VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

CHARAKTERIZACE KÓDU PRO AUTOMATICKÉ GENEROVÁNÍ UŽIVATELSKÉHO ROZHRANÍ
CODE CHARACTERIZATION FOR AUTOMATED USER INTERFACE CREATION

DOKTORSKÁ PRÁCE
DOCTORAL THESIS

AUTOR PRÁCE   ING. JAROSLAV KADLEC
AUTHOR

VEDOUCÍ PRÁCE   DOC. DR. ING. PAVEL ZEMČÍK
SUPERVISOR

BRNO 2010

Declaration

I declare that this dissertation thesis is my original work and that I have written it under supervision of Doc.Dr.Ing. Pavel Zemčík. All sources and literature that I have used during elaboration of the thesis are correctly cited with complete reference to the corresponding sources.

Acknowledgments

Abstract

This work presents novel approach to automation of user interface creation. A taxonomy based on data characterization was adopted and new taxonomy for code characterization is proposed. Taxonomy points most important aspects of data and code so that it can be used in a process of automated user interface creation. The taxonomy is platform independent and can be stored as a metadata in the application file or in a separate external file. A process of automated user interface creation based on the taxonomy is proposed and individual parts of the process are described in more detail. Presented taxonomy and process of user interface generation are demonstrated on examples.

Key Words

user interface, taxonomy, data characterization, code characterization, automation, automated user interface creation, evaluation, layout design

Contents

1 Introduction

During last few decades, humankind discovered lots of fascinating objects and developed millions of inventions – so much that it is much more that humankind was able to invent during its past evolution. New things created from these inventions and ideas are filling human lives and are getting more and more complex and complicated, and sometimes it gets so complex and complicated that it is almost impossible to learn how all these great and fascinating things work. One of the most fascinating objects that become common in everyday lives are – computers. Computers were, at first, huge machines covering several floors of building, used by scientist to help them compute very complex and difficult tasks. As the time passed, computers got faster, smaller and finally they are on desks in offices, in cell phones, TVs, watches and, in fact, they are in almost every manmade device that is powered by batteries or electricity.

Computers are very sophisticated devices that can be used for various tasks. However sophisticated devices require also sophisticated control, which is not trivial. For this reason, parts of computer, such as display, mouse or keyboard, have been designed to ease the communication between user and his computer. Because communication between computer and user is two-way, called interaction, part of the computer that is designed for such interaction is called user interface. This means it is not only hardware itself, but also a part of software – windows and buttons, texts on the display, etc. thanks to which the user is able to specify his requests for the task and computer can help him finish it. Without proper user interface, user would not be able of any interaction with computer and thus unable to finish any requested task.

That is why a main goal of this work was to find aspects that can affect design of user interface on computer devices. Because of large number of devices with varying user interface designs, this work focuses on aspects ranging from device's physical parameters to human context and preferences. This thesis explores needs for automated user interface design, why it can be useful and what are main advantages and disadvantages of such design. The results of this thesis have implications for future development of automated user interface design.

Chapter 2 – User interface architectures – describes history of user interfaces, basics of human

perception and how it can be used in human-computer interaction. Chapter 3 – User interface development – describes often used user interface creation tools and describes some principles of user interface design – choosing and using basic user interface elements and evaluation methods of user interface usability. Chapter 4 – User interface design automation – discusses automation of user interface creation by defining requirements needed to automatically create user interface and approaches used to generate them. Chapter 5 – Annotation and description – describes approaches based on user interface or data description and suggests novel approach featuring data and code description. Chapter 6 – Characterization - describes new approach based on the data and code characterization. All main parts of the proposed taxonomy are discussed here. Chapter 7 – Generating a user interface from characterized code – describes process that is using the proposed taxonomy to automatically generate user interface and presents examples of how each part of the process. Chapter 8 – Conclusion – points out most important aspects of the work and suggests possible future research.

2 User interface architectures

Evolution of technologies in computer systems in last decade has shown that device without properly designed and understandable user interface is rejected by the user community. Despite of the fact that device can implement hundreds of features that competitors are unable to implement to theirs devices. An example can be the Apple iPod Shuffle [1] – portable digital audio player with embedded memory card and limited functions – this player has become one of the most successfully sold portable digital audio players. Although there are digital audio players with much more functions and larger memory – simple design and ease of use of iPod Shuffle has become a major reason why people use this device and not the other [2][3]. iPod Shuffle has very simple user interface with only few buttons on its body that care about basic functions such as playing, stopping and selecting previous or next digital audio file. Even without any display that could show colorful information we can speak about user interface. It can be said that user interface is a part of device that is used for human to read status of the device and control its functions, and part of device that is used by the device to report its status and offer its functions to user [4]. An example of simple user interface can be a hammer (see Figure 2.1). Hammer is a device that does not contain any electrical part but consists of two main parts – a wooden helve which is used to control the iron head that is on the end of the helve.

Figure 2.1: Hammer [5]

The user interface of a hammer is very simple and most users can use it quite effectively to

deliver the blows to the intended target. From this example it is clear that, although it might not seem so, world is full of various devices that have certain user interface thanks which it is possible to control its behavior. Following chapters summarize current state or the art in the user interfaces relevant for this work.

2.1 User Interface Design

A hammer is one of the simplest devices used by humans. It can be said that a hammer is very successful product because it is being used by people all over the world. It is also possible to say that its user interface has been designed so well that people like it and it also might be one of the reasons why it's still being used. It is also possible to ask a question: "What is good user interface?" There are millions of devices used among the people and the definition of good user interface is still unknown. Generally, good user interface is such that time necessary to learn how to use the device is short and that using the device is somehow comfortable [6]. However it is quite difficult to design good user interface because people have different requests on device controls. For instance, when thinking about a hammer – there should be hammer with big helve for the people with big hands, small for the people with small hands that everybody is able to hold the hammer tight in his hand. When a person with a small hand will work with a hammer with a huge helve, it will most probably say that the hammer is not really good as it does not fit well in his hand. This means that user interface is not good at all for the person. On the contrary, a person with a large hand having hammer with a huge helve will be satisfied with the hammer and in this case the user interface will be good for the person. It is obvious that there is no phrase like: "This user interface is greater than 70, so it is good". There is no simple value that would distinguish between good and bad user interface. This can be said about diesel motors – the greater is power of the motor, the better the motor is – but not about the user interfaces.

User interface design is very difficult task and not always leads to successful design. User interface designers are using rules [6][7] that help them in decisions how a user interface should work. On the other hand, situation can be complicated by artists, who create unique and visually stunning interfaces which are sometimes hardly usable. That is why designers of user interface split design into several steps that are cycled as long as users, testing the interface, are satisfied [6][7][8]. For instance, user interface of a hypothetic device will differ for people with various knowledge of its application. Professionals with deep knowledge of its application will require possibility to set each parameter separately (to have complete

control over behavior of the device), however common user requests easy handling and any complicated settings of device before its use will result in device's uselessness. It is obvious that if user interface design is difficult for a human designer, it will be almost impossible for artificial intelligence to create high quality user interface without knowledge of user context and his requirements.

User interfaces, designed for computer applications, are usually built in integrated environments with tools and built-in functions that help designers to simplify task of user interface design [9]. Designer can create forms and place interface elements such as buttons and textboxes and assign them various actions (e.g. on button click save document). Design of user interface for computer application is even more difficult than user interface design of a real world device. It is because computer application is somewhat virtual, abstract, and sometimes much more complicated than a real world device's interface. Creation of user interfaces for computer applications is also complicated by fact, that development of application algorithms take some time and that is why not all requirements on user input are known at design time. Such complications lead in iterative user interface design, sometimes leading in major user interface redesign [6][7][8]. This is the situation where automated user interface design should simplify task of development application algorithms. Artificial intelligence user interface designer could create simple, yet usable, user interface that could help develop and test algorithms during development stage and later, in the final stage, a high quality interface could be created based on experiences with user interface created by artificial intelligence. This all could simplify task of frequent user interface changes every time an algorithm changes and thus speed up application development. That is why it is important to support automated user interface designer by additional information that would help with user interface design and create usable interface based on user's preferences.

Automatically created user interfaces could be also used to simplify development of cross-platform applications [10][11]. Application would only contain necessary algorithms to provide some functionality, thus reducing size of application, and artificial intelligence user interface designer would create user interface that suits best to computer or virtually any other device where application would be ran on. Devices such as PDA's, mobile phones, and tablets have different user input capabilities and that is why it is impossible to create single user interface that would fit all of these various devices. It would change its layout for devices with small resolution display, change sizes of control elements to fit touch-screens that are

controlled by human fingers, etc. To understand possibilities of control of user interface, it is very important to understand human perception, its limitations and benefits.

2.2 Human Perception

Children at basic school are usually taught that there are five senses – vision, hearing, taste, smell, and touch. Based on definition of sense it is generally agreed, that there are at least seven senses (hearing and balance, vision, touch and pain, and smell and taste) [12]. Other definitions of sense imply from nine to twenty senses [13] which are not containing senses observed in other organisms. Senses are only way how human can get information about his environment and also get information from various devices. Some of the senses are used very often (balance, vision, hearing, and touch) and some less frequently (pain, smell, and taste). Computers of these days most usually use for human-computer interaction displays to visually show information – employ human vision [7], sounds to alert important events – employ human hearing [7], and keyboard and positioning devices such as mouse or pen for data input – employ human touch [7]. There are also known other input output possibilities such as MindDrive [14] using brain waves, speech recognition [15] using speech, or Braille displays using touch [16]. All these devices are not widely used because their use is most usually limited for certain applications. Understanding to each of basic sensors of human body is important for future user interface design – sensors have different capabilities and features, which require different interaction models.

2.2.1 *Vision*

Vision is one of the most important human senses [17]. Most information acquired from surrounding area for most healthy people is thanks to vision. Organ of vision is eye. Simplest eyes are able to detect light or dark areas, while complex eyes can distinguish shapes, and colors. Human, as well as most of other animals, has two eyes which field of view overlap to allow better depth perception. Structure of human eye is described on Figure 2.2: Human eye. Function of the eye is to transform light into neural signals that can be processed by brain. Image (light) passes through cornea and pupil onto lens. Pupil is adjusted using iris to maintain constant level of light entering the eye. Too much light entering in could damage the eye, too little light causes bad vision. Lens and cornea is used to focus image onto retina which contains rods, cone cells and associated neurons to transform image into neural signals which are processed by brain. The fovea, placed directly behind lens, consists of mostly densely placed cone cells allowing highly detailed central vision. Its requirement for high

11

light intensity does cause problems when looking at dim objects because the light from these objects does not have enough intensity to stimulate cone cells. Rods, on the other hand, do not require high light intensities and allow vision in the dark. Rods and cone cells allow vision in both light and dark places.
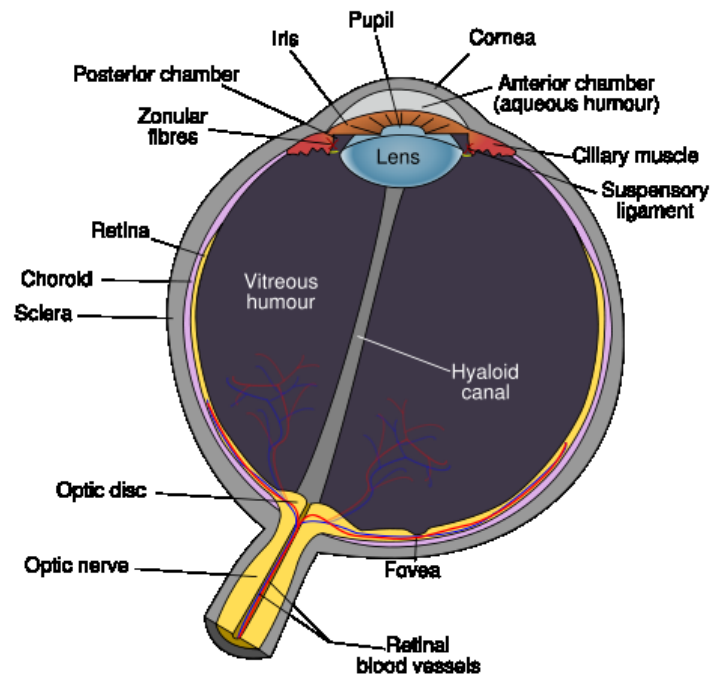

Figure 2.2: Human eye [18]

Human eye is great source of information for the brain. Retinal bandwidth is over 10 millions of bits per second [19]. It is almost 70% of all information that comes from the eyes to the brain. All information is then processed in the brain. The brain does heavy processing on acquired data and tends to find solution in known issues he detects. This tendency sometimes lead in various misinterpretations thanks to which human can think he sees something different that there really is. As an example, The Hermann Grid illusion [20] can be used, where all intersections appear to be grey although they are white.

