Integrating Crowd Intelligence into Software

Rick Salay*, Fabiano Dalpiaz† and Marsha Chechik*
*University of Toronto, Canada
Email: {rsalay, chechik}@cs.toronto.edu
†Utrecht University, the Netherlands
Email: f.dalpiaz@uu.nl

***Abstract*—The knowledge resources available on the Internet are increasingly being used to support software both at development time and at execution time. These take the form of conventional services as well as human knowledge work both through crowd sourcing and information stored directly on the World Wide Web. But while these resources are vast and rich, they are also unreliable. In this paper, we propose a novel software development pattern called *contributional implementation* (CI) inspired by the way humans mitigate this unreliability: sources are treated as opinion providers with varying amounts of trust and aggregating multiple opinions from different sources helps improve the quality of the answers. We sketch some detailed examples of how a CI could be coded, discuss issues related to the realization of CI's in practice and outline plans for an evaluation of the approach.**

I. Introduction

The crowd has the potential to support different phases of the software engineering process. Recent research has explored the use of the crowd to help develop and test applications [1], [2], as well as for eliciting and analyzing software requirements [3], [4]. However, it can also be used directly at execution time to implement application functionality [5], [6].

In this paper, we focus on the latter use of the crowd. Furthermore, we go beyond crowdsourcing and consider how to leverage the existing information and services available on the web, much of which produced by humans, at the application execution time.

Example. To illustrate, assume that a developer is tasked with designing a novel event planning application that assists the user with planning an event. Specifically, it should include functionality to (1) recommend possible accommodations near the event being planned; and (2) assist the event organizer by tracking expenses entered by scanning receipts.

The developer realizes that using the Internet to augment a traditional approach to implementing these functions (i.e., using local computation and local databases) would be helpful. For example, for function (1), an algorithm based on a fixed set of accommodation attributes such as room cost, star rating, etc. could be developed; however, assessing accommodations is highly subjective and context sensitive and these attributes may not be sufficient. Crowd-sourced ratings available on the Internet could address this weakness and make the recommendations more useful. For function (2), while OCR algorithms for extracting information from a receipt exist, the error rate may be unacceptably high. A crowdwork approach could mitigate this problem because humans are very effective at reading printed text. In both of these examples, the quality of the function implementation can be improved by leveraging knowledge sources available via the Internet.

Leveraging Internet Knowledge Sources. We propose a new kind of software development pattern that facilitates the use of the Internet knowledge sources to implement functions such as that in the Event Planner example. In particular, we exploit the fact that the Internet provides access to sources of the following types:

- **Services**. These include "traditional" service-oriented technologies (e.g., UDDI [7], BPEL [8], BPMN [9]) and newer, more adhoc web-based API support (e.g., Google custom search API [10]) used, for example, in mash-ups (e.g., Yahoo! pipes [11]).
- **People**. These include real-time social networks, point-to-point communication mechanisms (e.g., email), crowd-based metrics (e.g., restaurant rankings), human-oriented extensions to service orientation (e.g., BPEL4People [12], WS-HumanTask [13]), and the emerging potential of "crowd-work" where humans are recruited remotely to perform tasks, typically for pay.
- **Information**. This includes the traditional network of HTTP pages on the World Wide Web as well as extensions such as linked XML content, semantic web technologies (e.g., RDF [14]), RSS feeds [15], multimedia content, etc.

These three types of knowledge sources provide a rich potential resource; however, there are characteristics of the Internet that make it challenging to develop software that interacts with them:

- The Internet is *unmanaged*: any interaction with it may be unreliable. This includes the fact that the accessed services, people or information may become unavailable at unpredictable times, may take more time than expected, may be unable to fulfill a request or may appear to succeed but in fact returns partial or incorrect results.
- The Internet is *dynamic*: it is continuously changing, and the software must be able to adapt to cope with disappearing services/people/information and to maximally exploit new possibilities.
- Interaction via the Internet may incur a *monetary cost*, e.g., transaction charges for services, payment to people for performing tasks, information subscription fees, etc.
- The Internet is a *social* rather than a technical space.

> Software should be able to use different techniques to interact with different sources of information. For example, services might be accessed through web service techniques whereas interactions with information call for text processing techniques.

