DSML-Aided Development for Mobile P2P Systems

Tihamér Levendovszky, Tamás Mészáros, Péter Ekler, Márk Asztalos
Department of Automation and Applied Informatics

Budapest University of Technology and Economics
H-1111 Budapest
Goldmann György tér 3. IV. em.
Tel.:+36-1-463-2870

{tihamer, mesztam, ekler.peter, asztalos}@aut.bme.hu

ABSTRACT
The proliferation of Mobile P2P systems made a next generation mobile BitTorrent client an appropriate target to compare two different development approaches: the traditional manual coding and domain-specific modeling languages (DSMLs) accompanied by generators. We present two DSMLs for mobile communication modeling, and one for user interface development. We compare the approaches by development time and maintenance, using our modeling and transformation tool Visual Modeling and Transformation System (VMTS).

Categories and Subject Descriptors
D.2.2 [**Software Enginering**]: Design Tools and Techniques – *state diagrams, user interfaces.* D.2.13 Reusable Software – *domain engineering.*

General Terms
Design, Languages

Keywords
Domain Engineering, Methodologies, Graphical environments, Interactive environments, Specialized application languages

1. INTRODUCTION
Mobile Peer-to-Peer technology is a natural demand fueled by the appearance of Smart Phones on the market. The Applied Mobile Research Group at our department did pioneering work in this area. *Symella*, the first *Gnutella* client for Symbian OS, has been downloaded by more than 400,000 users since its first public release in the summer of 2005. *SymTorrrent* is the first *BitTorrent* client for mobile phones. The first free public version was available in 2006 October, as of writing the software has been downloaded by about 300,000 clients. In order to involve mainstream phones into P2P networks, Péter Ekler has developed a *BitTorrent* client named *MobTorrent* for Java ME platform [1]. The original goal was to examine whether mainstream phones are able to run such complex applications. The experiment has met the expectations, and *MobTorrent* became a suitable for communicate with the *BitTorrent* network. The experience stemming from the products made *MobTorrent* an apt environment where we could compare the manual coding and the Domain-Specific Modeling Language (DSML)-aided development.

Having developed the manually coded version, we started with creating domain-specific languages which can be used to describe P2P systems for mobile applications. We identified two main functionality groups where the DSMLs are useful: **processing the protocol messages** and **designing the user interface**. As a DSML platform, we chose the metamodeling and model transformation tool Visual Modeling and Transformation System (VMTS) [2]. In VMTS, we could create the DSMLs, and we could write the model processors that translate the models into Java ME code.

We tried to address the following issues:

- How can Mobile P2P application benefit from DSMLs?
- Does the DSML technology pays off at all in mobile P2P development in time?
- Do the domain-specific models require less maintenance effort?
- Could the DSML approach accelerate the development of the future versions?

We start with answering the first question by giving an insight of the used DSLs, moreover, we show how we got the facts that underpin the answers.

2. Domain-Specific Languages for Mobile P2P Systems
In VMTS, we developed an integrated environment to visually model different aspects of JAVA ME mobile applications, and code generators to turn the models into executable JAVA code. The *Java Resource Editor* DSL is appropriate for the rapid development of the static components of mobile applications, while the *JAVA Network Protocol Designer* can be used to model the static components and the dynamic behavior of simple, message-based network protocols.

2.1 Java ME Network Communication Support
The basics of BitTorrent technology [3] are as follows. In order to download content via *BitTorrent*, firstly we need a very small torrent file. This file contains some meta-data describing the content and the address of at least one central peer called *Tracker*, which manages the traffic. After we have the torrent file, the *BitTorrent* client connects to the *Tracker*, which sends a set of addresses of other peers back to the client. Next the client connects to these addresses and concurrently downloads the content from them via a BitTorrent-specific *peer-wire protocol* [2]. In BitTorrent, we can download the content simultaneously from different peers.