Most of the user interfaces display information that can be visualized because human can get most of the information at a time using vision. This, however, requires that eye has focus on information it wants to collect which means it cannot focus on other things. This situation can be dangerous in various situations (e.g. driving a car) but in most cases (work in office) it is

fine. For this reason, vision is number one for computer-human interaction.

2.2.2 *Hearing*

Another important human sense is hearing. Organ for hearing is ear. Ear is capable of hearing sounds of various frequencies – from 20 Hz to 22 kHz, and detect small differences in sound intensity (loudness) and sound frequency (pitch) [21]. Human has two ears placed on opposite side of the head, allowing location of sound direction and distance. Structure of human ear is described on Figure 2.3. The outer part of ear – pinna – collects surrounding sound, which is amplified in auditory canal. Amplified sound pressure is transferred to inner ear using tympanic membrane, malleus, incus and stapes, which presses onto fluid filled ducts of the cochlea. The fluid inside moves hair cells (mechanoreceptors) that emit sound information to brain. One part of inner ear is also dedicated for sensing balance and position [21].
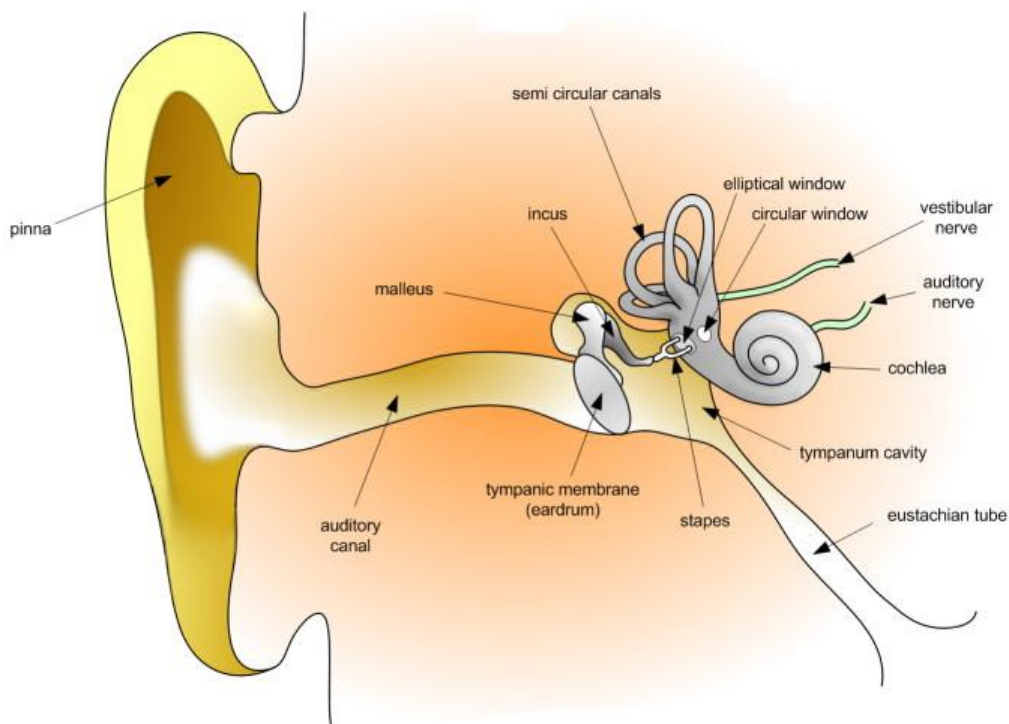
Figure 2.3: Human ear [22]

Although hearing is in common life second most used information source (people exchange information when talking), it is used in computer-human interaction rarely [7]. Main problem of hearing is that its input is serial and when compared to vision, equivalent information taken using hearing would take much longer than using vision. Hearing is most usually used to report alerts or requested information in specific applications where user has strong visual

13

attention, or is blind, and cannot get requested information visually [7][23].

2.2.3 *Touch*

Touch is the sense of pressure perception. This sense is most important for human-computer interaction [24]. Although it might not seem so, blind or deaf people can interact with their surrounding environment and have no problem holding any object. People that lose pressure perception (e.g. by an accident) cannot determine if they touched an object, or hold an object, or how heavy an object is [24]. All these actions must be visually controlled and request extreme caution, because of danger of crushing or dropping held objects.

Touch is important for human input and it can be used to detect computer-human interaction. Interaction performing high-fidelity sensations to feel the grain, texture, or material against human skin or nervous system is called tactile feedback [25]. Interaction applying force to parts of human body or entire body is called force feedback [26]. Interaction performing low-fidelity vibrations or shakes is called rumble feedback [27].

Human-computer interaction is, in most cases, limited to data input using keyboard, mouse, or other pointing device. People with limited touch sense cannot control computers easily and effectively [24] and in certain situations may require special user interfaces (e.g. based on voice commands), which make the control less problematic. On the other hand, feedback stimulating touch senses can help in many applications requiring knowledge of objects mass or material and increase the immersion in entertainment applications. Also, besides hearing, touch is another way of communication with devices for blind people.

2.2.4 *Other senses*

Other senses such as pain, smell and taste are important in everyday life, but computer interaction does not use these senses much [28]. Using taste is rather problematic – user interface would be limited for individual person because of hygienic reasons, and also amount of information that taste receptors can carry is not wide enough to be widely used. Pain brings unpleasant feelings for most of people and that is also reason why it is not used for computer-human interaction. Smell is sometimes used in entertainment industry to increase immersion in 3D cinemas [29]. Balance is, as well as smell, used in entertainment industry where balance sensing can increase immersion in 3D cinemas or simulations (aircraft flight, roller coaster movie etc.) [30].

2.2.5 *Memory*

Although memory is not a sense, it plays its role in the perception. Every signal from receptors is processed by brain and stored in memory. Processing is affected by memory contents. Brain, when processing signals, is using stored information (also called experience) to obtain best results. For this purposes, there are various types of memory. Sensor memory [31] contains information that is present on sensors. Short-term memory [31] contains temporal data. It is very quick and its capacity is limited. Information that is important is after some period of time moved to long-term memory. Long-term memory [31] has large capacity and can store information for very long time. Accessing long-term memory is not as quick as in case of short-term memory.

Information is stored in memory in packages that are called chunks [31]. Chunk can contain single letter, whole word, sentence or image – this all is based on experiences of individual human. When designing user interface it is important not to overload user with information, because short-term memory can contain only limited number of chunks [32]. Information should be divided to logical groups which can user easily locate and remember.

2.3 Human Actions

To create user interfaces, it is important to understand possibilities of information output – an input for user interface of a computer. Actions can be divided into several groups – motion, speech, and mind.

2.3.1 *Motion*

Motion is one of the typical actions that a human can do. Human can move various parts of his body. These moves can be detected and translated to language that is understandable by computer. Motion detection can be done mechanically or by software [33]. Mechanical detection requests user to directly interact with certain part of human interface device. Software detection is using computer vision [33] to recognize movement and its parameters. Mechanical detection is more accurate but limits free movement, while software detection has no movement restrictions and higher error rate of wrong motion detection. One of typical motion detector is computer keyboard [4] which is translating finger moves to appropriate letters so that it is possible to write text on a computer. Another motion detector is computer mouse [4]. Mouse tracks hand movement in two directions and computer can use movement

detection for navigation and other purposes. Similar is gaze movement [34] tracking that is used especially for movement-limited people that cannot use hands for pointing. Instead, eye-gaze movement is tracked and used for pointing [34]. Another movement application is gesture recognition [35]. Hand or body gestures are recognized and its meaning is interpreted. Such application can be useful in situations, when human cannot control computer directly but can interpret gestures or human body motion and translate it into commands.

2.3.2 *Speech and Sound*

Speech is another way of communication with computer. User can give speech commands to computer, which recognizes command and executes it [36]. Speech recognition [37] is difficult because speech differs not only by spoken language but even among people speaking same language. Speech recognition is complicated by surrounding noise and can cause a possible security risk because speech commands can be ordered from any source of audio signal. That is why speech commands are used only in several applications that include applications for blind people.

2.3.3 *"Mind Control"*

Under the "mind-control" is hidden detection of electrical and chemical changes in human body. This approach is not very common. Devices that detect only changes of electricity on skin [14] are not sophisticated enough to supply information for complex computer interaction and are most usually used for entertainment purposes only where are its capabilities sufficient. Devices combining multiple bio-signals [38] have better results but are still considered for entertainment level. Devices that are directly connected to neural system using brain-computer interface are complicated, and still in the phase of development, but allow control of complex object such as robotic arm, lights, TV [39].

It is important to understand that actions can be combined to produce very effective communication. User input can consist of keyboard input for text data and mouse input for two-dimensional data. Other devices can be used to increase communication performance. However various devices differ with its functionality and cost, which is most usually important for massive expansion. Devices detecting gaze movement, body posture and gesture recognition are expensive and that is one of the reasons why it is used only on specialized workplaces.

2.4 Human Interface Devices

Human Interface Devices (HID) are used for data exchange between human and device, typically computer [4]. Interface devices can be divided to two groups – input and output devices [4]. Input devices are used for entering data to device [4] while output devices are used to present data from device [4]. Important feature of both groups of devices is that both input and output data are in a format that is understandable by user (human) easily.

2.4.1 *Input Devices*

One of the simplest and widely used electrical input devices was a telegraph pad designed for generating pulses that represented Morse code. User was pressing a key that generated pulses which, based on the combination of press length, represented appropriate code. Input devices for computers work in a similar manner – user is operating input device, which is transforming his actions into electrical signals that are interpreted by a computer. How is signal interpreted depends on what input device is used. There are several groups of distinct input devices that are currently used with computers. Following paragraphs describe groups of input devices that are relevant for this work.

Pointing Devices

Requirements on pointing devices were caused by growing popularity of graphical user interfaces which required new, more effective ways, of interaction. Pointing device is a hardware allowing user to input spatial data to a computer [7]. Movement of the device is usually reflected on a computer display by movements of a pointer (typically cursor) and other visual changes. Most of the actions in current user environments (which are in most cases graphical) are controlled by pointer because it is more effective than most other common input devices. For example moving and clicking by pointing device is much faster than writing command for opening desired document, selecting object, copying or moving objects etc. Selection of objects of interest is one of the most often performed actions and is much simpler with pointing device than using textual or voice commands.

Well known and widely used pointing device is mouse [7][40] (see Figure 2.4). User is moving hand-held mouse across the surface and thus specifies position of the cursor on the screen. Mouse movement is detected using small ball [40] (called ball mouse), or optic sensor with high resolution [40], placed in the bottom part of the mouse. Mouse can be also equipped with several buttons [40] which can be pressed and thus call various actions. Typically, there

are two buttons for primary and secondary actions and roller for faster document browsing. Pointing using mouse is very effective and easy but requires certain skill.

Figure 2.4: Logitech Performance Mouse MX [41]

Novice users move mouse slowly and their accuracy when pointing is small (but increases very fast when mouse is used regularly). Because mouse is used with almost every desktop computer, most of the users are familiar with the mouse and that is why mouse has become a standard pointing device. However mouse requires quite great amount of flat surface to move on. If there is a little space, user might be unable to point at all objects of his interest. If surface is not flat, moving mouse is difficult and sensor detects false moves, which results in cursor hopping.

Besides mouse there are several interesting pointing devices that are not that widely used – trackball [7], eye-tracker [34], data glove [7], and pen [7]. Trackball consists of ball which direction of rotation is transformed to a pointer movement. Eye-trackers detect eye gaze movement which is transformed to pointer position. Data glove transforms hand gestures and movement to pointer movement. Pen is a device that is used to direct pointing on computer display or special desk (called tablet). Pen detects position on tablet or screen which is then reflected to cursor. Trackballs or pens are specifically designed for portable devices and that is why they require only small space for its operation. Eye-trackers and data gloves are quite

18

expensive pieces of hardware and are used in special cases such as virtual or augmented reality applications, applications for handicapped or disabled people etc.