Recognizing that the Internet is a social space [16] inspired us to consider how people address these challenges when interacting with the Internet. They mitigate the unreliability of the Internet by getting "opinions" from multiple sources and aggregating the results. Furthermore, they try to assess the trustworthiness of these sources using their own subjective assessments as well as measures of reputation such as crowd-based ratings. The dynamic nature of the Internet means that people must frequently reassess their information sources and discover new ones. Monetary costs play a natural role in way people assess the Internet resources they use: they know their budget and ideally attempt to stay within it.

Contributions. We introduce a novel software development pattern called *contributional implementation* (CI) that facilitates the implementation of functions using Internet knowledge sources and mitigates the challenges described above. In contrast to approaches that embed the interaction in an ad-hoc manner (e.g., [5]), we propose a disciplined approach that relies on patterns, an effective tool in software engineering [17]. Specifically, the CI approach simplifies the specification and management of the trust and cost of knowledge sources as well as how opinions from sources are aggregated. It does so by providing coding abstractions (see Section II) that a developer can use to more easily and uniformly interface her developed application with external sources.

The contributions of this paper include:

- an identification of the key components of a CI;
- a detailed and suggestive example of the use of a CI;
- a discussion of the issues with realizing a CI in practice.

The rest of this paper is structured as follows. In Section II, we describe the components of a CI and illustrate how to implement the Event Planner example using CIs. In Section III, we discuss the issues related to realizing the components of a CI. In Section IV, we discuss related work. In Section V, we conclude and discuss next steps.

II. Coding with Contributional Implementations

In this section, we introduce the CI pattern using two programming constructs: **Ask** and **Source**. Together these two constructs can be used to implement functions by aggregating opinions from multiple knowledge sources on the Internet about what the function result should be for a given input value. The **Ask** construct is used to query one or more sources and provides the means to specify different aspects such as budget, trust, etc. The **Source** construct is used to interface with particular knowledge sources consulted by an **Ask**. We illustrate how a developer might program the Event Planner example functionality using these constructs.

A. The *Ask* construct

Ask is used to specify a query to a set of sources and aggregate the result into a single value. Since we are not assuming any specific programming language, we use a hypothetical syntax in order to identify the different parts of the construct. These parts, illustrated by the code sketches in Fig. 1, are as follows:

- One or more *source-use* entries (e.g., lines 3–6) identifying the soures to be consulted. These consist of the following parts:
  - *source name*: This refers to a source defined using the **Source** construct elsewhere in the program (see Section II-B). For example, `RatingServiceRank` on line 3 refers to the source in line 22.
  - *argument list*: This is a list that contains the input values corresponding to the formal parameters defined in the **Source** construct.
  - *trust*: These values indicate how much confidence we have in an opinion expressed by the source.
- *aggregator*: The aggregator is used to combine the opinions from the consulted sources in order to produce a result for the function being implemented using the CI. The particular aggregator used depends on the context and type of value being aggregated. For example, the aggregator in line 2 is `Rank` which uses trust to combine ranked lists of items to produce the resultant ranked list. Another kind of aggregator could take the first opinion produced by a sequence of sources as the result (e.g., `Sequential` in line 28), etc. The aggregator is typically be defined through a class or an external function, depending on the chosen programming language. A standard library of reusable aggregators could be provided to reduce the effort for the programmer.
- *budget*: Consulting sources takes time and may cost money. The budget defines the maximum total time and cost that an **Ask** can spend in consulting sources. For example, the monetary budget for the **Ask** in line 2 is $10 while the time budget is 3 days.
- *source selection strategy*: This defines the order in which sources should be consulted. At the most general level, it performs a multi-objective optimization to find an order of consulting sources that stays within the budget and minimizes the time and the cost while maximizing the confidence in the result. Since these objectives cannot always be simultaneously achieved, specific preferences may be given to guide the strategy. For example, minimizing cost may be preferred over minimizing time. For example, the **Ask** in line 28 specifies that cost should be minimized (MinCost).

B. The *Source* construct

This construct defines the details of interacting with a particular knowledge source. It can be seen as a "driver" that enables plugging the actual sources into the calling program.

