# Example Design Decisions

***Report Writer (Civicore 2010)***

1. Rather than implementing XML reading from scratch, SimpleXML was used to do the base parsing and writing for both the report data and for the configuration file. Since our use of XML was limited and simple, it made sense to use a library rather than to write our own code to handle XML.
2. We used PHP because it was recommended by CiviCore who are in the process of moving much of their infrastructure to PHP. Additionally, it was much easier to learn and use rather than having to learn how to use tomcat for java.
3. User files are not immediately placed on the server, but are instead stored temporarily as binary. This allows for consistent handling of images from both the configuration file and from user upload.
4. User files created for the report are stored in temporary folders on the server. Unfortunately, they are needed until the last page of our current implementation, so they can not be deleted automatically without interfering with the report display. To facilitate the deletion of these files, a simple bash script is included.

***WiiGames (CSM3 2010)***

Apart from external libraries, we had a few design decisions that need explanation. First, we decided with the client that communication between all parts of the system should be done with network sockets. This made it very easy to develop and test each piece in parallel. For example, instead of needing to build from the ground up – from reading raw data towards the game event parser – we were able to develop each piece in parallel by sending "fake" input over socket. Also, the client has plans for future network-based games. Since sockets are a fundamental part of the framework, developing network games will relatively simple. To ease our development process, we chose to use the connection-less UDP protocol instead of the more reliable TCP. This allowed us to, for example, test the Wii Event Parser without needing to have 18an instance of the Game Event Parser running and listening for connections. Also, the relative importance of individual packets is small, so the framework can easily tolerate a few lost packets. However, future developers may choose to change communication to TCP.

Another design decision needing justification is our use of processes instead of threads for each component of the framework. It might seem natural to have the game process start the Wiimote Manager, Wii Event Parser, Game Event Parser, and Game Event Listener in separate threads. When the game is over, the process would simply kill all the threads. This is the direction we took initially, but we quickly found that threads are very difficult to use properly. We spent countless hours tracking down errors resulting from threads before the client advised us to de-thread and use processes instead. In retrospect, it was foolish to use threads in the first place. Almost all communication between the components happens over sockets – as opposed to directly sharing memory – so threads are superfluous.

Running each component as a separate process greatly increased the flexibility of our games. The games can function as a stand-alone application, starting and stopping the subprocesses when needed, but the processes can also be run separately. For example, while developing, we would often want to run the game several times without the hassle of re-syncing Wiimotes every time. Since the Wiimote Manager runs in its own process, we were able to sync the Wiimotes once, leave the process running, and continue testing. This flexibility will also be very useful for the client when he is teaching with the games, because he will not have to waste valuable class time re-syncing Wiimotes.

***Website/Mobile design (ModsDesigns 2010)***

*Privacy*

For the current version of the website, no sensitive customer data such as credit card or PayPal information is being stored and therefore cannot be compromised. The only other semi-private information stored in the system are users' contacts. These are stored in a password-secured database that cannot be accessed remotely. The website will only display contacts for loggedin users and only that user's own contacts.

Like any service, it has the potential to be abused. There are mechanisms in place to stop all delivery of picture messages to blocked numbers. However this feature has not been activated for this version.

*Security*

We have opted to only store a salted-hash of the users' password. This minimizes the risk to users even if the entire database is stolen. Although security breaches are of low risk, the security system would be enhanced by including captchas to prevent automated programs from using the website, rate limits (for the same reason), and automatic watermarks to protect the intellectual property of the client.

***Gesture recognition (TR Services, 2010)***

Our project required intensive research, and we had to abandon many paths that didn't appear to be close to being practical. These paths include basic object tracking through segmentation, shadow removal, k-NN AI trainers, and image matrix manipulation (See Appendices). Of these, we thought that segmentation and shadow removal were paths that might eventually be successful, but were too time-consuming. Shadow removal required a deeper understanding of signal processing mathematics that we didn't have. We also dropped the development of a coarse gesture detection AI because of its inaccuracy: when we experimented with this trainer by teaching it two distinct gestures (left and right), it could only perform correctly about 50% of the time on average. This amounted to the AI producing statistically random outputs (For more information, see Appendix A).

NOTE: Each student wrote a 1-2 page description of one technical approach.

***OS (Oracle 2012)***

*5.1 Modify the distro_const to create a dd'able image*

This consideration ended up looking remarkably similar to the solution we ended up with, with one key difference. We would edit a checkpoint further up the chain in 'distro_const' to correctly set up the boot record and partition tables for all future images produced in 'distro_const', thus making them all bootable. The checkpoint we had in mind was the 'create_iso.py' - if the image that checkpoint produced was already dd'able, then the .ISO image that 'create_usb.py' relied on would also be dd'able, allowing the USB checkpoint to skip that step.

We chose not to move forward with this solution because we felt it had a large impact on 'distro_const'; our focus was on producing dd'able USB images. While the solution would have worked, we didn't want to touch images that wouldn't end up as USB images. In short, the 'create_iso.py' happens too soon in the image creation process for our change to have only the desired impact, with no unintended side effects.