Keyboard Devices

Computer keyboard is a device modeled after typewriter keyboard [42]. Computer keyboard, as well as typewriter keyboard, is designed for input of text and to control operation of computer. First electromechanical keyboards were used to input data onto tapes (for feeding the computer data) [43]. Design of most commonly used keyboards is QWERTY (it has been extended and contains several new keys to support multimedia and other functions) despite of its low efficiency [44]. Efficiency of alphanumeric writing on QWERTY keyboard is about 70 words per minute [45](also called wpm), and when compared to 34 wpm for handwriting [46] or 250-300 wpm for speaking [47], computer keyboard is most effective device with direct input (speech requires further recognition). Besides traditional key layout of QWERTY keyboard there are other key layouts that are designed with means of higher writing efficiency. Dvorak keyboard [44] has different key layout to increase writing efficiency (word record of Barbara Blackburn from 2005 is 212 wpm [48]). Maltron keyboards [49] have completely different, ergonomic design (curved surface to reduce finger traveling) to reduce wrist pain and other injuries, invented in 1970s [49].

Figure 2.5: Microsoft Natural Ergonomic Keyboard 7000 with ergonomic 3D design [50]

Although using keyboards is very easy (most users can type using one, two, or more fingers), fast typing with all ten fingers is very difficult. Learning how to type takes some time, during

which user learns layout of the keyboard. This time is followed by several months of increasing typing speed and reducing number of mistakes. It is obvious that users using keyboard occasionally cannot use keyboard effectively and thus their data input frequency is very slow [51]. Because data input from keyboard is the simplest when compared to other devices (minimal hardware requirements, key press easily interpretable), and most users can easily recognize keyboards purpose, it is still considered to be number one input device.

Keyboard requires lots of space because keys must have minimal size and spacing to fit human fingers. This is the reason why keyboards with reduced size and number of keys were invented for small or portable devices. One of the most frequently used layouts is matrix of four rows and three columns, where each key represents three letters. When user wants to write certain letter, he must find appropriate key and press the key exactly as many times, as is position of selected letter on the key. Writing on such keyboard using this technique (called multi-tap [51]) is not very effective [51] and for this purpose were proposed various predictive technologies such as T9 (by Tegic Communication) [51] or iTap (by Motorola) [51] which speeds up writing of common words but reduces writing speed in case of special words or abbreviations.

Gaming Devices

Gaming devices are specific devices that have been developed to improve immersion from game or simulation. An example of gaming device can be joystick [4], which was originally designed to control aircraft's ailerons and elevators (see Figure 2.6).

Figure 2.6: Thrustmaster Hotas Cougar [52]

20

Because joystick allows precise control in several axes, it is not used only in airplanes and flight simulators but also in other applications where precise control is required (e.g. elevators, cranes). Gaming devices have been, however, developed for entertainment purposes and its design follows requirements of gamers. Gaming devices have reasonable amount of buttons that cover important actions and axis controls to enable movement or view. Gaming devices can be used to control various devices which handling can be reduced to fit gaming device such as gamepad [4]. Car can be controlled by steering wheel and several buttons to start engines and switch gear etc. Gaming devices are not widely used for common tasks but their functionality can be used to control general devices.

Video Input Devices

Video input captures image from digital camera, web camera (see Figure 2.7), and scanner and other, video capturing enabled, devices.

Figure 2.7: Logitech Webcam C300 web camera [53]

Video input can be used to capture and transmit video in network conferences [4] so that everybody can see each other, to capture or scan images or documents for their further editing, to recognize captured video and its contents. Recognizing captured content however requires image processing (requiring high performance hardware) and this is also reason why video input devices play role only in specific applications. These applications cover virtual conferences, where person image is transferred to every participant so that participants can see and hear each other, identification to recognize certain person [54] or object in the scene, gesture recognition [4][54] and others. Gesture recognition can be used in entertainment (in

21

games where player controls game using special moves) [55], sport (detect wrong movements decreasing performance) [56] and industry. Although video input devices play small role in user interface in these days, we can expect their increasing role in the future [57].

Audio Input Devices

Audio input devices, such as microphone (see. Figure 2.8), capture surrounding sounds. Captured audio usually contains noises and background sounds. To eliminate unwanted noise or background sounds, microphones with various directionality patters were developed [58].

Figure 2.8: Audio-Technica AT8015a shotgun microphone [59]

Audio input devices are most usually used (in the context of user interfaces) for speech capturing but it can be used to capture any audio signal. Captured speech is filtered and recognized by specialized software or hardware, which is transforming speech commands to a computer commands [60]. Speech recognition requires lots of computational power and that is why it is not suitable for every device, which does not have enough power for speech recognition. Speech recognition can be implemented from simple keyword spotting [60] to continuous speech recognition [60] depending on devices capabilities and user interface requirements. Speech recognition is useful in situations where user cannot use his hands to control other input devices (keyboard, mouse etc.) or is handicapped or disabled.

2.4.2 *Output Devices*

An output device is a piece of hardware that is used to present information to the user. There are several ways how output devices present information. In these days, it is usuallz information visualization, which is for most of the people the most effective way. Another way of presenting information is audio that represent information by various sounds. All presentation ways have its cons and pros and require from user different attention. Following

paragraphs describe groups of output devices that are relevant for this work.

Printers

Printers are one of the oldest output devices. First computers reported their status and computed results in printed form on paper or punched cards [61]. First printers were capable of printing text only. Latest printers can print text, graphics, or photographs with high image and color quality. However printing is relatively slow process in the means of information exchange when compared to other output devices in next paragraphs, and requires empty medium (usually paper) onto which are output data printed.

Figure 2.9: Canon PIXMA iX5000 printer [62]

Controlling computer with help of printed information is not very comfortable and efficient because user has to read current computer status that has been printed by printer, input commands and wait for the printer to print new status and results. It is clear that such interaction is not very fast and also it is not very comfortable. That is why other output techniques have been invented and printers are used for printing documents and other data, which is not used for human-computer interaction directly.

Monitors and Projectors

Visual display of text or graphics data is the most common form of output [7]. Visual output is usually performed using computer monitor [63](also called display) or projector. First monitors had no graphical capabilities and supported visualization of typed information (e.g. MDA – Monochrome Display Adapter, see Figure 2.10) [63][64].

Figure 2.10: IBM PC 5150 with keyboard and green monochrome monitor [65]

Later monitors supported besides text mode also graphical modes and were capable of drawing text and images on various resolutions [63]. Thanks to displays, it was possible to show multiple lines of text with status information, help, and other information. Reading text from monitor allows faster data searching (when compared to searching on pages of printed medium), reducing amount of space for data storage (need to store printed medium). Later, more advanced monitors were developed using high resolutions and color depths [63] (see Figure 2.11).

Figure 2.11: Monitor Samsung XL2370 [66]

Monitors, when compared to printers, allow immediate information visualization of multiple kinds of data on limited screen area. Printer can print changing information, but comparison of changing information is problematic because it is on multiple pages of paper. Monitor can change its image immediately and thus create animation. Thanks to animation and other

visualization techniques information can be faster identified and user can perform required action to finish his tasks. Because human vision can accept huge amount of information [19] (when compared to other senses), display is very good device for information output.

Figure 2.12: Devices with various screen sizes - cell phone Sony Ericsson C702 [67], netbook Asus EEE PC T91 [68], smart phone Sony Ericsson Xperia X10 [69] (from left to right respectively)

Contemporary devices use displays of various sizes (see Figure 2.12) to inform user about its state. Cell phones show battery status, contact list and much more information, which would not be possible to show without display. However, reading status from display requires user's attention. This can be problem in situation, when user must watch something else (driving). That is why using displays is not always possible. In such cases, different information output devices must be used.

Speaker & Headphone systems

Speakers [70] (see Figure 2.13, headphone systems contain small speakers) are used to transform audio data to waves that can be detected by human ears.

Figure 2.13: Logitech Speaker System Z520 [71]

Computer can use speakers to play various sounds from speech to asterisks [7]. Sounds can be used to inform user about certain events (reminders, alarms), or to notify about computer state [7]. Using speakers is great in situations where user needs to be notified about certain events but cannot watch regularly other output device (printer, monitor). Audio output is also good for blind people who cannot watch computer monitor and sound is one of few input/output possibilities. However, this type of data output is not suitable for loud environments because sound from speakers could be nullified by environment sounds.

Other output devices

Other output devices are using touch as a medium [7]. Devices using tactile feedback [25], force feedback [26], or rumble feedback [27] are generally used in entertainment or in specialized simulation applications where tactile, force, or rumble feedback has sense. For example in flight simulation, pilot needs the feedback from flight stick and other control devices. Another kind of devices using touch is Braille displays [16] which are transforming text into set of dots that can be read by people knowing Braille alphabet (see Figure 2.14). Many of other devices are in experimental stage, or used in special applications, for example for people with certain disabilities.

Figure 2.14: PAC Mate Portable Braille Display [72]

2.4.3 *Input-Output Devices*

In addition to devices, that are designed for input or output only, there is a kind of devices that are capable both input and output. It means that some parts of such device are designed for information input and other parts are designed for output. There are not many devices combining these features. It is common to extend functionality of input device to reflect certain event, such as incoming email by light indication. However, these output capabilities are too limited for advanced interaction. A typical example of advanced hybrid input-output device is touch-screen [4]. Touch-screen consists of touch sensitive layer, which can detect pressure changes, and screen layer, which displays visual information [4]. Such combination of input-output capabilities is excellent for devices which can benefit direct pointing on the

26

screen imagery (planning route in the car etc.). Such devices can also emulate almost any kind of visual interface because screen can contain almost any image that can represent certain user interface device. A typical example can be virtual keyboard. Screen can display image of keyboard and user, aware of touch-screen capabilities, can directly type on screen as he would type on a real keyboard. Touch-screen has also several cons - it can be easily scratched and thus become unusable. Another example of virtual keyboard can be laser projected keyboard [73] (see Figure 2.15). Output capability allows projection of any type of keyboard while detecting key presses.

Figure 2.15: Virtual Laser Keyboard [73]

Other example is gaming devices stimulating touch sense using tactile, force or rumble feedback. Generally can be said that it is possible to have device with various input and output combinations.

2.5 Design Factors

Creation of user interface of any device or application always requires certain decisions about what should the user interface be like. This chapter presents factors that should be considered when creating user interface. This step is very important as it is defining restrictions and requirements on user interface.

2.5.1 Factor I – Functions

One of the factors, considered when any application or device is being designed, is its functions [6]. Functions of application or device always specify what will the application or device do and how it will be done. The more functions it will have, the more complex user interface will have to be designed because it will contain more user controls with more complex options. The amount of functions can have influence on target group of users and price. It's more likely that more functions will mean longer development time and thus

increased price. Also, number of functions and their complexity can attract or distract customers (more in chapter 2.5.4). That is why number of functions and its complexity should be chosen wisely.

2.5.2 *Factor II – Price*

Price is a key factor for every manufacturer and for every user. All parts of every device and time spent by designer to create user interface consumes money. Materials of manufactured device, its functions, complexity of user interface, target group of users – everything is reflected in the final price. Expensive devices with high quality user interface will most probably buy only rich users, professionals, or hobbyists which will use all the features of the device. Common users will most probably look for devices with low price because they do not want to spend lots of money for device with tons of features they will hardly ever use.

2.5.3 *Factor III – Design & Look*

Design and look of user interface is one of the key features of the most devices and applications [6]. Although functionality and ease of use is important for most professionals, common users also care about look and feel of their devices and its user interfaces [74][75]. Many people use devices that fit their lifestyle – it can be cell phone, MP3 player or other device which should have their favorite colors, favorite shapes, or play favorite sounds. It is clear that many of the common users notice new device and its user interface if the design is appealing enough to fit their style and that is why such devices are often called *stylish*. Design and look are the next aspect after function and price that can tell whether user buys new device or not.

Some companies tend to spend on user interface design lots of resources to attract more customers, sometimes even at the cost of worse user interface quality. Artists and designers create impressive designs of cars, cell phones etc, but most of these designers usually does not have enough knowledge about functions and requirements on final product and sometimes happens that it is impossible to produce such product because it is too small to fit all required parts etc. That is why it is important to precisely specify requirements on user interface or device and then let designers create impressive designs that can attract potential customers (or users).