We introduce the **Source** construct as a way to support reuse: the same driver can be used in different applications.

A **Source** is characterized by a name, a list of arguments, a list of source attributes that define cost, time, and availability values for the source, and the body of the source. This last element contains the code that connects to the actual source by providing the right parameters and interpreting the results.

C. Illustration

Fig. 1 presents code snippets for two functions of our hypothetical Event Planner application: function `RecommendAccommodNear` (lines 1–9) is concerned with the recommendation of accommodation near to a venue, while function `DetermineEligibleExpense` (lines 27–34) is about the determination of the eligible expenses from a scanned receipt. Moreover, the figure also includes the partial declaration and body of some sources that the two functions use. Below, we illustrate the main features of coding with contributional implementations.

Function RecommendAccommodNear (lines 1–9). It returns a sorted list that represents recommended accommodations within a certain distance (parameter *maxDist*) from the location (parameter *venueCoord*) where an event will be held. The function relies on an **Ask** (lines 2–7) that aggregates four different sources, each of which returns an ordered list of hotels: `RatingServiceRank` exploits crowd-sourced information retrieved from social rating services such as Booking.com[1]. `RankHotelsFromDB` is a "traditional" (rather than Internet-enabled) algorithm that queries the internal database of the event planning company, `SurveyMonkeyRank` exploits crowdwork by creating a SurveyMonkey[2] survey to let previous participants judge their preferred accommodations, and `GMapsRankByDistance` exploits Google Maps to find the closest hotels.

The aggregation is done using a *Rank* function, meaning that a list of options is going to be ranked based on the results from the sources. The selection algorithm is based on an average with weights determined by the trust level of every source (*Rank*⟨*WAvg,trust*⟩). In this specific case, the highest trust level is that of the social rating services (0.9), followed by the survey (0.85), the ranking in the local database (0.75), and the simple distance-based ranking of Google Maps (0.6). The selection function is *Parallel*: all the sources are consulted at the same time. The **Ask** function is also fed with a maximum budget (its cost attribute is set to $10) and with a timeout of three days, defining the maximum time the function should be waiting for responses from the sources.

Table I shows possible results for an invocation of the **Ask** in line 2, and the results of the aggregation. Assume that our implementation assigns a number of points to the first three hotels that every source returns: 10 points for the first hotel, 6 points for the second, and 4 points for the third. The aggregation function *Rank*⟨*WAvg,trust*⟩ has to compute an average of the hotel points with weights, where weights correspond to the trust of each source. By doing so, the **Ask** returns a list where the first place is occupied by the hotel Magic, the second – by City, etc, as shown in the table.

TABLE I
POSSIBLE RESULTS FROM THE SOURCES OF THE **ASK** IN LINE 2 OF FIG. 1 AND RESULTS OF THE AGGREGATOR *Rank*⟨*WAvg,trust*⟩

Source	Trust	#1 (10pt)	#2 (6pt)	#3 (4pt)
`RatingServiceRank`	0.9	City	Magic	Luxury
`RankHotelsFromDB`	0.75	Magic	City	-
`SurveyMonkeyRank`	0.85	Magic	City	Budget
`GMapsRankByDistance`	0.6	Luxury	Budget	Magic

Rank	Hotel	Calculation	Total
#1	Magic	$6 \cdot 0.9 + 10 \cdot 0.75 + 10 \cdot 0.85 + 4 \cdot 0.6$	20.9
#2	City	$10 \cdot 0.9 + 6 \cdot 0.75 + 6 \cdot 0.85$	18.3
#3	Luxury	$4 \cdot 0.9 + 10 \cdot 0.6$	9.6
#4	Budget	$4 \cdot 0.8 + 6 \cdot 0.6$	6.8

Despite the many contacted sources, the function is robust enough to handle exceptions: in line 8, the case where no hotel is found within the given constraints, the value *Unk* is returned, indicating that the contributional implementation function could not identify any hotel, due to the lack of knowledge. This differs from returning an empty set, which signifies that no hotels exist.