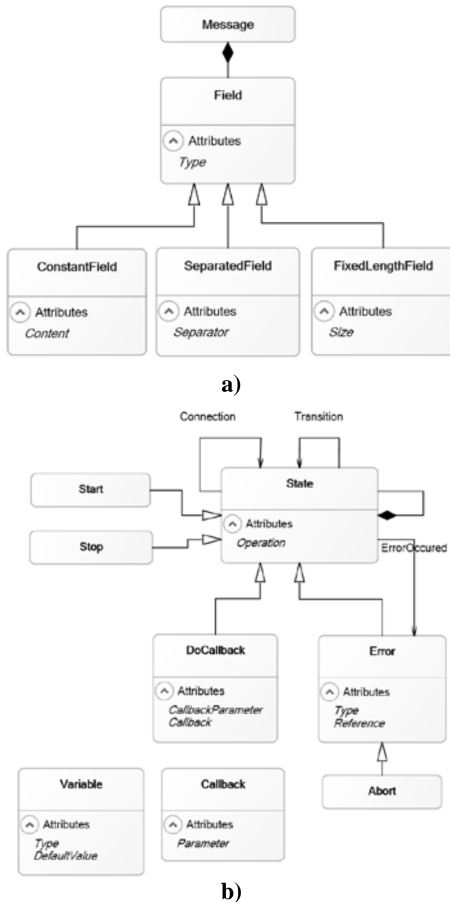


Figure 1 Metamodels for modeling the static and dynamic properties of message-driven state meachines

We developed two DSMLs for modeling the static and dynamic aspects of message-based network protocols, an integrated configuration environment and code generators to support the rapid modeling and implementation of communication through the network. It is capable of describing the peer-wire protocol and its processing logic. Our solution exploits the fact that numerous well-known and widely used network protocols take a message-based approach. This means that the entities communicating with each other use a well-defined language, which consists of exactly identifiable elements with a predefined structure. The *MessageStructure* DSML models the messages (the static components) of such a protocol. Furthermore, the *MessageProcessor* DSML is provided to describe the logic of a protocol. We use hierarchical state machines to define this logic: we can declare the possible incoming messages in a state and the messages to be sent when leaving a state.

With the help of model processors, we can generate a standalone network library, which can be adapted to the user interface or to business logic components. The generated network library provides its services through a unified callback interface. Via this interface it is possible subscribe to numerous events fired by the library during communication.

2.1.1 Modeling messages

Figure 1.a presents the metamodel of the *MessageStructure* DSML. The most important item of this language is the *Message* itself. The message is the unit of the communication of our approach. Each byte sent over the wire has to be the part of a message.

Each message consists of several *Field*s. *Field*s are the building blocks of the messages. *Fields* have a *Type* attribute which corresponds to a simple Java type. Currently *int, byte* and *String* types are supported. We distinguish three different types of fields: *ConstantField, FixedLengthField,* and *SeparatedField.* A *ConstantField* has an additional *Content* attribute which is used to define the exact content of such a field at modeling time. In a protocol where a user ID (e.g.: 123) is sent in the format of *#userid#123*, the *#userid#* part of the message is a *ConstantField*. When reading a message from the network stream, the content of a *ConstantField* must be found at the position defined by the field in the model. Otherwise, the message processing fails. Thus, *ConstantFields* play an elementary role when distinguishing between possible incoming messages in a certain processing state.

FixedLengthFields do not have a predefined content, but a predefined size (*Size* attribute). This means that the field represents a buffer for *Size* pieces of elements of type *Type. The Size* attribute does not have to be a constant value, instead, it can be contained by a field of the same message or global variable (see later), or an aggregated value of those. This means that if we recognize a *FixedLengthField* in a message it is possible that the size of this *FixedLengthField* depends on the already read content of a previous field in this message. *SeparatedField*s do not have a predefined value or size. Their start and end are marked by a character sequence specified in their *Separator* attribute. Reading such a field is finished with reading the value of the *Separator attribute* from the stream. This is a useful feature for textual protocols (e.g FTP or POP3), where the commands are separated with line-break characters.

