solid.

3.1.1 MELD applications

To give a sense of its versatility as a framework for musical applications, here are some examples of applications that have been built using MELD:

1. Take it to the bridge a platform for multi-performer communication during jam sessions. This MELD application displays the currently played score segment, and allows musicians to signal jumps to different parts of the score.

2. Climb! is a non-linear composition written for Disklavier, in which a pianist undertakes a metaphorical journey up a mountain, playing musical codes that are hidden within the score to control their path and trigger musical and visual effects, including the pianist and instrument playing together in an unusual duet.

3. SOFA Ontological Fragment Assembler (SOFA) enables the combination of musical fragments – Digital Music Objects, or DMOs – into compositions, using semantic annotations to suggest compatible choices. Also known as Numbers into Notes remixer.

4. Lohengrin TimeMachine Digital Companion. Exploring the Forbidden Question: the digital companion to a textual and video essay on motif use in Lohengrin.

5. Delius in performance uses MELD for publishing multimedia musicology articles as web applications in which musically-meaningful relationships are presented as links between text, audio, video, and musical score.

For this SSI3 sustainability project, our attention has been focused on the Lohengrin time machine study and Delius in performance projects.

See also:

- http://www.semanticaudio.ac.uk/
- https://www.qmul.ac.uk/media/news/2018/se/musics-changing-fast-fast-is-changing-music.html
- https://github.com/oerc-music/ForbiddenQuestion
- https://github.com/oerc-music/delius-annotation
- https://dl.acm.org/doi/10.1145/3273024.3273038

3.1.2 Server and client code

Web applications typically have two main parts:

- a ***web client***, in the form of a web browser (e.g. Microsoft Edge, Safari, Chrome, Firefox, etc.) which retrieves information from the web and presents it to a user, and
- a ***web server*** that hosts web site data, and uses it to construct replies in response to requests from web clients.

Client and server components typically exchange information using the HTTP (or HTTPS) protocol, with a client issuing HTTP requests, and a server replying to them with HTTP responses. Sometimes, other protocols are used (e.g. WebSockets, RTP, ftp, and more), but interactions are usually initiated using HTTP(S).

Traditional web applications were arranged with most of the application logic applied by the server, and with the browser used mainly for presentation and user interactions.

Modern web applications often share the application logic between the server and client components: some data processing may occur on the server before it is sent to the client, and further processing may be performed by code that runs in the browser client. The client-side code (usually Javascript) is sent from the server to the client as and when it is needed. Client-side Javascript frameworks (such as React) make it possible to write applications that execute entirely in the browser, and just use the Web as a way of distributing them, along with the data they use, to the client for execution. Many of the MELD applications we have considered are of this kind.

The traditional and modern approaches each have advantages. A key advantage of using client code for application logic is that it removes load from the server,

freeing it up to respond to many more requests from multiple users using the same service. It also allows the user interface to become more responsive, rather than a simple rendering of data that is sent by the server. This can come at the cost of increased application complexity, with code needing to support a range of different Javascript implementations in different browsers, and even between different versions of the same browser.

The MELD framework pushes a lot of application logic to the client (browser), leaving open a possibility that the server side functions can be handled by off-the-shelf server software (e.g. Solid).

In previous work on SOFA, we have also explored the use of middleware "agents" for MELD applications, that retrieve data from a server, apply some processing to it, and write new (additional) data back to a server. But in the end, interaction with a user usually uses a web browser interface.

3.1.3 React and Redux

The MELD client libraries are implemented in the Javascript programming language, and are designed to be used with the popular React and Redux browser application frameworks.

React is a Javascript library for building user interfaces.

Redux describes itself as a predictable state container for Javascript applications.

These powerful frameworks provide a lot of functionality that can be used by MELD applications, but also bring a fair degree of added complexity, which in turn presents challenges for sustainability of MELD applications.

3.2 Approach

The original intent of this sustainability project was to focus on testing the web (HTTP) interfaces between project-specific and generic back-end storage components, using existing work on SOFA and MELD as a starting point. But, delving into the software, we found that much of the essential logic to be tested was exposed through API calls within browser applications. This required a rethink of the approach, which has led to us facing and dealing with several specific software sustainability issues.

We found that much of the MELD functionality in the applications we surveyed for this project, Lohengrin time machine study and Delius annotation, was visible only to internal API calls, so a different testing approach was needed.

3.2.1 Initial plan

The original plan of work was to build upon command line tooling already created for testing middleware in the SOFA project. This tooling was designed with testing in mind, providing a range of command options that could be used to create, access and test resources stored in an LDP or Solid server. The intent was that for components that communicate via the LDP/Solid store, the command line tool could be used to set up test fixtures, and then interrogate the result of running a component. Pass/fail results for individual tests are available via command exit status values.

These commands could then be deployed in shell scripts (as we did for SOFA) to act as a basic framework for unit- and integration test suites, which could subsequently be used to confirm that the applications were behaving as intended, and also that the MELD components were behaving as expected by applications.

We envisaged focusing on a few MELD applications:

- Lohengrin TimeMachine Digital Companion.
- Delius in performance
- TROMPA's CLARA rehearsal companion
- Historical musicology with mixed media digital archives (Text, audio, image, score from British Library, New York Philharmonic Archives)

Our study would use MELD as a reflective case study of software sustainability in DH projects. While the MELD framework itself benefited from the substantial research efforts of FAST, the apps above have been created within the more limited resourcing of humanities projects. This is an opportunity for SSI to reflect on effective software sustainability practice in such situations, which are typical for DH.