Source SurveyMonkeyRank (lines 10–19). The code shows a possible implementation of a source for ranking a set of options through the SurveyMonkey online survey system, and relies on the existence of an API to connect to such a system. In lines 11–14, a form is created including a multiple choice question which requires respondents to select one option from a list. Once created, the survey is then distributed to a set of invitees (lines 15–16). The function then waits (line 17) until a timeout, defining the time during which responses are accepted. Finally (line 18), the obtained responses are returned, ranked by the number of preferences that the respondents assigned to the hotels, thereby representing the community's opinion on the best accommodation nearby the event venue.

In the case of the Event Planner, the source `SurveyMonkeyRank` is invoked (line 5, function `RecommendAccommodNear`) by specifying that the ranking concerns a list of hotels, the invitees are the participants who participated in previous editions of the event, the hotels are obtained from the database of the Event Planner system, and the timeout is two days.

Other sources for `RecommendAccommodNear` are sketched in lines 20–26. Lines 20 and 21 present possible declarations for the source that implements the ranking based on google maps, and the one based on the local database, respectively. Lines 22–26 provide slightly more detail on the source `RatingServiceRank`, showing that such a wrapper may make use of JSON[3] requests for social rating websites such as booking.com and TripAdvisor[4].

[1] www.booking.com
[2] www.surveymonkey.com
[3] www.json.org
[4] www.tripadvisor.com

```
1 String [] RecommendAccommodNear(Coordinates venueCoord, int maxDist) {
2   String [] rankedList = Ask Rank⟨WAvg,trust⟩ Parallel cost($10) time(3d) {
3     RatingServiceRank(venueCoord, maxDist) trust(0.9);
4     RankHotelsFromDB(venueCoord, maxDist) trust(0.75);
5     SurveyMonkeyRank('hotels', DB.GetPreviousParticipantsMails(),
        DB.FindAccommodations(venueCoord, maxDist), 2d) trust(0.85);
6     GMapsRankByDistance(venueCoord, 'hotels', maxDist) trust(0.6);
7   }
8   if (rankedList==∅) return Unk else return rankedList;
9 }
10 Source String [] SurveyMonkeyRank(String title, String [] recipients, String [] list, Time timeout) {
11   Form form = SurveyMonkey.create('Help us with ranking the following ' + title);
12   MCItem item = new MultipleChoiceItem('Which one do you like the most?');
13   item.setChoiceValues(list);
14   form.addMultipleChoiceItems(item);
15   for each rec in recipients
16     Mail.send('Help wanted with ' + title, form.getPublishedUrl(), rec);
17   Time.waitFor(timeout);
18   return Form.getRankedResponses();
19 }
20 Source String [] GMapsRankByDistance(Coordinates coord, String search, int maxDist) time(50ms) { ... }
21 Source String [] RankHotelsFromDB(Coordinates coord, int maxDist) time(100ms) { ... }
22 Source String [] RatingServiceRank(Coordinates coord, int maxDist) time(10s) {
23   JSONRequest booking = new JSONRequest('booking.com', coord, maxDist, ...);
24   JSONRequest tripadv = new JSONRequest('tripadvisor.com', coord, maxDist, ...);
25   ...
26 }
27 Float DetermineEligibleExpense(Image receipt, String handlingClerkMail) {
28   Float amount = Ask Sequential MinCost cost($0.2) time(1d) {
29     OCRParseExpense(receipt, 75%);
30     MTurkPostJob('Extract eligible expense', receipt);
31     AskClerk('Extract eligible expense', handlingClerkMail, receipt);
32   }
33   if (amount==null) return Unk else return amount;
34 }
35 Source Float OCRParseExpense(Image receipt, Percentage confidence) cost($0.01) time(0.1s) { ... }
36 Source Float MTurkPostJob(String jobSummary, Object object) cost($0.2) time(22h) { ... }
37 Source Float AskClerk(String task, String clerkMail, Image doc) cost($5) time(2h) { ... }
```

Fig. 1. Code snippets of functions `RecommendAccommodNear` and `DetermineEligibleExpense`, along with possible sources

Function DetermineEligibleExpense (lines 27–34). This returns the eligible expense reimbursement based on processing of the receipt images (parameter *receipt*). The function is intended for expenses incurred in the process of organizing an event through our hypothetical application. The function employs an **Ask** aggregation (lines 28–32) that combines an algorithm based on optical character recognition (`OCRParseExpense`), posting receipt processing tasks on the crowdwork platform Amazon Mechanical Turk (`MTurkPostJob`), and asking the handling clerk (`AskClerk`).