2.5.4 *Factor IV – Group of Target Users*

Complexity and ease of use of user interface are opposite features [76] and it is almost impossible to have complex, while easy to use, user interface [76]. That is why it is important to consider requests of target users and their skills. This factor was often overlooked but current devices, or applications, are becoming more and more tweaked for certain group of users, some of them support multiple user interface layouts to satisfy wide group of users with varying skills and requirements.

An application that can work with images can be an example. Basic functions cover opening, saving, closing and some image editing. Now it is important to define target group of users to which is this application designed for. Common users would appreciate tools that would allow easy and fast editing of basic image properties such as brightness, contrast, gamma, hue, saturation and some effects to reduce noise and red eyes effect. Skilled users need these functions too but also might require precise parameter settings and advanced filters which might be much more difficult to use but would provide better image processing with higher quality. While skilled users are able to use such advanced tools, common users would most probably try the tools but because of its complexity they switch to other application that will be more understandable. Adobe Photoshop [77] is a typical example of image editing application that is aimed to professionals. It has lots of advanced filters and tools that allow image editing and filtering. Opposite the Adobe Photoshop, there are applications such as HP Photosmart Essentials [78], which allow easy image printing and saving with only few options. All of these applications are excellent – but only when used by appropriate group of users. Most professionals would never use Photosmart Essentials because it does not have advanced tools for image editing and would not be able to do all the tasks they need. Common users would never use Photoshop because such simple function as image printing requires more than selecting image and pressing "Print" button (and it is similar with other functions).

That is the reason why certain users tend to select devices, or applications, with certain level of user interface complexity which reflect their skill in the area of application's functionality or cover actions that can quickly finish their goals. User interfaces designed for general public must be easy to use with commonly requested features. On the other hand most professionals prefer complex user interfaces with high degree of customizability so that their work will be of a high quality. However there is also category of user interfaces that are easy to use with

complex features, targeted to amateurs who request more than simple user interface and basic functionality. These user interfaces are successful too although group of such user base is not as wide as general public. And that is why user interfaces offering various degrees of functionality are needed to fulfill demands of all users.

2.5.5 *Factor V – Context*

Another important factor that significantly affect user interface and whole human – computer interaction is situation, in which will be user interface used [4][79]. It is obvious that situation can affect the size of the used device and thus size of whole user interface. Device that is primarily designed to be portable device should have limited size and light weight. Such device cannot use full size input – output parts (such as display, or keyboard), but must have size-limited parts that would fit size of the device. On the contrary, devices that are designed for home use can be of greater sizes because devices for home usage can take more space than portable devices and in some cases it is also their benefit (home theatre systems).

Situation is also element that can select input-output channels [79]. In most situations, visual information output is the most effective one because human vision can process lots of information in very short time. This, however, cannot be used in every situation. For example, visual information is not suitable in situations, where user must take visual attention on something important (such as driver has to watch the road). In this case, when driver has to watch some kind of visual interface installed in his car, he has to watch important details that are visualized on the device and thus stop watching the road. This could lead in an accident and could be dangerous. Driver could crash into car in front of him because he did not notice that car in front of him is slowing down. In such cases, it is more convenient to use other output channels such as audio. Computer voice can inform driver about all important events such as navigation, speed limits, distance from car in front of him and many others. Information input to a device in a car is also problematic. Driver cannot input data using keyboard because he has his hands on steering-wheel and releasing his hands from steering-wheel could cause an accident too. That is why another input method should be selected, such as touch-screen with several options on which driver could simply point by one of his fingers and execute selected action or voice commands.

From example above it is clear that situation, in which will be interface primarily used, is crucial for proper user interface design. This aspect can affect what input and output channels

will be used and how significant each channel will be and how large user interface can be for easy handling and operation.

2.6 Requirements for User Interface

The full set of requirements for user interface should include all the technical requirements for the system as well as the specific requirements that will ensure the user interface is easy to use. Requirements are usually gathered from users by using various techniques such as interviewing and observation. However many problems can arise when analyzing the requirements. Potential problems cover small number of potential users, communication with users with different levels of understanding of analyzed problem, or dislike of certain users to support new system etc.

Identified requirements for user interface can be expressed in following terms [80]:

- *learnability* – the time and effort required to reach a specified level of use performance, also described as ease of learning,
- *throughput* – the tasks accomplished by experienced users, the speed of task execution and the errors made, also described as ease of use,
- *flexibility* – the extent to which the system can accommodate changes to the tasks and environments beyond those first specified,
- *attitude* – the positive attitude engendered in users by the application.

These requirements were later transformed into five dimensions, referred to as five Es [81]: effective, efficient, engaging, error tolerant and easy to learn.

2.6.1 *Effective*

Effectiveness is the completeness and accuracy with which users achieve specified goals [81]. It can be determined by checking if user's goals were done successfully and if his work is done correctly. Effectiveness can be sometimes confused with efficiency, but they are not the same. Efficiency identifies how quickly goal can be completed, while effectiveness identifies how well is the goal done. Although it might seem that efficiency should be the first and primary requirement, it is not always truth. For example, in interfaces of banking systems (such as ATMs), effective use of the system – cash the correct amount of money, selecting the right account, making a transfer correctly – are more important than gains in speed. This assumes that the user interface designer has not created an annoying or too complex interface

in the name of effectiveness.

2.6.2 *Efficient*

Efficiency can be described as the speed (with accuracy) in which users can complete the goals for which they use the product [81]. ISO 9241[82] defines efficiency as "resources expended in relation to the accuracy and completeness with which users achieve goals" [82]. Efficiency metrics include the number of clicks or keystrokes required or the total 'time on task'.

2.6.3 *Engaging*

An interface is engaging if it is pleasant and satisfying to use [81]. The most obvious element of this characteristic is visual interfaces and their design – specifically the style of the visual presentation, graphics images, animations, colors, use of sounds and other multimedia elements – it is all part of user's immediate reaction. But there are also more subtle aspects of the user interface that can affect how engaging it is. For example the text size, font, and readability of the text can change a user's opinion to the interface, data and information grouping can change user's opinion as well. Very important is the style of the interaction – user can prefer, for example, textual menu system from abstract icon representation.

2.6.4 *Error Tolerant*

The goal of every application is to prevent errors from human-computer interaction. But developers are only humans, and that is why applications are not perfect so errors may occur. By error tolerant application is understood program, designed to prevent errors caused by human-computer interaction, and to help user recover from any error that occurs [81].

2.6.5 *Easy to Learn*

One of the biggest objections to "usability" comes from people who fear that an easy to learn application will be used to accomplish simple tasks, but which is not powerful enough for long, sustained use [81]. A user interface that is easy to learn allows users to build on their previous knowledge. This can include built-in instruction for difficult or advanced tasks, tutorials, or knowledge bases which are critical to effective use.

The five Es can be used in a number of ways to produce a successful user interface. It is important to keep in mind that requirements gathering for user interface is iterative and should

involve users. Typically, during requirements gathering some information is collected, which is analyzed, negotiated to users, and then another round of requirements gathering takes place. This cycle continues until all users are satisfied with requirements and project schedule starts application development.

3 User interface development

User interface development is an iterative process and it is always long distance run. When all necessary information and requests on required application (or device) are collected from potential users, user interface designers start their work with communication with users, defining most frequently used features and user interface requests. The collected information is analyzed and initial version of user interface is negotiated to users, who supply designers with their feedback on user interface satisfaction and remarks. Cycle with information collecting and negotiation repeats as long as users are satisfied with presented user interface, which might take long time. Sometimes it can happen that users may have very different requests on user interface which is leading in several kinds of user interfaces designed for each distinctive type of application usage. A typical example of such problem is well seen in applications that are designed for common users but still want to enable full control for advanced users. Such applications allow switching of user interface from basic or easy mode to advanced or expert mode, which offers much more sophisticated controls that require certain level of knowledge of advanced features. On the other hand, easy, or basic, mode offers limited functionality that most usually work with pre-set values of advanced features hidden in basic user interface mode.

3.1 User Interface Creation Tools

Each user interface consists of user interface components that have certain functionality. Early operating systems however did not have libraries with standardized user interface elements that could be used by software companies to create applications with high external consistency. This problem caused that each company, producing software, had their own set of user interface components, which were used in their software and thus software products from each distinctive software companies had high consistency but applications from other companies had completely different look and feel. This state was very problematic mainly for common users who had to take courses for almost every application they were supposed to use.

Fortunately, companies producing operating systems and user interface developers were aware of this problem and started to supply user interface elements libraries that could be used by general community of programmers to produce applications with similar look and feel with same user interface elements that users know well. With well known user interface elements,

users could be able to expect functionality of known user interface elements and thus start to use most applications intuitively. Since first user interface elements libraries were created, many ways of using interface elements libraries were proposed – first tools for user interface design were created. Following paragraphs discuss libraries and approaches, that are relevant to this work.

3.1.1 *API – Application Programming Interface*

First APIs, created before an object oriented approaches became widely used, were libraries of functions – called often as functional APIs [4]. Functions in these APIs were covering commands for creation of basic user interface elements such as windows, buttons, labels or textboxes, changing their general appearance like caption of the form or text of the button, and controlling their general behavior. Getting status and user input information from user interface elements is done in a message handler. Message handler is a function associated to user interface element and is called every time user element changes its inner state. A typical example of inner state change is when user clicks a button. Button's inner state changes from "released" to "pressed", appropriate event is raised and message handler function is called with parameters which can be further processed. Every user interface element that is created using functional APIs must have its message handler function. Without it, programmer is unable to catch important events and status changes of the element and the element becomes only visual part of the user interface. It is obvious that creating message handlers for almost every object in the user interface is ineffective and also the code is not well readable. Message handlers often contain huge switch statements with enumeration of processed events and their handling. Some message handlers can spread wide to several screens of source code.

Functions used for creation of user interface elements or for user interface element state change usually contain several parameters which are distinguishing type of user interface element, its attributes and flags and many other features. Many of the features are unique for each user interface element and programmer must know all the flags and enumerations to be able to implement user interface control properly. Learning all the flags and enumerations is difficult and most programmers prefer using documentation, learning only the most important and most frequently used constants. Thanks to such constant and functional complexity, functional APIs are difficult to learn, their using leads in low productivity (when compared to modern tools for user interface design), requires good knowledge of programming and library functions, every change in user interface requires significant changes in code. These cons led

later in an object oriented design of user interface APIs. Typical example of functional kind of APIs is Windows API [83] (also known as Win API).

Object oriented user interface APIs are libraries, created with respect to object oriented design, containing classes that represent each user interface element. Creation of an instance of a class creates user interface element whose features and behavior can be changed using properties, methods and events supported and implemented by instance of the user interface element. Object oriented design of user interface can benefit from object oriented API and directly inherit from objects that are available in the library of the API. Thanks to inheritance, user interface code can be well structured and can be created much faster because most of the code, required for user interface element creation and control, is already implemented in the base classes of the API. Programmer or user interface designer can focus on their task more than on user interface specific code, thus increase productivity and reduce time needed for user interface prototyping. However object oriented APIs can suffer from high level abstraction. User interface elements contained in the library contain many features and functions, but thanks to inheritance and information hiding, programmer or user interface designer sometimes could not be allowed to change specific features and behavior of desired object which can be sometimes unacceptable. That is why object oriented APIs are sometimes built over functional APIs and allow code mixing of object oriented API with functional API to enable both high level and low level abstraction so that programmers have full control over the features of the user interface and can choose whether to use high level approach or the low level one. Common practice is to use high level approach and in specific cases call few low level functions to alter the state of some objects at appropriate time. Typical example of high level object oriented API for user interface creation is Windows Forms [84] (also known as WinForms), available in .NET Framework [85] library. Objects in Windows Forms contain property of handle, which can be used in Windows API to control certain specific features of object that are in Windows Forms hidden by default.

Direct use of APIs for user interface programming involves writing source code in a source code editor. This requires lots of time and is prone to errors (e.g. typing mistake).