Intended outputs of this work were to be:

- The software testing framework for MELD (public repo).
- Unit tests for (some/all of) the above studies/apps (public repo).
- Documentation and a best practice report (public, online).

3.2.2 What we actually did

It was revealed early in the project that HTTP-level testing using a command line tool wasn't going to provide sufficient access to exercise key application logic for the Lohengrin and Delius applications. To test the internal interfaces, we wrote testing code in the applications' implementation language (Javascript), using existing test frameworks. This required a greater familiarity with the implementation of applications using MELD, and the React framework ecosystem on which the MELD client-side code is built. In turn, this required us to more immediately address a number of sustainability issues in the MELD codebase.

The main consequences of this revised approach was a deviation from the original plan, with the creation of a series of simple "Hello MELD" applications, designed with testability in mind. Implementing these applications has yielded several lessons in creating sustainable MELD applications, and also some re-evaluation of the attainable goals of this short project.

Revised goals:

- dropped (for now) the goal of testing at the HTTP interface
- implemented a series of simple "Hello MELD" applications, designed with testability in mind, and used for both learning and testing
- implementing some simple tests using existing Javascript test frameworks (Mocha and Jest were trialled, and settled on Jest as it integrates more easily with React).
- a partial shift from testing as lead activity, to recording and documenting observed sustainability issues with the MELD libraries and applications.

As a result, the outputs of this exercise will focus less on specific implementation of sustainability practices, and more on describing sustainability issues and possible mitigations, than was originally envisaged:

- The "Hello MELD" applications, which can be used as teaching as well as testing resources.
- A software testing framework for MELD based on the "Hello MELD" apps, and existing Javascript testing frameworks.
- Documentation and a best practice report (this report), and further details in Github.

3.2.3 Observations

Complexity of supporting software environment The MELD client support code is implemented using NodeJS, React and Redux. This environment offers a rich set of features, which have allowed complex functionality to be incorporated into MELD with relatively little initial software development effort.

But React is a complex, dynamic environment with many supporting tools. This gives rise to a complex web of dependencies, some of which may be frequently changing. This can give rise to incompatibilities and failures when the MELD software itself, and any supporting tools used, are not kept up-to-date with the latest React/Redux library modules. Many of the updates to underlying software are to address security vulnerabilities, so are important to incorporate into a public-facing web application.

Undetected problems in MELD code There were a number of areas where the MELD support code did not behave as expected. This would cause problems with existing applications, which might still work only by virtue of unrecorded changes made to the runtime environment, but which would fail when installed

in a new environment. When encountered, these failures are often hard to track down and fix.

These are problems that automated tests, and especially continuous integration, are intended to catch. The "Hello MELD" applications proved helpful when it came to isolating such failures, though repairs have proved harder to effect.

Effective testing is a cornerstone of software sustainability.

Retro-fitting tests to existing applications has proved to be challenging, for reasons that range from setting up a test environment that sufficiently mimics the application environment, to accessing internal interfaces that expose application logic to be tested.

Working with higher level abstractions Something that the React/Redux framework makes possible is creating higher level abstractions that hide some of the gnarly details of how the software plumbing works. This is a good thing, but the value of this approach can be difficult to realize if the abstractions used are not adequately described and documented. Without supporting documentation, developers can be left needing to dig down through multiple layers of code abstraction to gain an understanding of how the abstractions are supposed to work. (For example, when working on some MELD code, we spent a fair amount of time chasing callbacks through several layers of code to get a handle on how a moderately simple application was supposed to work.)

Further, if new abstractions are introduced, this increases the importance of testing to confirm that the implementations do properly present the higher level capabilities that they are intended to provide. When there are bugs in implementations of higher level abstractions, the effort required to isolate and fix them is much greater, especially when they are encountered at a later date when the original developer may no longer be involved with the project. Test cases can provide an executable and automatically testable specification of how an abstraction is expected to behave.

Callbacks and Promises This is a narrow technical issue, but also illustrates issues of broader system architecture choices concerning local vs global simplification. A key contributor to sustainable software is keeping it easy to understand (for new developers), which among other things means keeping its structure as simple as possible (but no simpler). This can be a delicate balancing act, and inappropriate choices either way can lead to greater code complexity, and attendant sustainability difficulties.

In traditional programming, when using a result that becomes available at some time after it is requested (such as reading data from an external source), it is common for the calling program to simply wait for the result to be available. This is synchronous operation. But in modern web software it is often not desirable or possible to do this, and the results must be handled asynchronously

(i.e. some time later, without pausing the application at the point of a request). An approach that is often used is to have the invoking software provide a function (a "callback") that is called when the requested operation has completed.

The problem with callbacks is that both the caller, the invoked function, and all intermediate functions need to incorporate the callback into their interface, which can leads to added complications in code paths that are unrelated to the operation being invoked. In this way, callbacks can add arbitrary complexity that permeates a codebase: changing one function may also require changes to a lot of unrelated code. Thus, while conceptually quite simple, the ramifications of using callbacks are very extensive. As a piece of software becomes more complex, managing and synchronizing asynchronous operations with callbacks becomes more difficult.

An alternative to callbacks is to use Promises, which are widely used in Javascript. A Promise (or Future) is a new abstraction that is used to manage synchronization of asynchronous operations. Compared with callbacks, Promises are conceptually more difficult to understand. But disciplined use of Promises can facilitate creation of easier-to-understand global structures in non-trivial software systems, by hiding the asynchronous aspects of operations from intervening code.