This aggregation employs the *Sequential* aggregator, meaning that the sources are consulted one after another, and that further sources are consulted only if the result is still unknown. The order of invocation uses the *MinCost* selection criteria; looking at the sources defined in lines 35–37, this would mean that the function `OCRParseExpense` is invoked first, followed by `MTurkPostJob` if no definite response is obtained, and finally followed by `AskClerk` if both of the previous two sources return *Unk*.

Notice that in line 29 and line 35, the OCR parsing is invoked by passing the value 75% for the argument *confidence*. This is due to the fact that image-to-text algorithms typically return results with a certain degree of certainty about the correctness of the recognition; in this case, if the returned confidence is below 75%, the source would return *Unk*, meaning that the source is unable to provide an answer with the requested confidence level. Finally, notice that, should the aggregation in line 28 be based on *MinTime* instead of *MinCost*, the ordering of source invocation would change: `OCRParseExpense` (time 0.1s), followed by `AskClerk` (time 2h), and finally `MTurkPostJob` (time 22h).

III. Realizing Contributional Implementations

Fig. 2 shows a high-level architecture to support the CI development pattern. The programmer provides **Ask** and **Source** specifications as illustrated in Fig 1. Standard CI library functions are available to simplify this task. For example,

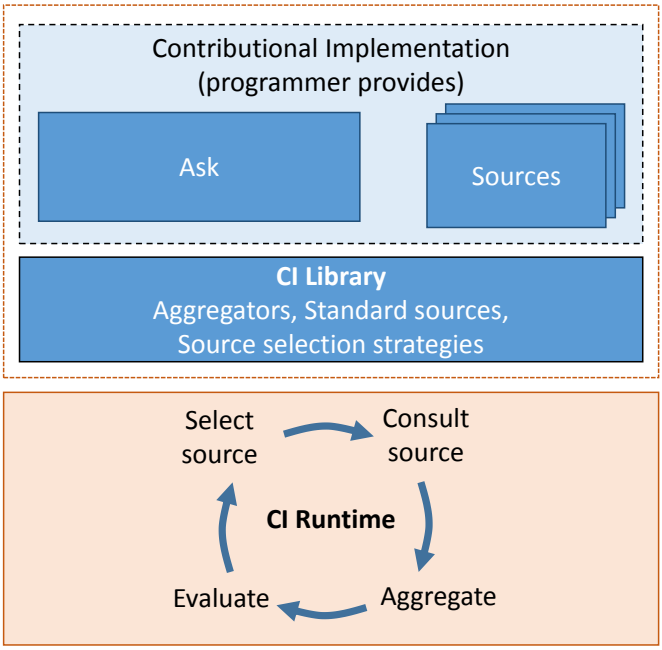

Fig. 2. Architecture diagram for contributional implementations showing the programmer provided components, library support and runtime middleware.

this could include aggregators (e.g., weighted average, rank, sequence, etc.), wrappers for standard source types (e.g., search engines, crowd work platforms, etc.), and source selection strategies (e.g., minimize cost, minimize time, etc.). The CI runtime layer enables seamless integration with current programming frameworks. It also implements synchronous and asynchronous versions of the **Ask** construct via the core "select-consult-aggregate-evaluate" loop:

1) Select source: The source selection strategy is invoked to select one or more sources that match its time, cost and trust criteria for being the next to consult.
2) Consult source: The selected sources are consulted by invoking the correponding **Source** constructs.
3) Aggregate: The aggregator is used to combine the opinions returned from the consulted sources with the opinions from all previous cycles of the core loop.
4) Evaluate: Based on the quality of the current aggregated result, the source selection strategy determines whether it should continue with another cycle of the core loop or exit.

A. Extending the CI pattern

In this paper, we have sketched a rudimentary version of the CI pattern. If we consider what a full-featured version of the CI pattern would address, there are several additional challenging issues to consider and we briefly discuss them in this section.