During code generation, Java classes are created based on the message elements. The contained fields of the messages will correspond to the fields of the Java class. Based on the model and the order of the fields of the messages, we also generate the member methods to read or write the message from or to the network stream. With the help of modeling messages and generating their wrapper classes, our solution completely hides byte-wise network stream operations, and provides an interface based on Java objects to the upper layers of the application.

2.1.1.1 BitTorrent messages

In order to discover and filter the incoming messages described with the *MessageStructure* DSML, we have implemented a message discovery algorithm. After a message is parsed, a callback method is being called which carries the different type of *MessageFields as* parameters. This callback method is used by the *MobTorrent* framework to execute BitTorrent-specific functions such as save the incoming data in a file.

Figure 2 presents the model of the BitTorrent protocol messages. The green fields are the *ConstantFields*, whereas the grey ones are the *FixedLengthFields*. BitTorrent protocol does not use *SeparatedFields*. Usually in every message-based protocol, the messages have a common structure. In the case of BitTorrent we can separate the messages into two parts. The first part contains the *MessageHandshake* (Figure 2) only, which is used during the *peer-wire protocol* to determine whether two peers are compatible with each other. *MessageHandshake* starts with two *ConstantFields* followed by three *FixedLengthFields*. The most

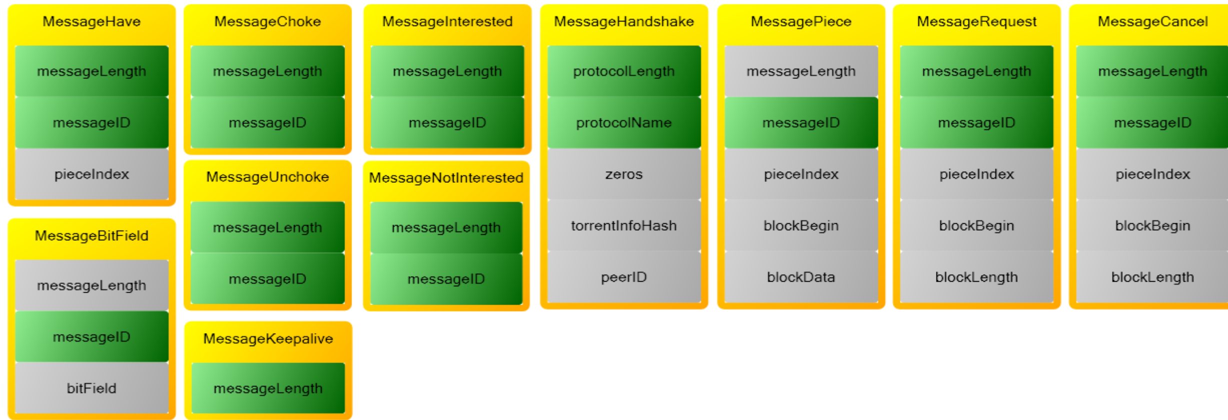


Figure 2 Message objects used by the BitTorrent protocol

important field in the *MessageHandshake* is the *torrentInfoHash*, which is basically the SHA-1 value of the torrent file. This value is used to determine whether the peers are interested in the same content represented by the torrent file.

According to the protocol when peers are exchanging the handshake message, this is the only message which can be accepted, thus, it is easy to discover. After a successful handshake every other message can be sent or received, there are no limitations. However we can see that the structure of these messages is the same. All of them start with a *messageLength* field which defines the length of the message in four bytes. Figure 2 shows that the *messageLength* field is green in all the messages, except for *MessagePiece* and *MessageBitfield*. The length of these two messages depends on the amount of data they carry.

Following the *messageLength*, each message contains a *messageID* field which makes it easy to filter the messages. Only the *MessageKeepAlive* does not have this *messageID* field, because it contains only a *messageLength* constant field.

In the *BitTorrent* protocol, after we have parsed the *messageID* field, we can easily determine which message has arrived, and we can pass the content of the incoming message as the parameters of the callback functions.