***Web Application (Full Contact 1, 2012)***

The initial idea for the project was to use OCR (optical character recognition) software. We evaluated many different OCR SDKs. After testing many open source solutions and some of the top paid softwares that claimed to solve the problem we were faced with, we decided that the accuracy was not high enough to further pursue this solution. Instead we decided to use the online human service which is Amazon's Mechanical TURK. We chose this because, with the addition of a straightforward web form that is used when a TURK views the HIT, the accuracy goes up significantly.

One complication of this approach is knowing when a HIT is completed. We considered using web hooks to run a function whenever a HIT was submitted. Using this method was considered throughout the project. The reason for not using this method was the uncertainty of how to create a web hook, how to test the web hook, and the fact that another solution presented itself early on in the decision making process. We decided to go with a scheduling plugin for grails called Quartz. Using this, we were able to use HTTP requests to check if there were submission available for review at an interval we define.

When a user submits a business card for transcription, the card must be stored somewhere, along with the user's information, including the username, list id, and access token. For the image file(s) we decided to use Amazon's S3 (Simple Storage Service). We decided to use this because this service is reliable, secure, and highly scalable. With the possibility of thousands of users submitting card images within a short period of time, the decision of which storage system to use focused highly on scalability and speed.

When an image is submitted, a message that indicates a new image has been submitted is created and stored. This message holds all of the information required for the system to create a HIT and retain all of the information needed to notify the correct user of the software. We wanted to stick to the centralized design that we used for storage of the images, so we decided to go with Amazon's Simple Queue Service (SQS). This allowed us to create a delimited string that contained all of the information we needed, push it onto this queue service, and pull it off

when we needed to use it. This solution provided scalability and security all within a centralized service that can be accessed by any computer that has access to the internet.

***Another Web App (Full Contact 2, 2012)***

Our first design decision was the primary language in which to implement the system. Confined to the Java Virtual Machine, we chose the Groovy language, because it remained close to the traditional Java syntax, while being considerably less verbose and supporting additional features like closures, array literals, and run-time dynamic metaprogramming. Other JVM languages such as Scala, JRuby, and Clojure were considered but were not used because using Groovy enabled us to build on developer knowledge from the rest of FullContact, who use Groovy extensively.

Our system was compiled and built by Gradle, a Groovy-based build system. Gradle has an ease-of-use advantage over similar Java build systems like Maven and Ant, while still being accessible from the command line as opposed to a build system integrated in an IDE such as Eclipse. However, the long startup times of Gradle proved an annoyance during development. When researching methods of extracting contact information from emails, we found several research papers on the subject, detailing the methods used. A paper from Microsoft Research described a machine-learning approach to extracting the signature blocks from emails, using a Java-based machine learning library (MinorThird). We opted instead for a heuristic-based approach, due to the time constraints of the project and the time required to deliver partially-finished but working software during Scrum sprints.

This method involved first attempting to find what we consider the signature in the message, and then running a second step to extract all of the contact information from this extracted signature. This made it easy to separate concerns of making sure that we have the correct contact information and finding the information in the unstructured document.

The scheme for signature extraction involves first segmenting the text into what we refer to as "blocks" - these are segments of text that are separated by non-content lines, which consist of empty lines or lines containing only a single, repeated character. This simple heuristic does a very good job of separating a person's signature from other parts of the email, such as the actual body of the message.

To choose which of these blocks represented the signature, we assigned each block a score based on the presence of certain \features"; we gave positive scores for things like the users' first name, last name, and email address, and negative scores for things like the presence of the recipient's name/email, presence of MIME headers, and indicators of reply chains.

This works fairly decently, but sometimes people do weird things with their signatures, such as double spacing their signature, or putting their signature into two or more adjacent blocks. By adding some block refinement heuristics, we were able to deal with these issues.

Data field extraction (emails, names, etc) were also handled with a heuristic approach. While an AI-based machine-learning approach might have higher accuracy in the long run, we again opted for a primarily regular expression-based approach to minimize iteration time. However, to enable the product to scale as the test data corpus and accuracy requirements grow, we designed both the signature extraction and data extraction interfaces to facilitate switching to an AI approach when a sufficient training corpus can be built.

Name extraction has by far the most extraction methods, and was also the most difficult to get to work precisely. Originally, we used only a Markov model to determine the most name-looking items; however, this had precision of around 30%. We since added ways to extract names based on common patterns within signatures, the display name provided in the MIME headers, as well as deducing it from common email address formats.

Email address and phone number extraction in many ways work similarly. Each has two primary extraction methods: One uses a simple pattern that will find most data, but with lower certainty; another uses a more comprehensive check against known common formats and yields much higher certainty. In particular, we compiled a list of common phone number patterns for nearly every major country. Email addresses may also be extracted from the MIME headers, a very reliable method which usually outperforms signature-based extraction.