3.1.2 *Visual Programming*

Visual programming [4] is completely different from standard programming using APIs. Programmer or user interface designer, who spend hours in integrated development

environment (IDE) [86], does not code user interface in a source code editor (as it was typical in the case of direct use of API), but drags components of user interface in a special part of IDE designed for graphical user interface construction. Most modern IDEs are equipped with various tools to simplify application development and graphical user interface construction tool is one of them.
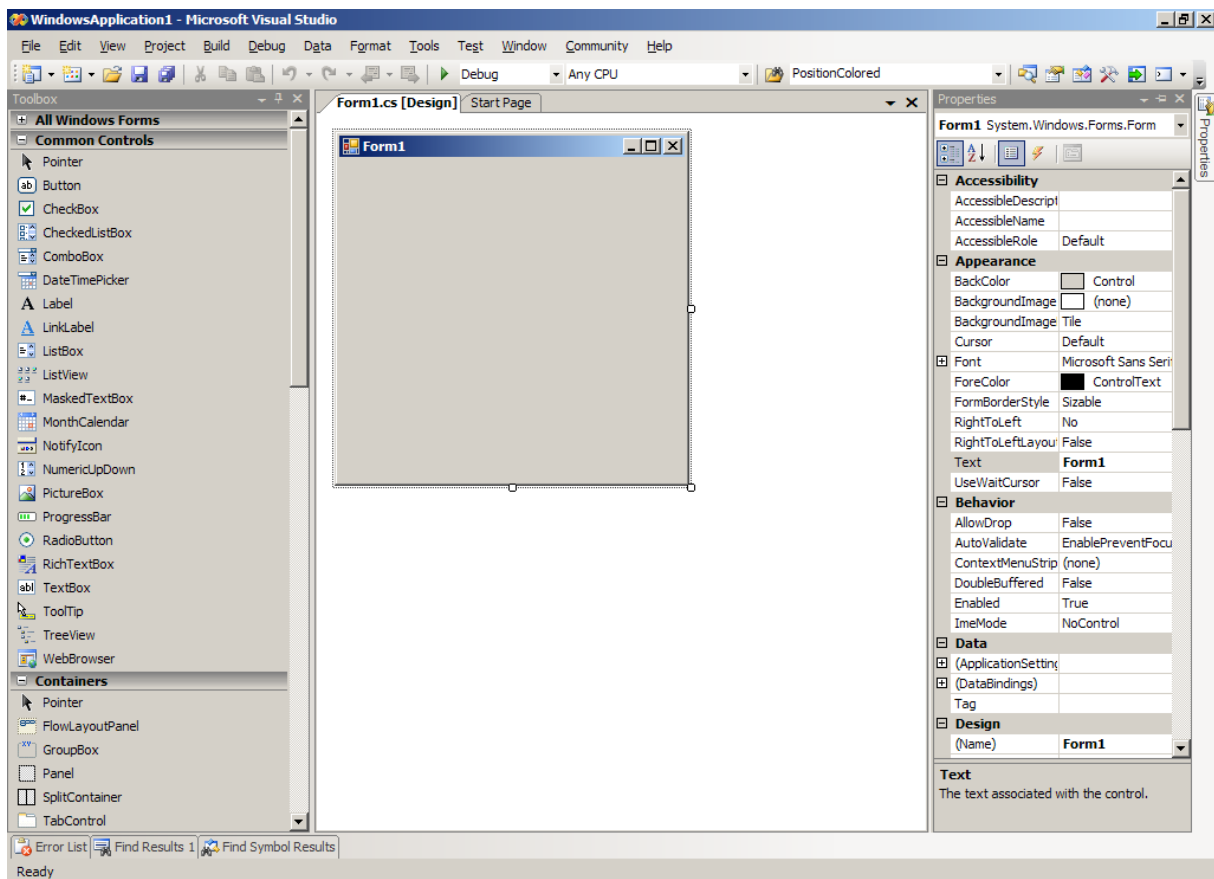

Figure 3.1 Microsoft Visual Studio 2005 IDE

Most of the graphical user interface construction tools consist of several windows that speed up and simplify design. Most important window is usually design window showing design and layout of user interface elements. Thanks to design window, user interface designer can see final look of created user interface component, change layout and properties of added user interface elements and modify the behavior of the user interface component. User interface elements are added to design window from a list of available user interface elements, also called toolbox. Toolbox contains user interface elements that can be used in the user interface design. All elements contained in the toolbox are usually divided into several logical categories (containers for panels and group boxes, menus for menu items etc.) to speed up user interface element lookup. Placing of new user interface elements from the toolbox is, in

37

the mind of visual programming, dragging selected user interface element to design window and dropping on its place. This approach is very fast and simple user interface components, dialogs, or windows can be designed in a short time.

Third common window in a graphical user interface construction tools is usually so called properties window. Properties window is used to alter default behavior and properties of user interface elements added to designer window. Properties of selected user interface element, such as color, background, fonts and others, are displayed in properties window and can be easily modified. Events and behavior of selected user interface element can be modified too by specifying some event or behavior handler. These handlers can be implemented in code editor by designing person or by another programmer depending on coding skills of the designing person.

Although visual programming is much faster than direct programming using functional API calls, there are still several drawbacks. The first problem is choosing the right API. Most of the graphical user interface (GUI) construction tools are designed to work with certain user interface API. For example, Microsoft Visual Studio 2005 [87] supports Windows Forms [84] and Microsoft Foundation Classes (also known as MFC) [88]. Other IDE support other APIs, some of them more than one and there is also community of programmers that are trying to create IDE which would support virtually any user interface API. Another thing is that visual programming is, based on added elements in the designer window, generating application code that is later modified by programmers who are implementing events and behaviors of the created user interface elements. Application code that is modified by developers can be difficult to interpret by designer window and additional modifications or additions of user interface elements can lead into several problems, when code generator is unable to add or modify content of design window or visually reconstruct user interface from the source code. Reason can be mistake in application code caused by developer or application code complexity.

3.1.3 *Scripting Languages*

Scripting languages are programming languages that are stored in textual, readable, form and are interpreted during execution [89]. Main benefit of scripting languages is that it can be modified by any user at any time so that anyone has full control over the application and can extend its functionality or correct possible errors. However programming in scripting

languages require good knowledge of the script keywords and libraries because most of the scripting languages do not come with source code editor, but rather use simple text editing software for the purpose of programming. That is why beginners often use manuals because text editors do not highlight possible typing errors. Typical example of scripting language is TCL/TK [90], Perl [91], PHP [92] and other.

TCL/TK is scripting language that has graphical user interface toolkit (called TK) allowing creation of user interface with automated layout techniques. Using TK, it is possible to create instances of various user interface elements and define only parent container of the created user interface element (for example, see Figure 3.2). TK interprets during execution script commands and creates windows and dialogues from the script automatically and selects exact positions and sizes of user interface elements. Although TCL/TK is powerful scripting language that could be used in the process of automated user interface generation, TK contains only few basic user interface elements.

```
package require Tk
set w .states
catch {destroy $w}
toplevel $w
wm title $w "Listbox Demonstration"
wm iconname $w "listbox"
positionWindow $w
label $w.msg -font $font -wraplength 4i -justify left -text "A listbox
containing items."
pack $w.msg -side top
frame $w.frame -borderwidth .5c
pack $w.frame -side top -expand yes -fill y
scrollbar $w.frame.scroll -command "$w.frame.list yview"
listbox $w.frame.list -yscroll "$w.frame.scroll set" -setgrid 1 -height
12
pack $w.frame.scroll -side right -fill y
pack $w.frame.list -side left -expand 1 -fill both


$w.frame.list insert 0 "One" "Two" "Three" "Four" "Five
```

Figure 3.2 TCL/TK script showing listbox with numbers

3.1.4 *Markup Languages*

A markup languages [93] used for description and rendering of graphical user interface are called user interface markup languages [4]. Main goal of these languages is to simplify and reduce work needed to develop and design user interface. These languages allow description of user interface in declarative form (most usually in a text form) and abstract it. Instead of specifying location and look of the user interface elements they rather define what elements will be used on what data and how they should behave. Thanks to abstraction, same user interface can be used on various platforms with various capabilities. User interfaces, defined with user interface markup languages, can be used by platforms with graphical output to show graphical user interface, platforms with audio output to report information in audible form, platforms with audio input to receive input commands using speech etc.

Huge amount of different capabilities is the main problem of languages that tend to be as much general and universal as possible. Although this universality and versatility is not bad in the sense of platform portability, however it is also problematic for most of universal user interface markup languages. That is also the reason why there are domain-specific markup languages attempting to describe user interface for certain, or limited amount of, domains. These languages do most usually their jobs better but they are less flexible.

To enable platform portability, it is also important to have compiler that would allow connection between interface and non-interface code. Situation gets even more complicated when porting on various operating systems, where a need for special compilers that are aware of capabilities of the target platform exists. Compiler must select appropriate user interface elements, select optimal input (keyboard, speech) and output (audible, graphics) method, generate user interface, connect generated user interface with application system code and make all this working together. For this purpose, two major types of frameworks were developed – compilers and interpreters. Compilers to native code compile system and interface code to one single package that is directly usable on target platform which is native for the package. This concept does not require any additional libraries or frameworks on the side of target platform. However there is need for specialized compilers that support each distinctive target platform. Also single package is not portable besides its target platform to another platform and there is need for many various packages for various target platforms.

Interpreters compile system and interface code to one single package that is not in the native code but in some kind of intermediate code. Package is target independent and can be ran on virtually any platform that has installed interpret for the intermediate code. This intermediate code is interpreted on the target platform and creates user interface based on platform capabilities so that it is suitable for the platform.

It is obvious that both concepts have their cons and pros. While there is need for multiple distinctive compilers for various platforms, interpreters require multiple framework implementations for each distinctive platform.

UIML – User Interface Markup Language

The pioneer in user interface markup languages is certainly UIML [94][95]. UIML is an XML [96] language that is used to define the actual interface elements. It defines location, design of controls, user interface structure, events, data and styles to allow user interface displaying on various platforms.

```
<UIML>
<HEAD>
  <AUTHOR>Jaroslav Kadlec</AUTHOR>
  <VERSION>1.0</VERSION>
</HEAD>
<APP CLASS="App" NAME="DialogApp">
  <GROUP CLASS="Dialog" NAME="PrintFinishedDialog">
    <ELEM CLASS="DialogMessage" NAME="PrintFinishedMsg"/>
    <ELEM CLASS="DialogButton" NAME="OKButton"/>
  </GROUP>
</APP>
<DEFINE NAME="OKButton">
  <PROPERTIES>
   <ACTION VALUE="DialogApp.EXISTS=false" TRIGGER="Selected"/>
  </PROPERTIES>
</DEFINE>
</UIML>
```

Figure 3.3 UIML code example

Because this language is well extensible it allows description of user interface technologies

not yet invented and that is why it is using structures, events and data that imply particular interface technologies. Because of its general user interface description, there are few compilers capable of producing user interfaces. That is also the reason it does not attract much attention.

XUL – XML User Interface Language

This user interface markup language is developed by the Mozilla project [97]. XUL is built over existing standards such as CSS [4][98], JavaScript [99] and DOM [100]. Because XUL has no formal specification, its only implementation is only in NGLayout (also known as Gecko layout engine) [101]. The main benefit of XUL is that it provides simple and portable set of user interface elements, which reduces software development effort in a way similar to fourth-generation programming languages. While XUL is used mainly for Mozilla based applications, it can be also used in remote and locally ran applications. However because XUL is supported in NGLayout only, it is available to users of browsers, which are based on Mozilla. Because NGLayout is designed for graphical output, XUL is designed for platform with graphical output support.

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<window id="findfile-window" title="Find Files" orient="horizontal"
  xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
  <button id="find-button" label="Find"/>
  <button id="cancel-button" label="Cancel"/>
</window>
```

Figure 3.4 XUL document example

XAML – Extensible Application Markup Language

This markup language is developed by Microsoft and is extensively used in .NET Framework 3.0 [85], particularly in Windows Presentation Foundation (WPF) [102] and is XML-based. Because XAML contains also program logic and styles, it is often called application markup language. XAML defines user interface elements, data bindings, events and their handling, and other features. Every XAML element maps directly to instance of user control in .NET Framework and every attribute maps to its property. From this is obvious that XAML is designed for platforms that support graphical output (as well as XUL) and is mainly supported on desktop computers and pocket PCs based on Windows platform.