Action vs. Query functions. In this paper, we have focused on functions that are queries (e.g., to rank accommodations). Functions that perform actions (e.g., prepare a poster for an event) are challenging to handle contributionally because their effect is to change the state of the world, and, unlike a query, it is not clear how to aggregate these effects.

Other costs of consulting a source. In this paper, we have considered only the monetary cost and the time for consulting a source. However, other kinds of costs are also possible: memory consumed, environmental footprint, etc. These require further investigation.

Incurring costs is source type-specific. For example, invoking an algorithm locally often has no (or low) monetary cost but doing so remotely (i.e., via a service-type source) may have a transaction cost associated with it. Remote invocation also incurs network latency time costs. Consulting a people-type source typically involves a monetary cost in the form of a transaction fee. The required timeframe for completing the task can be specified; however, there is no guarantee that the task will be completed in the required time. To increase the chance that a person accepts a task and delivers on time, the cost may include providing different types of incentives and rewards (including game elements such as badges and leaderboards in social networks). For an information-type source, the access fee is often a subscription fee for a longer time period. Thus, a per-invocation cost can only be estimated. Moreover, the cost of a source may vary over time, and proper mechanisms to deal with this dynamism need to be devised.

Since our focus is to provide a framework for CIs that applies to any source type, we are primarily interested in identifying the different *kinds* of cost and their characteristics rather than their source-specific details. Such details are addressed by complementary approaches, e.g., the field of the human computation research (see Section IV).

Trust in a source. This paper provided a simplistic account of trust which is an extremely complex concept [18]. Trust is dynamic and varies depending on the evolution of the social context and as a result of the interaction with a source. Moreover, trust depends not only on experiences with a source, but is often affected by referrals from other actors' experiences with a source. An important aspect in CI development is thus to provide a comprehensive view of trust as a function of these factors instead of a simple constant (as in our example). Furthermore, being a social notion, it seems likely that computing the trust in a source may require its own CI. For example, the trust level of a source could be based, in part, on the crowd-sourced reputation information available on the Internet.

Source discovery and interaction. In our examples, we have assumed that the sources were known by the calling application. This is certainly not always the case – as we discussed in Section I, the dynamically changing nature of sources on the Internet could require the discovery of new sources and deciding when to augment existing ones or replace defunct ones. Some important questions to address are: How to determine when a static or dynamic set of sources should be used? How to dynamically discover sources with different types? Interestingly, dynamic discovery itself can be a CI.

Table II discusses some alternatives for the discovery and invocation aspects of the three source types.

TABLE II
MECHANISMS OF INTERACTION WITH DIFFERENT SOURCE TYPES

Type	Discovery	Invocation
Service	Location services for service oriented infrastructures over the Internet such as the UDDI registry for the Web services. Discovery of websites is also be possible using a search engine query (e.g., Google).	Web services invocation interface protocols such as SOAP [5] or RESTful [19] APIs. For websites discovered using a search engine, web scraping tools (e.g., Scrapy [6]) can be used to programmatically interact with the website.
People	Crowd work platforms such as Mechanical Turk allow tasks to be posted and then accepted by a person willing to do the task.	A social connection mechanism such as Email or social networking platform may be used to exchange information with the person. An initial interaction required is either the implicit (i.e. upon acceptance of the task) or explicit "signing" of a contract between the requester and task performer. Information exchange related to the task may require the generation and interpretation of natural language.
Information	Search engines index the Web and provide a way to locate information via a query.	Web content may be available in machine readable form (e.g., XML) or can be harvested using web scraping tools.

IV. RELATED WORK

In this section, we briefly review research that is closely related to the concept and realization of contributional implementation.

Human computation [20] is a computing paradigm in which task execution is delegated to online humans to perform. Crowdsourcing is the special case where humans are found through an open call. Research in human computation includes proposals for middleware frameworks to simplify interacting with humans (e.g., [21], [6], [22]), guidance on how to design tasks to make them more appealing to humans (e.g., [23]), and methods to improve the quality of the results produced by humans (e.g., [24]).

In general, human computation approaches recognize the need for aggregating results from different human judges to improve the quality of results; however, the focus here is specifically on special requirements of human sources. In contrast, the contributional implementation pattern is independent of the types of sources used. This key principle is intended to separate the concern of software development from the details of the different types of sources.