2.1.2 Modeling dynamic behavior

The core concept of our approach is that that communication layer performs status changes as a consequence of receiving specific messages from the network stream. In addition, we may also instantiate and send network messages during a status change. In our approach, the communication layer can run standalone, and it informs the connecting components of the application through a callback interface about the important events of the communication. The business logic can influence the behavior of the communication layer through the parameters of the layer and by sending messages directly through the network stream. The behavior of the network layer can be modeled with the help of a message-driven state machine.

Figure 1.b presents the metamodel of the *MessageProcessor* DSL. The most elementary item of this state machine is the *State*. There are two special types of states: the *Start* and the *Stop* states. The *Start* state indicates the entry point of the state machine, while the *Stop* state indicates the exit point. As states can be nested (see the containment edge-loop in the metamodel), the start and end states may also be used as the entry/exit point of a sub-state machine. States can be connected with the help of *Transition* edges. A *Transition* edge may trigger the reception of a specific message from the stream: the type of the expected message is defined by the *MessageTypeIn* attribute of the edge, which references an already modeled message. If several outgoing transition edges are connected to the same state, then the transition whose triggered message first arrives will be chosen. If a transition is chosen, the state pointed by its right end will be the next active state. When activating a state, the instruction described in its *Operation* attribute is executed.

An important issue in every message-based protocol is the phase where we have to decide which message has exactly arrived. We have implemented an advanced message handling algorithm especially for the presented *MessageStructure* DSL: in each state of the protocol we have a set of messages which can arrive in the state. Each message knows how many bytes it can consume to process its current (still not read) field. If the size of the current field cannot be determined (in case of a *SeparatedField* or a *FixedLengthField* with variable size) then this message can process only one byte. The key point is that we can increase the efficiency of message parsing, because if we have a set of possible in a state, the minimum amount of bytes to read can be determined. Thus, we do not read the stream by single bytes. Based on the bytes received, we can filter the set of possible messages by the fields with constant values. After a reasonable amount of incoming bytes we can restrict the number of possible messages to one. If neither of the existing transitions is compatible with the data read, we have two possibilities to handle this situation. Either we assume that a *Protocol error* has occurred, and handle the error with a special *ErrorOccured* edge,) or – if no *ErrorOccurred* edges are present and the current state is nested – we let the container state handle the message. We may also attach preconditions to the transitions so that the transition is selected only if the appropriate message arrives, and the condition (*Condition* attribute of the edge) is evaluated to true. In addition, there are two special types of transitions: *non-reading* and *fallback*. A transition is non-reading if it does not trigger any type of incoming message. These transitions are checked only for their *Condition* attribute before choosing them. Therefore they always have priority over the *reading* transitions. *Fallback* transitions (their condition attribute is set to *[fallback]*) are chosen when neither of the other transitions in a state can be selected. This feature is quite analogous to the handling of protocol errors, however, fallback edges are not to handle errors, but it is regular behavior. Recall that a transition may also send a message through the stream. The type of the message sent is defined by the

MessageTypeOut attribute, and the initial value of the fields of the message can be set through the *MessageOutParameter* attributes of the edge.

As already mentioned, the *ErrorOccured* edge is used to handle the errors (either Protocol or I/O) of the network layer. I/O errors occur, when the reading or writing to the stream fails. I/O errors can be handled with *ErrorOccured* edges whose *Type* attribute is set to *I/O*. If none of the outgoing transitions are applicable in a state, and an *ErrorOccured* edge is present whose *Type* is set to *Protocol*, then – regardless of possible container states – we treat this situation as a protocol error. An *ErrorOccured* edge always points to an *Error* state. Error states are special in the sense that a callback method is assigned to each of them on the callback interface. In case of I/O errors, the callback method receives the last exception as well. A special form of *Error* nodes is the *Abort* node, which immediately finishes the execution of the network layer.

The state machine may be customized with *Variables*. Each variable has a name, a type and a default value. Variables can be considered as global parameters, which can be accessed by all states and edges. Variables can be used in the conditions for transitions, during the instantiation of a new message and also in the *Operation* attribute of states. The code generator generates a member variable for each *Variable* node in the model, along with their getter and setter methods. The member variables are initialized with the content of the *DefaultValue* attribute of the corresponding model element.