```
<DockPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0">
    <Canvas DockPanel.Dock="Left" Width="230" Height="100">
        <Image Source="helloworld.png" />
    </Canvas>
    <Label VerticalAlignment="Center" FontFamily="Trebuchet MS"
        FontWeight="Bold" FontSize="18" Foreground="#0066cc">
        Label Hello World
    </Label>
</DockPanel>
```

Figure 3.5 XAML code example

There are also other markup languages, such as GTK+[103], OpenLaszlo[104], which are mainly used for graphical user interface creation or web content definition. The three presented here are most common and typical representatives. UIML is one of the few targeting general user interfaces, XUL targets mainly web applications and XAML can handle web, 2D and 3D applications. When comparing available user interface markup languages, UIML is the only general markup language general enough to be used to describe virtually any user interface for any platform. The reason why this user interface markup language has only limited amount of attention is because of the complexity when creating user interface. Although the description itself enables general user interface description, its design for wide number of platforms with varying capabilities is very difficult. That is also reason why user interface languages such as XAML or XUL are much more successful – they always render graphical user interface and have limited predefined number of user interface elements and well defined platform capabilities.

3.2 Choosing "interaction elements"

With the right tool for user interface creation, it is important select right interaction elements. Most important user interface elements, based on their information role, are text, color, image, animation and sound [6]. Although it might not seem so, other elements such as buttons, list boxes, and menus are command and control user interface elements and their proper usage is based on the requirements on user interface rather than means of information communication. Above mentioned user interface elements or properties are basic, and usually easy and cheap to change but applying them in a consistent manner can make big difference to the usability of whole user interface [6].

3.2.1 *Text*

Text is the dominant element of almost every application. It plays main role in web sites, education and training software, word processors, file managers etc. It is a flexible means of communication. That is also the reason why it is important to keep text legible [6]. Legibility is a product of blend of several factors, such as font, size, spacing, color and others. This is a complex area and it is very difficult to develop guidelines that apply in all circumstances. It is always important to take into account needs of the users, because they might have poor eyesight.

3.2.2 *Color*

Color might be used for variety of reasons – to draw attention, to show status, to make information clearer, to make information more attractive. That is why it is important to think of what will color mean to the target group of users. Red color might mean warning or danger for most of the people, but in china, it is a joyful color [6]. Color saturation and brightness are important aspects that will affect ability to read text or notice and recognize information. A good practice is the usage of colors with sufficient contrast between the background and foreground. Also choosing too bright colors might tire user's eyes when used on large areas intensively. Making user interface allowing user to change colors is always preferred, because every person perceive colors differently [6].

3.2.3 *Image*

Images can be used in several ways: to motivate, to attract attention, to amuse, to communicate information (especially spatial), and to support interaction. Because there are several types of images, that represent different types of information, it is important to consider using right image [6]:

- choosing appropriate type of image (picture, diagram, graph, chart) according the information that need to be conveyed
- design the image so that it meets the requirements of the task as closely as possible – unnecessary images or graphs might distract user
- combining text with images is more effective than using image without description

Using right images improves usability and readability of user interface but it must be used wisely. When using images, resolution of the display and storage capability should be considered, because images are usually large files.

3.2.4 *Animation*

When incorporating animation clips or video clips into user interface, it is important to provide high quality animation/video without any glitches, flickering or other defects. Users are used to seeing high quality animation and visualization from TV and other sources and low quality animation is generally distracting, because users expect some level of quality and using user interface that do not meet the standard will not motivate users to use it [6].

Animation can be used to illustrate movement or provide dynamic feedback. A typical example is copying or moving files animation that illustrates files flying from one folder to another. This feedback is confirming that something is happening and system is working. Animation can also be used to attract attention. A typical example is advertisement banners on web pages. However animation attraction attention could be after longer period of time visually irritating.

3.2.5 *Sound*

In most of the current user interfaces, sound is still considered to be secondary user interface element and in most user interfaces it does not play big role as most user interfaces focus on visual appearance. However sound is very important and useful output when user cannot easily see or notice visual output or prompt. These situations cover visually impaired users and applications, where user's eyes and attention are required on something else (process control, sport etc.).

Sounds can be subdivided into several categories – ambient sounds and sound effects, music, and speech [6]. Sound effects are usually used to simulate ambient sounds that confirm device functionality (clicking of onscreen keyboard), attract user's attention or confirm success of some operation. Music as a means of communication is not very common and is mainly used in games or specialized applications for evocative purposes. Speech is very powerful in the means of communication because it can tell message with additional information such as tone and accent. Speech is especially advantageous in case of applications for visual impaired people. Using sound, however, have several drawbacks. Using prerecorded audio clips requires storage and generating speech requires lots of processor power.

3.3 Evaluation

Complete user interface with all features implemented and tested to be working is usually enough to be released to public. This, however, does not guarantee that user interface will be effective and easy to learn for wide area of users all over the world. Only limited number of applications is usually evaluated for performance and effectivity. It is most usually goal of user interface of each particular application because there is many applications designed for same tasks doing them in a different way – it is hard to say which of the ways is more effective or productive. Also, it is important to consider habits of user and his preferences when choosing application for his needs.

3.3.1 Aspects

Evaluating user interface has several methodological aspects that need to be considered before own evaluation is started [105]: user selection and training, task selection and presentation, so called Wizard-Of-Oz simulation (also known as Oz Paradigm) [105][106].

User selection and training

Evaluating any user interface requires test participants who represent the intended group of target users [105]. It is very important factor because it depends on how it matches desired user knowledge about the domain of the application. Therefore, test participants should have knowledge similar to that had by the users in the target domain. Some evaluations prefer to train their own evaluation participants that will be employed in the evaluation in the tested domain. There is probability that such participants, employed in the evaluation, will be less motivated than existing users. On the other hand, employing trained users provides a measure of control over the common understanding of the domain. User training might vary from minimal introduction to extensive training including instructions on common tasks and mistakes. Because the quality of training significantly affects user performance, the kind and amount of training should represent training and knowledge that users are expected to have.

Task selection and presentation

The goal of evaluation of user interface is gathering experience of using application, its features, and user interface elements as they are used during assigned tasks. These tasks should represent type of work users are expected to do with the user interface. Tasks must be presented in a way that will not affect the evaluation. Range of tasks, selected for the evaluation, should cover all the available functions of the user interface. It is also possible to

let the users generate own tasks. Main advantage of this approach is that users are free to express needs or problems that are beyond the capabilities of the user interface and can lead to major user interface improvements. This however leads to loose of control over evaluation because users might not cover whole range of functions offered by the user interface.

Tasks can be presented to users in two distinct forms. First form can guide user step by step to complete the task. This, however, can result in absolutely correct user interface usage which is not desirable. Second form presents only bulleted list of required goals that need to be done to complete the task and it is user's choice how to do them. This form of task presentation gives the evaluator most control over the task complexity when compared to free will of participants while performing the task.

Wizard-Of-Oz simulation

Wizard-Of-Oz (WoZ) simulations simulate system using a human to interpret participant's commands and actions [106]. In a typical system, participant generates a command by keyboard or mouse, that appears also on the computer of an operator (the Wizard) hidden in another location. The Wizard interprets the commands and does actions that result on the computer of the participant as a result of his action. It is obvious that results and responds of the Wizard will not be as consistent as computer would. This fact should be taken into account when reviewing results of evaluation. However, WoZ simulations are great for quick evaluation of potential designs and requirements because the Wizard simulates behavior and actions as the final system would.

3.3.2 *Methods*

Evaluation of user interface is difficult task because it is important to define what to evaluate. It is always good to answer questions on what is the purpose of the evaluation, what information should be collected or who will be the target group of users. Knowing answers on these questions simplifies choose of right evaluation method and produces best results. In fact, the goal of evaluation is not to say user interface is good, or bad, or it is that good. In general, the purpose of the evaluation methods is to help in the identifications of the problems and solutions. Then difference between "how good" to "identify problems and solutions" makes a big difference in choice of methods. Following text describes some of the most commonly used methods for evaluation: verbal reports, questionaries, data collection, walkthroughs, and reviews [105].

Verbal reports

Verbal reports can be used at any time of user interface design cycle. Verbal reports are summaries of users' cognitive experiences of the user interface usage.

Questionaries

Questionaries can be used at any time of user interface design cycle. Contain subjective responses of users on certain user interface aspects and can generate very interesting results.

Data

Data collection is used since user interface model is complete. It measures specific data from real or simulated system. It is meant for specific test of alternatives or user interface coverage. Data measuring is best for performance evaluation and for competitive evaluations.

Walkthroughs

Walkthroughs are usually done in early stage of user interface design between user and design team. Thanks to feedback of users, it is possible to make important user interface design decisions. Walkthroughs can be repeated several times for various user interface features.

Reviews

Reviews by experts from user interface designers are important to solve general usability problems and in decisions on general acceptability of user interface architecture

Besides empirical methods, there are also automated methods that can be used to evaluate user interface. These methods work on collecting metrics such as distance from user interface elements or number of clicks etc. A task is specified in form of commands that a user has to perform. Each command is algorithmically performed and during the command execution, metrics are collected (travelled distance, clicks, time). Collected metrics measure effectiveness of the user interface. However these methods cannot measure efficiency or other qualitative

Evaluation, however, does not end by obtaining results. Next step is to redesign the user interface to meet whole feature set such that users are able to achieve their goals and are satisfied with whole system.

4 User interface design automation