Self-adaptive software systems [25] is an approach for automatically changing a system strategy or configuration at runtime to address different contexts or system failures. Thus, a self-adaptive approach could be used to mitigate the unreliability of the Internet. These systems assume the existence of a predefined and finite set of alternatives from which a configuration can be chosen and a design space among which the system can choose at runtime [26]. In contrast, CIs have a narrower scope – to compute a function – and a less restricted approach because the design space is bounded only by the existence of Internet sources. For example, a CI can be used to ask a source to return a set of sources for a given task.

Multi-agent systems [27] consist of autonomous agents that interact to achieve their respective goals. Like a CI, agents use trust information to control their interactions. However, multi-agent systems are typically implemented through the use of shared middleware (e.g., JADE [28]). Our approach, on the contrary, is aligned with a more decentralized view on multi-agent systems, although there is some work in this direction that relies on social primitives such as commitments [29], [30]. These works may constitute the theoretical foundation for the social reliance that CIs call for.

Social computing [31] recognizes the crucial role that humans play, by participating in the Internet, in assisting computers in problem solving. An interesting initiative within this field is a *social computer* [32], an approach that enables people to initiate computations and adopt roles within an existing computation, while preserving certain social properties. While these approaches do not consider the idea of aggregating results from multiple sources, their programming paradigm suggests useful primitives to represent and evolve computations that transcend the boundary of a software system.

V. CONCLUSION AND STATUS

The knowledge resources available on the Internet are increasingly being used to support software both at development time and at execution time. These take the form of conventional services as well as human knowledge work both through crowd sourcing and information stored directly on the World Wide Web. But while these resources are vast and rich, they are also unreliable. In this paper, we proposed the contributional implementation (CI) software pattern inspired by the way humans mitigate this unreliability: sources are treated as opinion providers with varying amounts of trust and getting multiple opinions from different sources and aggregating them helps improve the quality of the answers. We sketched some detailed examples of how a CI could be coded and discussed the issues related to them.

An important next step is to evaluate the practical feasibility of using CI's and the potential benefits they may provide. To this end we are close to completing a Java-based prototype for the development time and run time infrastructure required for supporting a CI as shown in Fig. 2. Specifically, we have defined a small set of standard aggregators, provided a customizable source selection strategy and have defined a set of abstract classes to implement the generic **Ask** and **Source** functionality. Once this is complete, we plan on conducting some case studies to implement applications using CI's. In concert with this activity we intend to develop guidelines on

how and where to best use CI's within an application. Our eventual goal is to make the CI a first class concept for programming in a post-Internet age.

REFERENCES

[1] K.-J. Stol and B. Fitzgerald, "Two's company, three's a crowd: A case study of crowdsourcing software development," in *Proceedings of ICSE*. New York, NY, USA: ACM, 2014, pp. 187–198. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568249

[2] T. D. LaToza, M. Chen, L. Jiang, M. Zhao, and A. van der Hoek, "Borrowing from the crowd: A study of recombination in software design competitions," in *In Proceedings of ICSE (to appear)*, 2015.

[3] S. L. Lim, D. Quercia, and A. Finkelstein, "Stakesource: harnessing the power of crowdsourcing and social networks in stakeholder analysis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 2010, pp. 239–242.

[4] R. Snijders, F. Dalpiaz, M. Hosseini, A. Shahri, and R. Ali, "Crowd-centric requirements engineering," in *Proceedings of the Second International Workshop on Crowdsourcing and Gamification in the Cloud (CGCloud 2014)*, 2014.

[5] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich, "Soylent: A Word Processor with a Crowd Inside," in *Proceedings of the 23nd annual ACM Symposium on User Interface Software and Technology*. ACM, 2010, pp. 313–322.

[6] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin, "CrowdDB: answering queries with crowdsourcing," in *Proceedings of the ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 61–72.

[7] OASIS, *UDDI Version 3.0.2*, http://xml.coverpages.org/UDDIv302-CommSpec20041019.pdf, Std., 2004.

[8] ——, *Web Services Business Process Execution Language Version 2.0*, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, Std., 2007.