Recall that a callback method is generated on the callback interface for each error node. Furthermore, a method is generated for each stop state as well. However, one may extend the callback interface arbitrarily, and call its methods from any point of the state machine. For this purpose we have invented the *Callback* node, which symbolizes on method stub on the interface. The generated method can be parameterized with the help of the *Parameter* attribute of that node. Callback methods on the interface can be invoked in two ways: either through a *DoCallback* state, or with the transition edges, as a method invoke can be assigned to each transition. The parameters of the method invoke can be set with the *CallbackParameter* attributes in both cases.

Network connection handling is also modeled with *Connection* edges. Depending on their *Type* property, such an edge either opens or closes the network connection. The target host for the connection is specified by the *Host* attribute. However, the parameters of the connection (connection type, direction, timeout handling etc.) can be customized during code generation.

Figure 3 illustrates the model we have created for the BitTorrent protocol. The yellow boxes represent the global variables of the state machine: (*i) peerAddress* – the address of the peer we are connected to, (ii) *torrentInfoHash* – the hash of the downloaded torrent, (iii) *peerId* – the unique identifier of the connected peer and (iv) *ownPeerId*- our own identifier. The blue boxes with a small yellow lightning denote the callback methods created on the callback interface.

The protocol works as follows. After the start state (1), we call the *Initialize* callback (2) to instruct the framework to perform initialization steps. Then the protocol tries to connect to the target host (the *Connect* edge is parameterized with the *peerAddress* variable). If the connection succeeds, we get to the *Connected* state (3), otherwise an I/O error occurs (4). On error, we perform an *IncreaseErrorCounter* callback, and disconnect the stream. Moving from (3) to (5) a *MessageHandshake* (Figure 2) message is sent to the remote peer. Edges (6) and (7) trigger the answer-*MessageHandshake* message, and check if the parameters of the answer are valid. On an invalid handshake answer, either the *IncreaseErrorCounter* or the *DeletePeer* state will be active, and the communication is closed with the current peer. Edge (8) is a fallback edge meaning that edge (8) is chosen if neither of error transitions (6-7) can be selected. The state *PWConnected* can be considered the default state of the protocol: almost any type of messages can be received at this state (that is why there are so many loop edges around it), and each message arrival performs the appropriate callback invocation. As you can see in Figure 3, the edge parameters (and also other model parameters) can be changed with the help of smart tags. They appear when the mouse is hovered over an item. State *PwConnected* can be left only if a