This use of callbacks vs Promises illustrates a broader issue of local *vs* global simplification. Compared with callbacks, Promises introduce a new layer of abstraction that may make some code harder to understand in isolation, yet often make the overall global structure of the code easier to follow. Many programming abstractions come with similar pros and cons, and it is not always easy to tell if using one will make the overall application more or less sustainable.

Where such abstractions are introduced, it is probably better, where possible, to use ones that are already widely known and understood (like Promises). There are a number of books containing catalogues of software design patterns that might help mto make such choices. One of the better known of these is Design Patterns by Gamma, Helm, Johnson and Vlissides, sometimes also known as the "Gang of Four".

Browser vs non-browser code In theory, Javascript code is highly portable, possible to run in a browser, desktop or server computer. This is very appealing for testing, as it can be tricky to run automated tests in a browser, and continuous integration platforms are generally server-based. (There do exist mechanisms for testing in "headless" browsers, but these can add complexity for a small project)

In practice, there are a number of significant differences: Javascript language variants, run-time support, module loading mechanisms, and more. We ran into several cases of code that would run as expected in a browser, but would fail when run in a non-browser test environment, due to differences in the ways in which modules were loaded.

The original intent to test the HTTP interface was dependent on a significant

amount of application logic being implemented on a server. When this is not not the case, effective testing requires that it uses code-level APIs, with test code directly invoking APIs in a way that that resembles application code.

Setting up even a very simple test environment for testing browser-based application code turned out to be challenging. Theoretically, the testing framework would paper over any differences from the browser environment. In practice, there were areas where differences would persist, and complex workarounds were adopted to avoid provoking these.

Hello MELD application series The "Hello MELD" applications were extremely helpful, both for learning the framework when bringing a new developer on board, as it allowed the various parts of a complex set of interfaces to be mastered separately, and creating code that was easier to test. Also, the "Hello MELD" series of applications, which performed increasingly complex operations using MELD, provided fallback options when failures were encountered, allowing code-induced and environment-induced problems to be identified and separated.

It was important that the first of these "Hello MELD" applications was so simple that one might easily think that there was nothing to go wrong. In practice, things did go wrong, and even a trivially simple application served the important function of proving that a minimum set of runtime requirements had been set up correctly. Having a trivial application provided a useful platform for exploring testing tools and frameworks without getting bogged down in application code minutiae, and allowed easy experimentation with application design options (using established MELD patterns) that would facilitate testing.

Value of part-time technical expertise The availability of occasional access to an experienced software developer was reported to be very valuable for project researchers who were primarily humanists, but engaged in software development. While it can represent a significant cost to a project to employ a full-time developer, having part-time availability potentially offers many of the benefits of a full-time developer, and potentially facilitates training and technical capacity building for digital humanities projects.

For MELD, the funded developer was at 20% FTE (1 day/week) for the duration of our study, with the effort split between (a) software maintenance and building testing capabilities (i.e. direct technical work), (b) supporting other project developers (indirect technical work) and (c) project management, SSI3 introspection and and reporting activities. One day/week is relatively easy to schedule and manage, but the above suggests that 10%/0.5 day/week might be sufficient for technical support aspects.

An area we noticed that might particularly benefit from a little technical expertise is in early review of software architecture choices. We observed some areas where code maintenance was complicated by some initial choices of software structure

that resulted in repetition of logic that an alternative structure would allow to appear just once. (Such repetition makes it more likely that subsequent software changes will introduce inconsistencies and non-obvious problems). Such architecture review could occur at the start of software development, but it be more effective if performed when a very early prototype has been created, and the method of implementation has been clarified and tested. (In the observed example, some 6 different types of "annotation" has been implemented, each with different data model and user interaction requirements. If just one or two annotation types had been implemented at the point of review, it would have been easier to suggest a software structure that would facilitate adding new annotation types with minimal changes to existing code.)

Distributed collaborative working Work on MELD software involves researchers from multiple institutions in UK, Germany, Austria, Spain, and more. Software development by a widely dispersed team is challenged by disparate goals of different groups and conflicting demands on their time, complicating the process of scheduling meetings to work through issues. On the positive side, 2020 (the year of the Covid-19 pandemic) has seen a rise of tools to facilitate remote working, notably much improved videoconferencing systems. Yet we have also seen that effective participation in remote meetings is much harder work, and more draining, than face-to-face meeting. The MELD team has long used Slack instant messaging as a medium for ad-hoc coordination of activities and discussion of technical issues. This is not new, and continued to be an important channel for communication.

For the MELD sustainability work, we scheduled a week of remote pair programming work as a way to build MELD technical capacity for the Beethoven in the House project. We found that Zoom screen sharing was an effective technique for this (other video conference systems might work as well or better; we found the affordances provided by Zoom seemed about right for this work). There is a retrospective review of this activity with more details.

More broadly, for supporting *Beethoven in the House*, the pattern we followed has been a series of short online development sessions building and understanding simple early incarnations of "Hello MELD", followed by the longer sprint to build a more complex application. This pattern seems to have been very effective, and reinforced value of simple early "Hello MELD" apps. These focused development sessions did prove to be an effective way of making technical progress and transferring knowledge - an important aspect of these sessions was actually doing the work, to see all the details worked out. Yet these online pair programming sessions are hard work, and we found it most effective to have frequent shorter online sessions (an hour or so) with longer breaks between for participants to work separately and consolidate their knowledge. This observation was offered by Mark Saccomano, who participated in this process:

> I think any project needs to make judicious use of teleconferencing

> software, and be realistic about expectations. I am finding meetings larger than 2 or 3 people to actually be counterproductive. Chat in Slack, though, has been great since we began! It is easier to take in information and you can always scroll back and refer to it when needed.