Many tools and approaches were developed to support user interface design because applications are getting more and more complex and thus requirements on the user interfaces were getting more and more complex too. To allow creation of more complex user interfaces, a more complex tool had to be created that would allow creation of demanded capabilities. Creation of more and more complex application and increasingly complex user interfaces however brings several drawbacks. Complex user interfaces are hard to build even with today's user interface creation tools. Application architecture must be well designed to enable proper implementation of all application features. Architectural structure has impact on user interface design because designer must be aware of requirements of each application package. Packages can have various objects requiring specific data inputs, and can be stored in various collections that can or cannot be browsed by users. Opposite of proper user interface design is requirement on application testing. Early implemented packages with significant impact on application behavior and functions must be tested because other application packages might be dependent on these packages. Testing packages with incomplete user interface is sometimes very difficult and user interface designers are forced to create simple test modules that can be run to test desired behavior. This iteration cycle is usually part of every application development process. After all, when first implementation of user interface is complete and evaluated, packages are modified based on evaluation data and requests on user interface design are taken into account when new architecture of user interface is being designed. Such process of creation of user interface is very difficult and that is why there is a kind of user interface tools that are trying to simplify the development phase of application, including user interface design, by applying additional coding, modeling, and application behavior describing to provide automated user interface creation.

The goal of this chapter is to describe some principles used in these days to support automated user interface creation. Methods mentioned in this work are not complete enumeration of available methods. Instead, only those most important and successful methods will be described that cover general principle of automated user interface creation or which have direct impact on the concept of this work.

4.1 Separating user interface

Before discussion over separation user interface from application code, it is important to

understand how current applications are built and what their structure is like. Most of the applications in the past that were not implemented with object oriented approach in mind contain most of the user interface code completely mixed inside of other functions that somehow affect user interface appearance or behavior. Although this is not good solution, it is important to remember that user interface APIs, in that days, were not common and most companies created their own user interface libraries of various quality. Also user interfaces based on textual output such as in MS-DOS applications were, when compared to these days' user interfaces, very simple. Graphics of these user interfaces were composed of various symbols with colored background and text mode allowed display up to 80 symbols on each of 25 lines. There is no doubt that without object oriented design, functions drawing information on the screen were mixed through the application code. Major drawback of poorly designed applications required lots of work every time user interface needed to be modified. It is because application code was calling on various places stand-alone methods for setting background color, text color, text printing etc. We can speak about mixing user interface code into application (non-interface) code.

With applications based on object oriented design, where most of the user interface elements were available as single objects, situation changed. Application code become cleaner from user interface code and contained only few user interface code calls. Developers, trying to maintain their application code as much independent from user interface code, loaded more work on the user interface designers, who were forced to mix their user interface code with application code. This approach became very popular. Application code contains objects that perform certain functionality and user interface code creates its instances and modifies their parameters and behavior. Situation is extreme opposite of first applications, where application code was mixed with user interface code. It is clear that user interface and application code cannot exist without each other. User interface requires application code to perform user requested tasks and application code needs user interface to present results and status of application objects.

From this point of view, separation of user interface from application code is almost impossible. Separating two things that are tightly connected seems to be very difficult. User interface is built on requirements of application objects and their outputs, requests on features from users and experience of user interface designers. User interface code is compiled with knowledge of application code and is extensively using it. However, during last few years

were developed markup languages such as above mentioned XUL or XAML that allow some kind of user interface code separation.

General approach of user interface code and application code separation is in creation of packages or libraries that contain algorithms and objects, performing various tasks and actions – a digital media library for example. Such library can contain various methods for loading and saving of digital media files such as images, vector graphics and others, methods performing image data filtering and modifications such as brightness and contrast changes, automatic brightness and contrast balancing, color corrections and others. Having such library however does not necessarily means it is an image editing software. It is important to create user interface that is capable of calling digital media library methods and work with the results returned from the methods. In order to be able using all library functions, user interface is created importing library interface. The created user interface contains user interface elements such as menus, buttons etc. that are calling library methods or setting library objects properties to support required functionality and allow user to achieve expected goals. It is obvious that application code is completely independent from user interface and it is possible to create different user interface with same or similar functionality that might have completely different look and feel. XAML, for instance, creates human readable XML files with user interface definition. This user interface can be run as a standalone application as long as it is not referencing any particular objects or methods from imported assembly. This allows creation of different XAML user interfaces working over same assemblies thus featuring same functionality and behavior. It is even possible to modify XAML files and extend the functionality of user interface at any time.

User interfaces, as it is clear from the text above, are usually tightly connected to application code because it is its functionality and features it is presenting to users. A complete separation of user interface code from application code is not possible unless there is a framework that would allow communication between application code and user interface.

4.2 Task models

An abstract way of specifying the user interface is through the usage of a task model [107]: modeling a user interface on the basis of itemizing the steps of interaction a user has to perform. This approach has an advantage that, when user interface is built for every platform or device where it is executed, the task models are taken into account for each of them. User

interface is created with respect to device's or platform's capabilities. Another advantage of using task models comes during software development process, because task models are not just collections of user interface elements with no semantic meaning but provide additional information regarding the interaction process. Basic form of task models can be defined in the early stage of software development process and help with validation of the software during its development phase and also in the end whether the software meets all the requirements regarding user interaction with the software. Because traditional creation of user interfaces without looking at pre and post conditions of the application is error prone to previous steps taken during software development process, the task models can become the glue between application code and actual user interface [108].

4.2.1 *Creating task models*

Before creation of task models, user requirements analysis must be done and the task designer must understand all the tasks that were deducted from the analysis. After this, some aspects of task models must be defined by the task designer:

- hierarchical structure of the interactive system consisting of tasks and subtasks
- link between user and system for each subtask
- additional properties such as importance or complexity must be set

Creation of task models is most usually done in textual or graphic form. A typical representative of textual form is LOTOS [109], a representative of graphic form is ConcurTaskTrees [110]. The advantage of graphical representation of the formal methods for modeling human-computer interaction is its simplicity. The syntax and semantics are usually easy to learn and do not require knowledge, or experience, with formal languages such as above mentioned LOTOS. Graphic notation of task model allows easy hierarchic task specification. The nodes of the tree represent tasks. If task can be divided into subtasks, these are represented as children of the node. Such tasks, consisting of subtasks, are usually abstract tasks, which only hold other tasks in logical group. Leaf nodes then represent user tasks – completely performed by the user, for example deciding what to do next, interaction tasks – tasks that require user interaction with the system, for example data input to fill the form, and application tasks – performed entirely by the system without user interaction, for example processing information from inputted data and presenting results to the user.
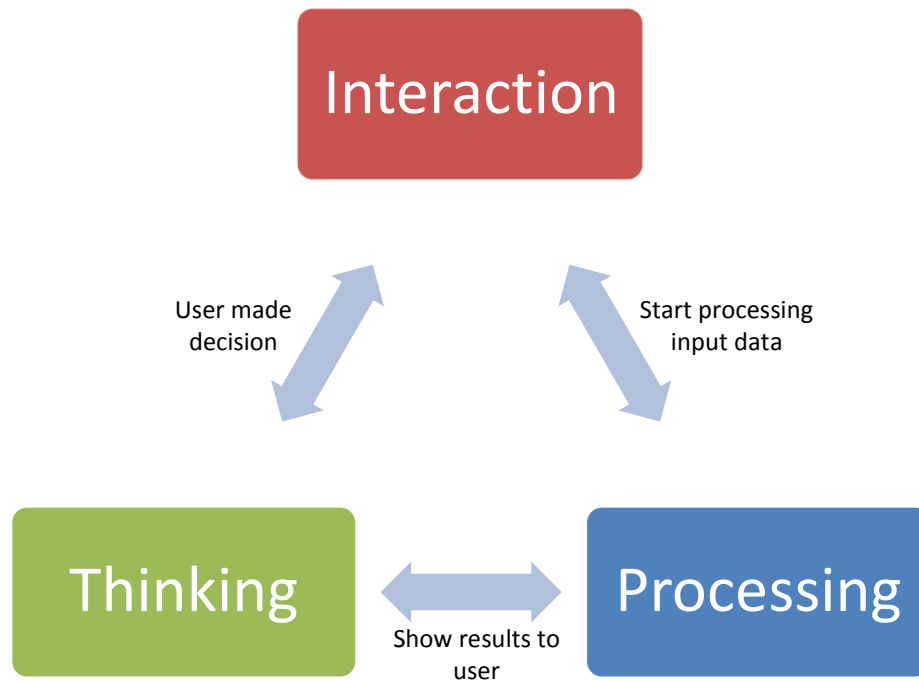
Figure 4.1 Interaction process of user and system

Dividing tasks to distinct categories have practical reasons (Figure 4.1). Interaction usually works as follows: a user decides to use a system. Therefore and interaction with the system is needed. User inputs required data and system starts the processing of the data. When processing is completed, the results are shown to the user. Now the user has to interpret the results and decide what to do next. After user decides what to do, starts the interaction and inputs the data. Inputted data are processed and results are shown to the user. User must interpret the data and decide what to do next. This process keeps going on until user decides to perform a task that shuts down system. This process is analogous to above mentioned task categories.

4.2.2 *Generation of user interface*

After the task model is created, it is possible to run application on target system and let the interpreter to generate the user interface. Following example of the user interface generation process is taken from Dygimes framework [111].
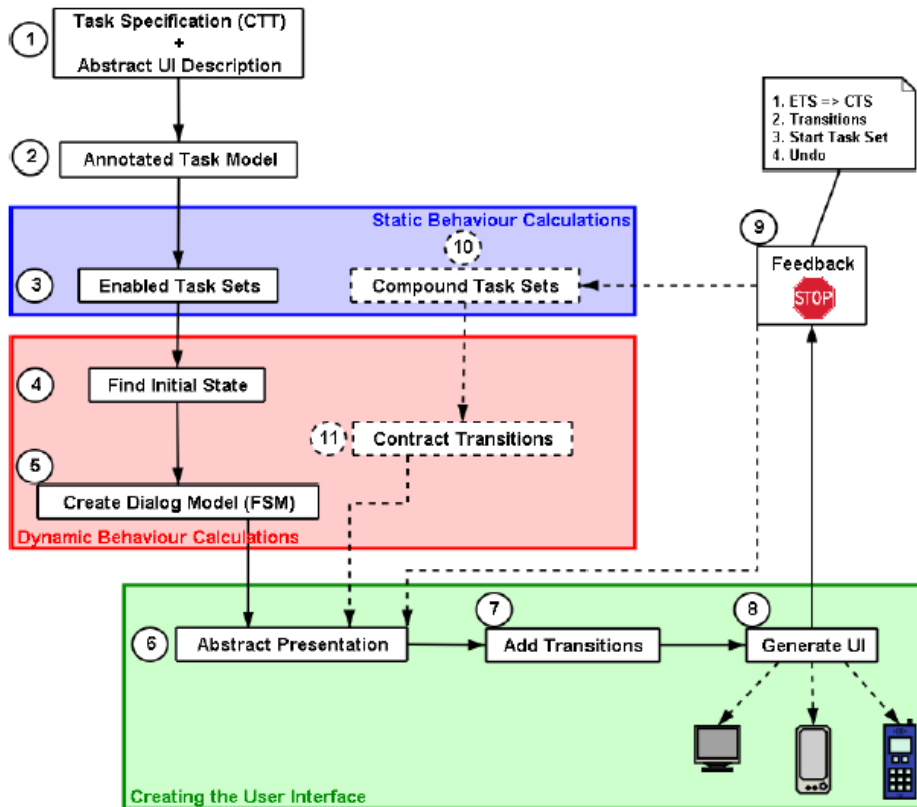
Figure 4.2 User interface generation process in Dygimes framework

First step (1) in the generation of user interface from task models is loading task model and so called abstract user interface description. Both input files are XML-based and abstract user interface description is optional, containing general document appearance (groups of data etc.). In the step (2), input data are loaded and a tree hierarchical structure is created that represents the task model. Then (3) a set of enabled tasks defining which tasks can be performed at the same time is computed. From enabled task sets and task model tree a set of initial tasks is created (4). Initial task set contains tasks that can be presented to user when application starts. Afterwards, a dialog model is extracted from the enabled task sets (5). The dialog model is represented by finite state machine. The construction of the dialog model has to be preceded by the creation of enabled task sets, because it is required during creation of finite state machine. This is the reason why distinctive phases of the user interface creation are separately in this order. From dialog model is created abstract presentation (6). Presentation contains abstract user interface elements that are not connected so the transitions between various parts of the presentation must be created (7). From this point appropriate user interface can be created (8). After this a feedback window is presented (9), where user interface designer can adjust various properties of user interface and let the user interface

recomputed. After the user interface designer commits the user interface, it is used as the application interface.

4.3 User interface management systems

Concept of user interface management systems (UIMS) and first thoughts and implementations of UIMS lead back to 1970-1980s when user interfaces were getting much greater role thanks to PC boom. First interfaces consisted of command line systems, which had to lead user through set of questions to get all required parameters that were needed to successfully perform requested task. Remembering all the parameters of huge amount of commands was very difficult so UIMS came to help create dialogues. To allow creation of additional commands and extending the user interface to support new commands, user interface had to be separated from commands, so that new, more complex, commands could be created from simpler ones – UIMS were created to separate application from the user interface. However saying that that the UIMS and the application are separate is not to say that they can operate independently of one another; information must be communicated from the UIMS to the application and ideally from the application to the UIMS. Such communication of information, where separation is required, demands a clearly specified description of the protocol of information transmission: interface between the user interface and the application. The nature of this user interface-application interface has the major influence on the types of interactive systems that can be built.