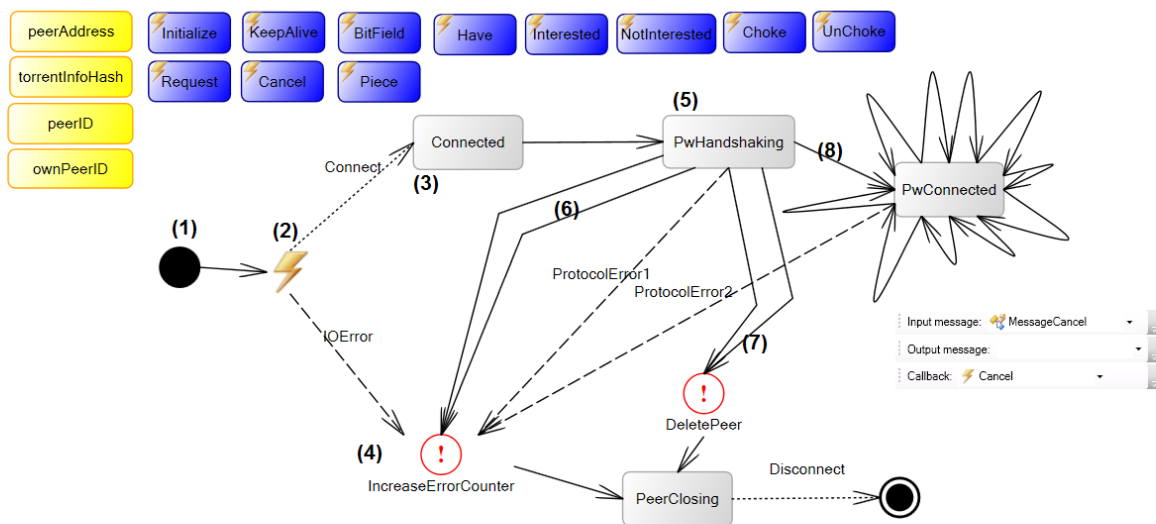


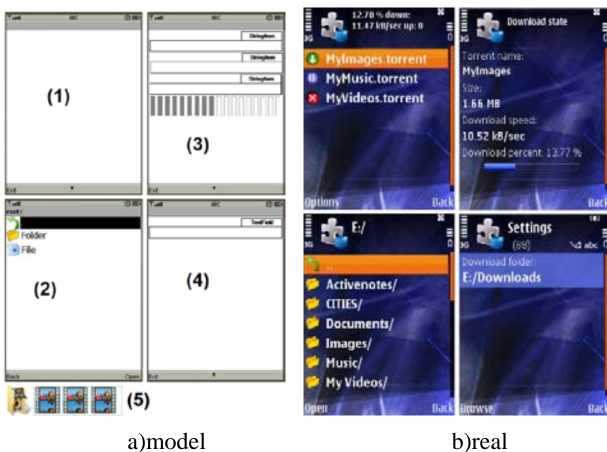
Figure 3 BitTorrent client protocol model

protocol error occurs, or the business logic over the network layer changes the current state.

2.2 Mobile DSL for User Interface Development

Having generated code from the network model and integrated it with the *MobTorrent framework*, we started to work on the user interface of our new mobile BitTorrent client. With UI DSMLs, we can model the static structure of user interfaces, and generate the platform-specific source code according to the models. The UI DSML also has a metamodel, but its detailed explanation is irrelevant. Instead, we are focusing on the models we have created for the BitTorrent application. VMTS supports [4] all the *Screens*, *Commands* and *Controls* available in Java ME both on the modeling and the generator level. Besides the general elements we also provide additional controls which are not part of the Java ME API, but are used in numerous scenarios, such as the *FileSelectDialog*. In P2P applications we usually download multiple contents at the same time and these downloads are displayed in a list where the icon of the list item represents the status (downloading, finished, error, etc.) of the download. With the help of an *ImageList* we can easily access image resources and use them in other components, for example in a List.

In Figure 4, the four screens of the application can be seen both at modeling time (a), and when executing the application on a real hardware (b). Screen (1) is used to present the torrents being processed (*TorrentList*). It is modeled with a simple *JList* item which is replaced with a class derived from Java ME *List* during code generation. Screen (2) is the *FileSelectDialog* itself, with which one can browse for a torrent file to be processed. Screen (3) is used to show the download state of the selected torrent. Screen (3) is built from a *JForm* item, which contains three *StringItems* for presenting the name of the torrent file, the size of the downloaded data, and the actual transfer rate. A *JGauge* element represents a progress bar which shows the progress of the download. Finally, screen (4) is used to modify the application settings such as the download path. It is also based on a *JForm* element, which contains a *JTextField* item. (*JTextField* corresponds to the *TextField* Java ME class).



a)model b)real

Figure 4 UI model of the mobile BitTorent client in VMTS

With the VMTS UI DSML, we can also set the commands (menus) for the screens. The *TorrenList* contains commands for torrent handling such as add torrent file, start download, pause download, and commands responsible for navigating to another screen like *Settings* or *Download state*. Thus, we can also describe the high-level UI logic. The model also contains an *ImageList* (5) with three icons. This list represents the icon set used by Screen (1). Finally, after modeling and generating the network layer and the user interface, one task remains: integrating the generated components with the *MobTorrent* framework. The integration is not supported with visual techniques in the current release of VMTS, the glue-code has to be written manually.