Yet, talking to other project participants, it was noted that Slack (or similar) should not be seen as a replacement for other project planning and reporting documents. Rather it should be seen as an alternative to a physical office presence, with recognition that Slack conversations are not a substitute for planning and scheduling activities.

It was also noticeable that a the problems of dealing with code instability, noted elsewhere, are exacerbated by the challenges of remote working. With limited contact time, it is easier to discuss an issue with reference points (e.g., a stable codebase and working tests) as a baseline. These can help to focus attention on where problems have been introduced. The "Hello MELD" series provided useful anchor points for our work, albeit somewhat hampered by failure to create a meaningful test suite.

4. Sustainability issues encountered

This section aims to highlight key issues encountered. Raw observations and notes are contained in a separate document: MELD sustainability issues observed.

4.1 Working with a complex and dynamic software ecosystem

We ran into a number of compatibility problems while working with MELD on recent versions of NodeJS, React and Redux. Attempting to update dependencies to later versions gave rise to failures that proved quite difficult to identify and remedy, and required changes to the MELD software. Further, MELD itself is undergoing several changes in response to new application requirements, raising further library compatibility concerns.

Keeping application and support code up-to-date with a complex evolving environment like React/Redux requires significant ongoing engineering effort, which can be challenging for a small research project. This is especially true if the development is split over several research projects that don't necessarily have continuously available development effort - the wider world doesn't wait, and fragmented research activities are easily left behind.

Suggested mitigations

- before adopting a complex support platform, consider whether the value provided (in terms of easy-to-use features) justifies the additional complexity and maintenance requirements. (The consensus for MELD is that the features offered by the React/Redux platform did enable results that otherwise would probably not have been achieved.)
- when adopting a complex support platform, consider a strategy or practice to ensure that the underlying dependencies can be safely updated. Ideally, this will involve having some form of automated testing and continuous integration in place. Try to keep dependencies up-to-date as you go, rather than relying on infrequent updates of multiple dependencies.
- in selecting dependencies, consider whether they are actively maintained. We encountered some problems with dependencies that were not maintained, and as a result were not updated to use newer versions of the underlying React platform, leading to multiple incompatible versions. The exhortation to keep dependencies up-to-date applies also to indirect dependencies (which depends on other developers).
- consider also that the code you create may become a dependency for some other "downstream" application. Sustainability of any such downstream application can be improved if your code has frequent releases that are up-to-date with respect to changes in its own dependencies. This allows any downstream application to use the latest versions of dependencies, mitigating one aspect of the so-called "dependency hell" problem where multiple (and potentially incompatible) versions of the same dependency are included. (Note that this primarily facilitates new application developments, and does not necessarily force older applications to be upgraded to use newer versions of the dependencies.)
- consider how to create a complete snapshot, including all supporting components, of a working version of an application. For example, as a Docker image? (Bear in mind that for a web based application, there remains a possible dependency on the browser used.)

4.2 Inconsistent build and runtime environments

We found different developers were using different versions of the runtime environment and build tools. This led to applications that worked in some environments and failed in others.

Some of the build tools used were not always reliable. For example, Node Package Manager (npm) is supposed to record package versions used, and thus ensure consistency. In practice, we found that differences between different development environments were not being corrected, leading to inconsistencies noted.

Clear identification of supporting build tools used was lacking. The Javascript/React/Redux environment in particular relies on a range of tools to compile code to a form that can be run in a base runtime environment. The tools needed are themselves somewhat dependent on the NodeJS and/or web browser versions used to run the application code.

Suggested mitigations

- create automated scripts to create clean runtime and development environments from scratch. Such scripts also serve usefully as documentation. Creating a Docker container for an application is one way to achieve this, and may also provide a pre-built ready-to-go environment that others can use.
- establish a continuous integration environment: this forces the environment to be rebuilt from scratch (or from established and documented resources) whenever the software is updated.
- maintain development/test environments separately from other aspects of the host system (e.g., using tools like Node's `nvm` or Python's `venv`). Provide detailed instructions and/or automated script for setting up a new development/test environment, or resetting an existing one.

4.3 Lack of automated testing and continuous integration

Although MELD has a simple test application, there was no automated test suite or continuous integration set up. (Continuous integration is used here to mean an environment that automatically installs the software into a clean environment, runs the tests and reports any failures, whenever any changes are made to the software. This allows problems to be detected rapidly, hence easier to fix because the breaking changes made are still fresh in a developer's mind.)

The lack has resulted in uncertainty about the status of the software. Attempts to install the simple test applications resulted in failures that would probably have been picked up by automated tests and continuous integration.

Suggested mitigations

- establish an automated testing regime at the outset of a project, or very early in its lifetime. Much of the value of automated testing can be realized even by very lightweight tests - 90%+ test coverage is great of you can afford it, but even very minimal coverage can provide valuable proof of life, and can be quick to implement. In practice, once a testing regime is in place, I find it tends to collect coverage-improving tests over time with

very little additional effort. Expect automated testing to pay back any initial investment of effort within a few weeks of development activity.

- if it can be done quickly and easily, set up continuous integration early in the project. A prerequisite for this is some form of automated testing, even if it only to ensure that system can be built and installed. In practice, I've have found that having easy-to-run tests can be almost as valuable (but may be less effective for picking up installation and environment setup problems). It should be very little effort to set up continuous integration that does no more than install the dependencies and build the package, which can help to catch a lot of problems, and provides a place where further tests can be added as the project evolves.
- consider testability when designing code: ensure that important features are easy to invoke and access for testing. For example, separating data manipulation functions from I/O and user interface allows them to be tested in isolation with fewer execution environment dependencies.
- treat tests as an important source of documentation: some of the effort put into documenting an interface may be better directed toward creating tests that can also be used as examples.