[9] Object Management Group, *Business Process Model and Notation (BPMN) 2.0*, http://www.omg.org/spec/BPMN/2.0, Object Management Group Std., Jan 2011. [Online]. Available: http://www.omg.org/spec/BPMN/2.0

[10] Google Inc., "Google Custom Search API," 2014. [Online]. Available: https://developers.google.com/custom-search/

[11] Yahoo! Inc., "Yahoo! Pipes," 2014. [Online]. Available: http://pipes.yahoo.com/pipes/

[12] OASIS, *WS-BPEL Extension for People (BPEL4People) Specification Version 1.1*, http://docs.oasis-open.org/bpel4people/bpel4people-1.1.html, Std., 2010.

[13] ——, *Web Services - Human Task (WS-HumanTask) Version 1.1*, http://docs.oasis-open.org/ns/bpel4people/ws-humantask/200803, Std., 2010.

[14] W3C, *RDF Schema 1.1*, http://www.w3.org/TR/2014/REC-rdf-schema-20140225/, Std., 2014.

[15] ——, *RSS 2.0 specification*, http://validator.w3.org/feed/docs/rss2.html, Std., 2002.

[16] K. Hellenga, "Social space, the final frontier: Adolescents on the internet," *The changing adolescent experience: Societal trends and the transition to adulthood*, pp. 208–249, 2002.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[18] C. Castelfranchi, R. Falcone, and F. Marzo, "Being Trusted in a Social Network: Trust as Relational Capital," in *Trust Management*. Springer, 2006, pp. 19–32.

[19] R. T. Fielding and R. N. Taylor, "Principled Design of the Modern Web Architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002.

[20] E. Law and L. v. Ahn, "Human Computation," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 5, no. 3, pp. 1–121, 2011.

[21] P. Minder and A. Bernstein, "Crowdlang: A Programming Language for the Systematic Exploration of Human Computation Systems," in *Social Informatics*. Springer, 2012, pp. 124–137.

[22] S. Kochhar, S. Mazzocchi, and P. Paritosh, "The Anatomy of a Large-Scale Human Computation Engine," in *Proceedings of the ACM SIGKDD Workshop on Human Computation*. ACM, 2010, pp. 10–17.

[23] E. Huang, H. Zhang, D. C. Parkes, K. Z. Gajos, and Y. Chen, "Toward Automatic Task Design: a Progress Report," in *Proceedings of the ACM SIGKDD Workshop on Human Computation*. ACM, 2010, pp. 77–85.

[24] P. G. Ipeirotis, F. Provost, and J. Wang, "Quality Management on Amazon Mechanical Turk," in *Proceedings of the ACM SIGKDD Workshop on Human Computation*. ACM, 2010, pp. 64–67.

[25] J. Andersson, L. Baresi, N. Bencomo, R. de Lemos, A. Gorla, P. Inverardi, and T. Vogel, "Software Engineering Processes for Self-Adaptive Systems," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS. Springer, 2013, vol. 7475, pp. 51–75.

[26] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel *et al.*, "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS. Springer, 2013, vol. 7475, pp. 1–32.

[27] J. Ferber, *Multi-Agent Systems: an Introduction to Distributed Artificial Intelligence*. Addison-Wesley Reading, 1999, vol. 1.

[28] F. Bellifemine, A. Poggi, and G. Rimassa, "JADE–A FIPA-compliant Agent Framework," in *Proceedings of the 4th International Conference on the Practical Application of Intelligent Agents and Multi Agent Technology (PAAM'99)*, 1999, pp. 97–108.

[29] M. P. Singh, "An Ontology for Commitments in Multiagent Systems: Toward a Unification of Normative Concepts," *Artificial Intelligence and Law*, vol. 7, no. 1, pp. 97–113, 1999.

[30] M. P. Singh and A. K. Chopra, "Programming multiagent systems without programming agents," in *In Proceedings of ProMAS*, 2009, pp. 1–14. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14843-9_1

[31] F.-Y. Wang, K. M. Carley, D. Zeng, and W. Mao, "Social Computing: From Social Informatics to Social Intelligence," *Intelligent Systems, IEEE*, vol. 22, no. 2, pp. 79–83, 2007.

[32] D. Robertson and F. Giunchiglia, "Programming the Social Computer," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 371, no. 1987, 2013.