In order to integrate the UI with the *MobTorrent framework* we have applied the *Observer design pattern.* The framework provides an interface which we have to implement in the UI code. This interface contains functions that are called from the framework when the status of the download changes, such as download speed changed, download progress increased. When we initialize the framework we have to set which object implements the observer interface in the UI. By using this observer the framework can notify the UI if something changes and the relevant information can be displayed on the screen of the mobile phone easily.

3. Conclusions

So far we have shown how mobile P2P development can benefit from DSML technology. We found well-separated functionality groups, and supported them by DSMLs and code generators. Table 1 depicts the development times with manual coding and with DSMLs taking one developer into account who had previous experience of this sort of application.

Functionallity	Time with manual coding	Time with DSL
User interface	5 days	2 day
Peer network connection	8 days	1 day
Peer-wire protocol	6 days	1 day
Message handling	10 days	2 days

Table 1. Development time with and without DSMLs

Additionally, there were functions, which we did not support with DSMLs. These required the following amount of time:

- File and database handling: 8 days
- BitTorrent specific functions: 13 days
- Tracker communication: 5 days
- Download for other clients: 8 days

The DSML infrastructure, i.e. the languages and the generators, is developed by an engineer with extensive DSML and tool experience. The time spent per person is the following:

- *MessageStructure* and *MessageProcessor* DSMLs: 4 days
- *MessageStructure* and *MessageProcessor* generators: 5 days
- UI DSML: 9 days
- UI generator: 10 days

So the development effort for the functions supported by DSMLs is as follows:

- With DSMLs: 6 days
- Without DSMLs: 29 days

The development time without the time for the DSML development:

- With DSMLs: 40 days
- Without DSMLs: 63 days

Including the DSML infrastructure development:

- With DSMLs: 68 days
- Without DSMLs: 63 days

These numbers shed a light on the fact that DSML technology is a generative technique: a generator is much harder to develop than the generated code once. Therefore, the more times you run the generator, the more the DSML approach pays off. From the second time on, the DSML and generator development does not appear as an additional cost. Our intention was to support UI changes, protocol changes and updates in the forthcoming version of a P2P application. That is why we support these functions and not others. As a matter of fact, our figures look better: we inherited the UI DSML from another project targeting cross platform UI development. Thus, we can subtract it from the total, and we have 47 days for the *MobTorrent* project.

As long as only the models need to be modified, DSMLs increase the maintainability. If the generator must also be modified, the necessary effort can arbitrarily increase. We expect that we need to modify the models for the next versions because of the generality of the DSMLs, and subtle generator modifications if the Java ME UI changes. These DSMLs can be reused for any Java ME mobile development where UI or network support is required, but the approach is not limited to the Java ME platform, since it can be extended to other platforms by modifying the code generators. The proposed case study can be used as well in other solutions where BitTorrent technology is used for content distribution. Since our department is involved in developing such applications on a regular basis, we have a rational expectation to have the return of our investment in the DSMLs and the supporting generators as it happened in the case of the UI models.

We tested the generated code, and decided to include it in the first release of *MobTorrent*. Thus, *MobTorrent* is expected to be publicly available in January 2010 on its website, as the first mobile P2P client developed with the extensive help of DSMLs.

ACKNOWLEDGMENTS

The found of “Mobile Innovation Centre” has supported, in part, the activities described in this paper. Infragistics has supported, in part the activities described in this paper.

4. REFERENCES

[1] P. Ekler, J. K. Nurminen, A. J. Kiss “Experiences of implementing BitTorrent on Java ME platform”, CCNC’08. 1st IEEE International Peer-to-Peer for Handheld Devices Workshop, pp. 1154-1158, 2008, USA

[2] Visual Modeling and Transformation System Website: http://vmts.aut.bme.hu

[3] BitTorrent specification, Oct. 13, 2008. [Online]. http://wiki.theory.org/BitTorrentSpecification

[4] I. Madari, L. Lengyel, T. Levendovszky “Modeling the User Interface of Mobile Devices with DSLs”, Proc. of the Computational Intelligence and Informatics 8th International Symposium of Hungarian Researchers, pp. 583-589, 2007, Hungary