Test fixtures and mocking

For data-intensive applications, including many DH applications, consideration should also be given to creation of text fixtures (e.g., datasets against which the tests are run), or "mocking" of the interfaces through which such data is accessed. This aspect has not been explored as part of the project reported here.

See also the section Data preparation below.

4.4 Application code version management issues

The MELD libraries are being used by different developers in different institutions, who also apply fixes and updates to the libraries themselves. This has meant that different branches of the MELD code being used in different applications over extended periods of time, exacerbating the consistency issues noted previously.

There was some early lack of clarity about the code governance (e.g. the exact requirements for updating the reference version of the software that should be the basis of all new applications (and preferably any that are actively being developed).

Suggested mitigations

- use a code version management system, such as git. This shouldn't need saying.
- establish a governance structure early in the project. It doesn't need to be complex: one of it's main purposes is to let all developers know what is expected in order to update the main codebase. An appropriate level of requirements imposed by project governance should take account of project resourcing constraints. But also remember that, in the long term, sensible quality requirements save more effort than they cost.
- try to make use of available affordances to automate governance constraints; e.g. using GitHub reviewer requirements for merging a pull request.
- don't make "kitchen-sink" updates to the code base. Tackle changes in small chunks, and include a clear statement of purpose with committed changes.

4.5 Application code complexity

The MELD support libraries, and especially the underlying React/Redux framework, present a rich and flexible interface, which in turn forces a degree of complexity onto the application code that uses them. This has meant that applications are initially difficult for new developers to understand. We found that, in practice, it took a significant amount of hand-holding from an existing MELD application developer to bring a new developer up to speed using the MELD framework.

Suggested mitigations

There's no silver bullet for solving this - it's a perennial software problem. But some things to consider might be:

- create some very simple, easy-to-understand applications/instances with minimal functionality. These can provide a gentle onramp, helping developers to understand which aspects of a platform or system are providing what kind of functionality (and which aspects might be ignored).
- try to conceive an application as independent modules with separate concerns. An example of this approach is Unix command line tools that can be used alone, or combined in various ways. It's not always easy or possible for web applications, but we had some success with SOFA (an earlier MELD application) using Solid (LDP) for data storage, and chainable command line tools for various features.

- use abstractions judiciously. Sometimes, a well chosen abstraction (or encapsulation of some common pattern of operation) can lead to simpler code. But abstractions bring their own additional layer of knowledge required to understand, so should be selected with care. (See also Working with higher level abstractions and Callbacks and Promises.)
- don't let too much technical debt accumulate (for some value of "too much"). (See also: 4.8 Accumulated technical debt.)

4.6 Run time error detection and reporting

Run time error detection and reporting in the MELD libraries, and sometimes in the underlying React/Redux libraries, was very patchy. This would result in applications failing for no apparent reason, or failing for indicated reasons that have no discernible connection with what the application is trying to do. This can make problem diagnosis and rectification difficult and time-consuming.

Suggested mitigations

Mitigations here are common software coding practices.

- don't fail silently. If something is wrong, or if an option is not (yet) implemented, report it. Such reports don't have to be pretty, just obvious. If possible, they should refer to application data and concepts, not internal software concepts.
- code defensively! Don't assume that certain inputs can't happen: check them and generate a report (or crash with a stack dump) if they aren't satisfied.
- provide debug options to report inputs received, to help identify where any problem might be occurring.
- consider using an application monitoring tool such as Sentry, which allows an application to submit a diagnostic report when an error is encountered while running somewhere else - these reports can be useful for locating hard-to-replicate errors, or when users are not equipped or motivated to provide bug reports. (There is some discussion of front-end application monitoring and alternatives to Sentry on this Geekflare page.)

4.7 Data preparation

MELD applications, in common with many Digital Humanities applications, typically work with existing data, which may be created using other tools (e.g. music files), or extracted from existing public resources (e.g. Wikidata). For an application to be useful, or testable, suitable input data must be available.

The MELD framework does provide data for testing and demonstration use, but it wasn't always easy to find.

For creating a simple stripped-down application, we needed to create a new dataset using tools that are not part of the MELD framework. The MELD documentation did not contain pointers to easy-to-use options for creating such data, and this process alone took a few days of investigation and experimentation.

Suggested mitigations

- for testing, provide ready-to-use datasets. Ideally. they should be as simple as possible for the feature they are intended to test.
- for users, provide simple instructions for creating input data. For example, in the case of MELD, an MEI file can be created from a MuseScore musical score, exported as MusicXML and then processed using the Verovio web service to convert it into MEI (e.g. as described for `hello-meld-2`)
- provide pointers to file format and vocabulary term descriptions.

4.8 Accumulated technical debt

This isn't really a separate issue, but more a consequence of the other factors reported above.

Technical debt refers here to software additions and patches that are used to get some software running, but which don't represent an optimal way of solving the underlying problem. It often results in code that has duplicated and/or tortured logic, and fails to maintain a clean separation of concerns between software elements.

When developing software under time-and-resource constraints, it is often expedient to take short-cuts to validate an approach, or to demonstrate a result. Individually, these shortcuts are often not problematic, but if they are allowed to accumulate in a codebase the resulting software can quickly become harder to understand and update. And faced with with such difficulty, a developer will often use additional shortcuts or workarounds to avoid having to deal with the difficult code. In this way, the accumulation of technical debt can grow exponentially, and the speed of making changes and fixes will decay accordingly.

Further, for MELD, technical debt was accrued over a number of smaller side projects (MELD applications), each with their own priorities for what MELD should provide, but for which long term maintenance of MELD itself was not a central concern (apart from the fact that they would ultimately depend on sustainability of MELD). Such debt accumulates when each side-project makes its own separate enhancements to the MELD code base, resulting (among other things) in code management issues noted previously.

Some technical debt recovery was performed as part of this sustainability work, and it was noticed by other project members that there is value in paying down technical debt separately from immediate pressures of application development.

Suggested mitigations

Again, no magic bullets here. It's mostly common software engineering practice and skills.

- be aware of technical debt; keep notes of where it exists, and ideas for eliminating it. When a change turns out to be more difficult than it should be, that may be an indication that the code first should be refined.
- don't let too much technical debt accumulate.

  How much is too much? This is hard to say - it's also possible to spend too much time refining code. A possible rule of thumb is this: when a piece of code needs to be modified, first see if there are any changes that make the desired changes easier to apply while preserving its existing functionality.
- try to maintain separation of concerns between software components.
- try to avoid duplicated logic (also known as DRY: "don't repeat yourself").
- don't combine changes that address technical debt with changes that update functionality. No existing test should fail as a result of refinements that pay down technical debt.

5. Lessons in sustainability for DH

Achieving long-lived utility for the software outputs of a digital humanities research activity (or any research activity?) is often a delicate balancing act between getting the work done, and putting in the extra effort to ensure longer term viability (i.e. so that the work done "stays done").

This balancing act affects resource allocation, conduct of the project, and technical design choices, which are discussed below under the headings "Resourcing", "Technical activity" and "Technical design".

The discussion that follows is not unique to DH software, but it is intended to focus on aspects that arise more commonly there.

5.1 Resourcing

Considerations of resourcing will necessarily involve trade-offs between immediate requirements and longer term utility and sustainability. Such considerations are

not unique to DH software: in "The Mythical Man-Month" chapter "The Tar Pit", Frederick Brooks discusses near order-of-magnitude costs associated with the evolution of a "program" (solving an immediate problem) into a "programming systems product" (providing long-term utility, and supporting evolution to address new problems).

More recently, such trade-offs are considered by "agile development" practices, which propose that it is futile to expend up-front effort to address future problems, but which also propose that evolution is a fundamental element of software development, and needs to be supported by development practices (such as continuous testing, refactoring, etc.)

5.2 Technical activity planning

When planning (and budgeting) a development activity, some factors to consider are:

- code management
- dealing with technical debt
- combating software decay ("bitrot")
- code testing
- automated builds

This is not an exhaustive list, but reflects some key concerns that have been raised by our work with MELD.

Code management probably needs little discussion, as it seems to be a given for just about any software project. The existence of a shared repository of source code, which includes a history of all changes made, is very common practice and, among other things, helps to safeguard all the efforts that go into creating some piece of code. A popular code management tool is `git`, which is supported by services such as GitHub and SourceForge. There are plenty of challenges associated with source code management and version control, but these have not proved to be a major issue with MELD.

Technical debt arises when the overall structure of some software is misaligned with the functionality it is required to provide. It accumulates through the application of quick (or not-so-quick) fixes to add new functionality, without adjusting the software structure to accommodate the changes. Addressing technical debt requires a level of ongoing effort that does not, of itself, advance the capabilities of the software. Not attending to technical debt typically results in software that becomes harder to maintain or modify with the result that new features become more difficult and more expensive to add. But paying too much attention can result in wasted effort.

Where technical debt arises (mainly) from changes to the software itself, decay may arise from changes to the external environment in which it is deployed and/or developed, including operating system, language compilers, runtime libraries,

external libraries and utilities used, etc. In particular, software may decay even when it is not being changed; security mitigations are a particular source of such changes. Combating decay requires a degree of ongoing engineering effort to recompile and retest software with updated tools, libraries and operating environments, and to apply any software changes that may be needed to ensure its continued operation.

Testing is required in all stages of technical activity. While manual testing may be deemed sufficient when initially developing some software, automated testing is generally a cornerstone of activities to combat technical debt and decay. There are many forms of automated testing discussed in software engineering literature, but for the purposes of discussion here it is probably safe to say that *any* automated testing is better than none at all.

To be confident of being able to successfully install and use a software system, an automated build and deployment system is extremely valuable. Many modern programming languages have relatively easy-to-use automatic build systems (e.g. NodeJS has `npm`, Python has `pip` and `setuptools`, Ruby has `gem`, etc.). Or one might write a shell script to perform the required installation steps. A key benefit of an automated install system is that, when combined with automated testing, it allows continuous integration, where tests are run every time a code change is submitted to a code repository.

5.3 Technical design choices

Sustainability of a codebase is affected by software architecture, and myriad technical design choices. There is extensive technical literature on this subject, which is not covered here. Just a few points for consideration are offered.

When considering creation of sustainable software, there is a risk of creating inflexible monoliths. Most software systems tend to ossify over time, becoming harder to adapt and evolve as early design choices become more pervasive though the system implementation.

Large monolithic applications are often harder to maintain, and hence harder to sustain. Does an application need to be monolithic? Sometimes, especially where non-technical users are involved, the answer may be "yes". The alternatives here would be to conceive an application as consisting of a number of smaller, independent pieces that are run separately, and pass information via files or data handling pipelines. But this can place a greater burden on a non-technical user learning to use the application.

A monolith can sometimes be decomposed by separating data processing components from data presentation. This can also facilitate testing, by creating explicit interfaces that can be used as easy test targets.

In choosing to use a complex dynamic supporting framework (such as React, or any of numerous other web application frameworks), it is possible to incur

dependencies that render the resulting software more prone to decay, and in needing additional effort for sustainability.

Complex user interfaces require a lot of effort to build and maintain, and can be a source of sustainability problems. Sometimes, sophisticated visualizations and flexible interactions are important for research software. It may alternatively be the case that the important research value of the software can be achieved by simplified data presentation, or generating data in a form that can be consumed and presented by an off-the-shelf application.

6. Recommendations for DH software sustainability

The following commentary is not intended to be prescriptive or definitive. It is based on qualitative observations rather than detailed quantitative evidence, and as such offers topics to consider for improving sustainability rather than detailed guidelines. Generally, all projects are different, and each will need to make its own trade-offs, but consideration of some common themes may help to improve outcomes.

6.1 Recommendation 1: Technical architecture

Can the research software be effective if conceived as a set of independent tools that are designed to work together, possibly sharing information via well defined interfaces (such as simple files or data pipelines)? Smaller, independent programs are usually easier to write, test and maintain. Consider separating data presentation from data processing and/or retrieval. As long as the interfaces remain stable, changes to one part of an application suite should not require changes to another part. The downside may be that multiple independent applications are harder for non-technical users to learn and use.

Avoid creating complex user interfaces, or limit the complexity of any that are needed. Implementing a sophisticated user interface can require development effort disproportionate to the value provided, and subsequent maintenance effort is similarly magnified.

In choosing to use a supporting application support platform, consider whether the value provided by the platform outweighs the additional dependencies that are introduced, and that may require ongoing maintenance effort to keep up to date. Note that such platforms can take significant effort to learn, which may lead to difficulties finding future developers to help with software maintenance.

When writing or modifying code for execution in a Javascript environment, consider using Typescript. Typescript is an enhanced form Javascript, which allows some checks for code correctness to be performed to catch errors before

the code is executed, and providing diagnostic information that makes it easier to pinpoint and correct such errors. A valid Javascript program remains valid and performs the same function when processed as Typescript; this means that code changes that support the checking can be applied incrementally, without requiring a "flag day" change to the entire application codebase. To use Typescript, ensure that the tool chain used can support automatic processing of Typescript code.

See also:

- Complexity of supporting software environment
- Working with higher level abstractions
- Callbacks and Promises.)
- Browser vs non-browser code
- Working with a complex and dynamic software ecosystem
- Application code complexity
- Accumulated technical debt
- Technical design choices
- Incremental development

6.2 Recommendation 2: Design for testing

When designing and planning a system implementation, think about how it can be tested. Think about intermediate results that can provide insight into how the software is working, and how they can be made visible for testing.

As the software is developed, implement some tests, however minimal. These will serve to confirm that the software is (to some degree) testable, and also a simple "I'm alive" kind of test is extremely valuable. Later, as problems are encountered, additional tests can be added to explore and diagnose the problem, and help to avoid future regression.

Our experience of working with React and Redux also suggests that using a complex supporting framework increases the importance of having some kind of basic testing framework in place. This helps to document and verify the way that application code is expected to interact with the framework, and will help to confirm that the framework-based code is actually testable.

Consider budgeting (say) 20% development effort for implementing automatic tests. This number is arbitrary, without supporting evidence, so a different number might be chosen. But consider this: if a piece of software is developed over a week, does spending one day of that week writing some basic tests significantly undermine what can be achieved? Even if that effort is not recouped within the week of initial development, I estimate from personal experience that the time would be recouped over a further 2-3 weeks of continued development. Such tests can serve several purposes: thinking more clearly about how the software should work; confidence that the code is (at least partially) working as expected; examples for new developers to learn about the software; executable

specification of how the software is intended to work. The effort saved in manual testing, documentation and learning can quickly outweigh the effort of writing the tests in the first place. Sometimes, writing tests first can even speed up the process of writing the code, in part by forcing clarity about what it is that the code needs to do.

Software engineering literature discusses the desirability of high levels of test coverage. Yet even lower levels of test coverage can be very valuable, and with a test framework in place, additional tests are relatively easy to add as problems are uncovered.

See also:

- Undetected problems in MELD code
- Inconsistent build and runtime environments
- Lack of automated testing and continuous integration
- Run time error detection and reporting
- Data preparation
- Technical activity planning

6.3 Recommendation 3: Continuous integration

With automated testing in place, even if very cursory, and assuming that software building and deployment is automated, it should be relatively easy to set up a continuous integration system so that errors can be detected very when code changes are applied.

See also:

- Undetected problems in MELD code
- Inconsistent build and runtime environments
- Lack of automated testing and continuous integration
- Technical activity planning

6.4 Recommendation 4: Incremental development

Develop in small increments, with testing at each stage. When a failure occurs after a small number of changes have been made, it is easier to pinpoint the cause of failure compared with when many changes have been made to add multiple new features.

At each stage, keep a copy of the working code (typically annotated in a version control system) along with instructions for running it (where the instructions can be as simple as identifying a script to use).

When adding a new feature, consider first refactoring the code: restructuring the code to accommodate the planned changes, without actually changing its

functionality. All existing tests should continue to pass.

Ideally, any set of code changes should be a sufficiently small advance on an existing known working system that they can be thrown away if they don't work. (This isn't always easy to do in practice, but something to think about.)

See also:

- Undetected problems in MELD code
- Lack of automated testing and continuous integration
- Application code version management issues
- Accumulated technical debt
- Technical activity planning

6.5 Recommendation 5: Dealing with technical debt

Don't allow too much technical debt to accumulate.

This begs a question: how much is too much? A rule of thumb might be to eliminate technical debt when it becomes an impediment to adding a new feature. It's usually better to eliminate problem code before changing it's functionality, rather than trying to do so after it has been further modified.

Treat elimination of technical debt separately from adding new functionality, and save changes so that any functional enhancements are made from a recoverable baseline.

See also:

- Accumulated technical debt
- Technical activity planning

6.6 Recommendation 6: Keep dependencies up to date

Don't put off updating dependencies. As with incremental development, it's easier to troubleshoot any problems if relatively few things have changed.

See also:

- Complexity of supporting software environment
- Undetected problems in MELD code
- Inconsistent build and runtime environments
- Lack of automated testing and continuous integration
- Application code version management issues
- Accumulated technical debt
- Technical activity planning

6.7 Recommendation 7: Minimal application examples

The value of minimal applications for introducing computing concepts has been recognized since Kernighan and Ritchie presented "Hello world" in The C Programming Language.

For any system that has aspirations to be community-sustained, such materials can be invaluable. So include the simplest possible examples of how to use a piece of software, and make sure that they work when the software is installed as described. Then build upon these to introduce more advanced concepts. Minimal applications can also provide a useful platform for testing, as they help to keep important concepts isolated, or at least to ensure that common foundations are tested in isolation from more complex interactions that depend upon them.

No application is too trivial to be worth presenting: even if it appears pointless, there may be tacit knowledge exposed that otherwise gets glossed over, or buried by other details.

See also:

- Complexity of supporting software environment
- Hello MELD series
- Application code complexity
- Data preparation

6.8 Recommendation 8: Give some thought to project governance

Project governance shouldn't be a drag - it helps project members to understand how they are expected to be working together. It can also help to establish a basic level of quality (e.g. requirements for code review, tests, etc.) that is expected for any new code added.

See also:

- Application code version management issues
- Accumulated technical debt
- Technical activity planning

7. Proposed further investigations

In this section, we suggest a number of additional sustainability issues that could usefully be investigated through the MELD use-case.

The current short project has succeeded in exposing a number of sustainability issues with MELD, which this document has attempted to explain. We have

also described how many of these lessons are more widely applicable.

Yet there are many questions and issues about MELD sustainability that have been raised, but which there has been no time to investigate. Some of these questions and issues include:

- Further development of MELD testing. MELD is currently undergoing a version freeze, and this would be a great opportunity to "fix" the version functionality upon which applications depend in a test suite. This would allow any changes to MELD that break this functionality to be detected quickly, and steps taken to ensure that dependent applications continue to work: the sustainability advantages here are many-fold: sustaining MELD itself, and also the diverse applications that depend upon it.
- Example test fixtures and "mocks". These are aspects of testing that allow tests to be run independently of a complex deployment setup. This in turn makes the tests easier to run (hence more likely to be used), and also facilitates continuous integration
- Continuous Integration (CI) setup. The benefits of CI have been discussed above, but the present project has not been able to explore setting up a CI environment.
- Performance evaluation and tuning. It has been noted that there are features of MELD whose performance is not as good as would be expected, but we have lacked tooling to isolate the inefficiencies. A test suite is a great place to add in performance measurement options, and the results of this should greatly assist identification of performance bottlenecks. Further, having performance instrumentation embodied in a test suite helps to evaluate any "improvements" made to the code - both to confirm that functionality is not impaired, and also to provide qualitative and repeatable information about the extent of any performance improvements achieved.
- Solid-based application examples and tests. The Solid project, led by Tim Berners-Lee and other web luminaries, is a layer for web application services that separates data custody from application logic. These affordances of Solid play well to the goals of many DH applications where data custody is distributed among many independent players, as information provided and maintained by diverse sources is combined for the needs of some particular research. Efforts are under way for common themes, such as people, places, artifacts, historical eras, etc., to be served by public hubs, but the nature of DH data means that these can rarely be complete for the requirements of any particular line of research.

  MELD is already being developed to use Solid servers for data storage - for both source data and intermediate results ("annotations"). But further exploration and testing is needed to better understand how Solid security models can best be used by DH research software.

- Documentation of abstractions used. One of the problems we observed was the use of application-specific abstractions that were not adequately described. An exercise to identify and document the various abstractions introduced by MELD to simplify creation of music and musicology applications, and how they may be composed, would be useful in its own right for MELD developers, and may lead to identification of design patterns that are more widely applicable in DH applications.

One lesson of the work reported here has been the value of a project having occasional access to a software engineer who has a good understanding of the goals and nature of the DH project, and experience with the kinds of issues faces by DH researchers. Such occasional access can be very cost effective, yet is not always easy to arrange, and once in place can be difficult to maintain without some continuity of resource provision. The MELD project is currently better-placed than many DH projects in this regard.

8. Acknowledgements

This work was funded by the Software Sustainability Institute (SSI), SSI phase 3 .

It was performed with collaboration and feedback from the following MELD-related projects:

- MELD framework, part of the EPSRC-funded FAST project.
- Unlocking Musicology
- Companion for Long-term Analyses of Rehearsal Attempts (CLARA).
- Digital Delius

Thanks to participants of the online MELD mini-fest meeting on 8 February 2021 for comments and feedback on this work: Kevin Page, David Lewis, David Weigl, Mark Saccomano, Alastair Porter, Laurent Pugin, Stefan Münnich, David John Baker, Cynthia Liem. Alastair Porter and Mark Saccomano provided particularly helpful review and commentary on an earlier version of this report.