There are three UIMS control architectures in existence [112]: internal, external, and mixed. With internal control UIMSs, control always resides in the application. The UIMS is viewed as a subroutine package to be called at the appropriate places in the application to perform the necessary communication with the user. In external control, the situation is reversed. The application is viewed as a set of subroutines, and the task of the UIMS is to interact with the user and invoke the appropriate application subroutines at appropriate times. A mixed control UIMS allows control to reside in both the application and the UIMS. As in external control cases, the application can be viewed as a set of subroutines to be called by the UIMS at the appropriate time. However in the mixed case, the application can take control of the dialogue and request or present data from or to the user through the UIMS. With mixed control, the UIMS and application can be co-routines or independent processes.

4.3.1 *Models*

Forms and Controls

The most commonly used model of user interface management system is using forms and controls. In fact, there does not seem to be a name for the model so calling it forms and controls is just for the purpose of this work. This model uses generic user interface controls placed on the form and thus form defines the arrangement of the controls on the screen and behavior of the controls and their interaction. It is obvious that the form is specific to its target application and in most cases it is usually built by user interface designer. A tool called data binding is often used to keep information on the form synchronized with application data. Data synchronization is very important task, because data on the form must reflect current data in the application. Data binding is able to propagate any data change on the side of form or underlying application to application or form respectively. Data binding handles functionality of client-server architecture and thus enable very good application-user interface separation. Data binding is not the only way of communication of form and the application. A system of events and their handling is often used to communicate various status and task information. The decision, whether to use data binding or event is usually during application development process. Most common scenario of using data binding is for simple data types while using events cover structures, more complex data, and commands.

Model View Controller

Model-View-Controller (MVC) [113] is a design pattern often used by applications that need the ability to maintain multiple views of the same data. MVC pattern is based on clean separation of objects and their communication and can be called as separated presentation. The separation is done into three classes [114]: model, view, and controller. The model manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller). The view manages the display of information. The controller interprets the mouse and keyboard inputs from the user, informing the model and/or the view to change as appropriate.
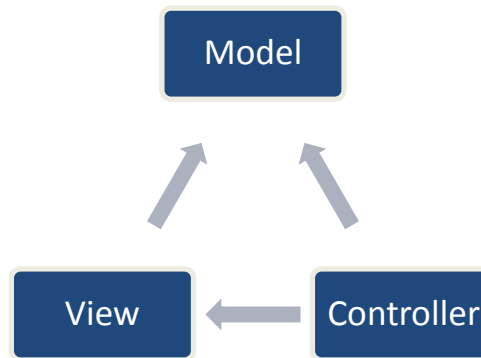
Figure 4.3 MVC class structure

It is important to note that both the view and the controller depend on the model. However, the model depends on neither the view nor the controller. This is one of the key benefits of the separation. This type of separation allows the model to be built and tested independent of the visual presentation. The separation between view and controller is secondary in many rich-client applications, and, in fact, many user interface frameworks implement the roles as one object.

Two basic variants of MVC exist, passive and active. The passive model is employed when one controller operates the model exclusively. The controller modifies the model and informs the view that the model has changed and should be refreshed. The active model is used when model can change its state without controller's involvement. It can happen when other sources are changing the data. Because informing the view from the model would break separation of the three classes, an observer object is usually used to alert objects about state changes without introducing any dependency on them. When a model changes, it iterates through all registered observers and notifies them about the change.

Model View Presenter

Above mentioned forms and controls model provides an easy to understand concept and makes a good separation between reusable user interface elements and application code. MVC has strong separation of application code and its user interface. Model-View-Presenter (MVP) can be seen as a step forward taking the best from the two above mentioned models [115]. View is treated as structure of user interface elements that corresponds to the controls of the Forms and Controls model. It does not contain any behavior that describes how user interface elements react to user interaction. The active reaction to user actions lives in a separate presenter object. The fundamental handlers for the user actions still exist in the user interface

elements, but these handlers merely pass control to the presenter. The presenter then decides how to react to the event. As the presenter updates the model, the view is updated through the same observer synchronization approach that MVC uses.
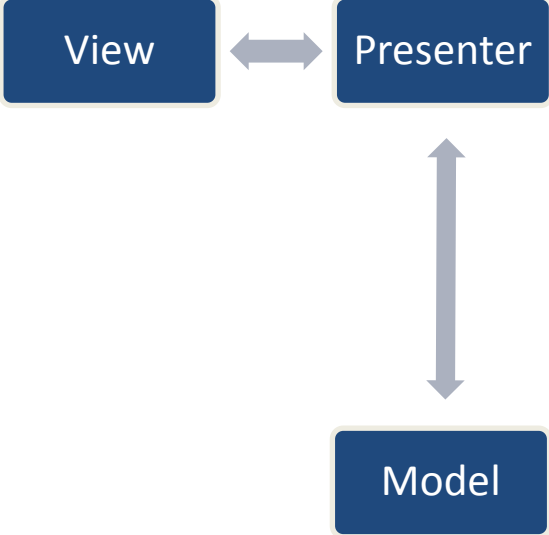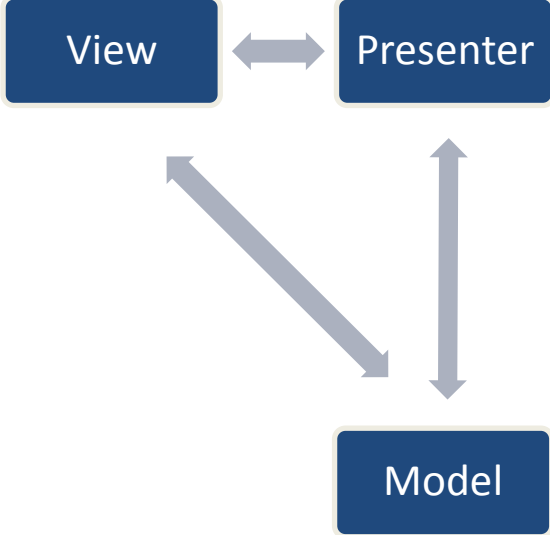

Figure 4.4 MPV (Passive View)   **Figure 4.5 MPV (Supervising Controller)**

View updates can be handled in several ways. The MVP variants, Passive View and Supervising Controller, specify different approaches to implementing view updates. In Passive View, the presenter updates the view to reflect changes in the model. The interaction with the model is handled exclusively by the presenter; the view is not aware of changes in the model. In Supervising Controller, the view interacts directly with the model to perform simple data-binding that can be defined declaratively, without presenter intervention. The presenter updates the model and manipulates the state of the view only in cases where complex UI logic that cannot be specified declaratively is required.

5 Annotation and description

There are several approaches that were developed to separate user interface from application code. With a separated application code and having the separation concepts in mind it can be said that it is possible to automate creation of user interface, because with separated application code, independent user interface can be created to communicate with application code. It is also clear that creation of the user interface is something complicated and cannot be done easily. Information contained directly in application code is not sufficient to create fully functional user interface automatically. Of course, it depends on application complexness. Simple application that would control light switching could be generally simple and thus user interface could consist of few checkboxes to switch on or off the lights. To provide additional information, for the automating of user interface creation process, it is important to describe the data and the functionality of the application.

5.1 User interface description

Creating user interface from its abstract description is one of the first thoughts that most of the developers have in mind when speaking about automating user interface generation. Although generating user interface from user interface description is not fully automated, there are still some advantages as mentioned in chapters above. Such systems, generating user interface from user interface description, can benefit on various platforms with different capabilities, because in such environments, user interface can be made to fit capabilities of the executing platform. Descriptions can be stored in various forms such as in XML-based description XAML, XIML [116], or AUI [117], or in a custom text [118] or binary form (ProcSee [119]), and parsed during execution. The description can also be stored directly in the application like in SUPPLE [10][120]. Main benefits of textual or xml form are in possibility to change user interface to the needs of each particular user. Such modifications would be more occasional and most probably done by enthusiasts. The main disadvantage of such solution is in need to describe user interface in another language – from this point of view, creation of user interface is semiautomatic. In the case of ProcSee, creation of user interface is done by user interface designer, who creates user interface based on requirements and available libraries, so it is not automatic at all. Supple provides some remarkable features to create user interface from functional description. However, functional description leads into mixing application code with instances of descriptive objects, referencing application objects and their properties. Such solution requires well designed description objects to be able to describe basic and

complex structures and in the form of maintainability creation of larger projects is difficult.

5.2 Data characterization

Before automatic creation of user interface, it is necessary to understand which of the properties of the information to be visualized are related to user interface design and how they are related. That is why a process called data characterization was proposed [121][122]. With this knowledge it is possible to choose appropriate user interface elements and construct user interface based on user preferences (e.g. blind user would prefer voice commands from mouse input).

First data characterizations were used mostly to guide the automatic design of visual presentations [123]. Data characterization method was used to describe quantitative data such as tables and graphs. Later, Roth and Mattis [121] extended data characterization set to describe more complex data types. They also analyzed data properties based on a user's seeking goals. These data characterization approaches have been often used for 2D presentation systems with static data environment. This means the data characterization approaches did not take into account data properties related to dynamic displays with changing values or multidimensional environment. This approach, however, was not lucky because, considering current applications, there i data that is not static and may change or varies in time. To allow multimedia presentations in dynamic environment, a taxonomy describing qualitative properties of the data has been proposed. The taxonomy introduced a vocabulary for characterizing multimedia information dividing data into categories such as shape, location, and structure [124]. The data categories were during last few years tailored to contain all relevant information allowing characterize any heterogeneous data [122].

Because there was need to characterize mainly database data, besides the database each application, working with the database has to provide a data characterization, which is, as mentioned above, description of the semantic and structural properties of its information that are relevant to presentation design. A nice example of how data characterization can work is in MedAide [125], which can automatically generate multimedia summaries of patient data, obtaining coordinated text, speech and graphics.
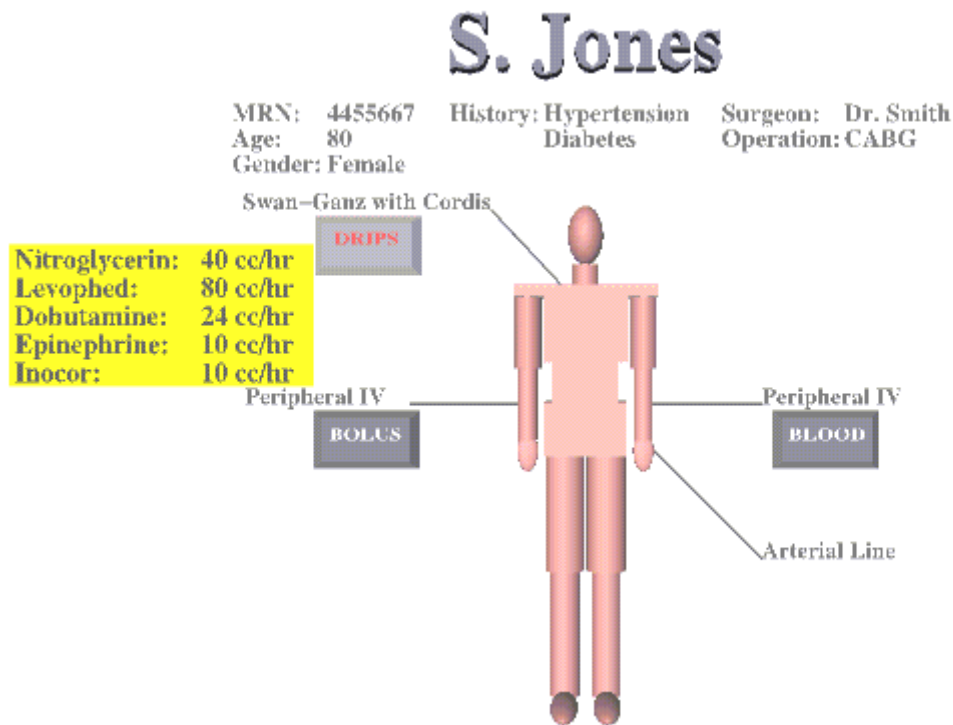
Figure 5.1 MedAide displays patient medical records [125]

The project focuses on automated user interface presentation and from this point of view it provides all expected functionality. This, however, is not sufficient in today's user interfaces, where more interactive environment is required. Although data characterization is required for proper user interface creation, it does not provide sufficient information to create rich user interfaces with complex user input.

5.3 Code characterization

Characterization of code seems to be a complement of data characterization. Characterized data together with characterized code enable fully automated user interface generation with minimal mixing of application code. Unfortunately, there does not seem to be any research in the code characterization and most researchers focus themselves on data characterization or other user interface description methods or approaches. The reason might be in bad support from the side of available programming languages or existing frameworks that do not enable searching libraries, packages, or assemblies for existing objects and their properties or attributes. Since Java [126] and .NET Framework support reflection, performing code characterization becomes possible. Thanks to the reflection, it is possible to search through packages or assemblies, create instances of publicly visible objects, know their methods with parameters, properties, etc.

Using semantic information from code characterization can reveal many features of user interface. Stand-alone data characterization describes data and their relation but is unable to describe functions and actions that can be done over the data. In fact, it is possible to define set of actions and functions that can be done with the data. This, however, is not lucky because that would mean there is a kind of huge library that offers many methods for loading of different kind of data, saving different kind of data, showing images, playing sound effects and music, movie clips and works with other data types. This could also be contra-productive because some professionals are using special applications with specific algorithms that can do similar tasks but with different quality and speed.

Combining data and code characterization is obviously very important to achieve automated creation of rich user interfaces. Semantic meaning of data and description of operations over the data and adjacent objects is a key to support required functionality of modern applications. That is why this work is focused on creation of data and code characterization taxonomy. Together with other already existing and newly proposed concepts it is proposing new way of application description for automated user interface creation.

The presented data characterization taxonomy is based on [123], where was presented first approach for data characterization. Later, the work was extended in [121] and [127]. In [124] was introduced rough partitioning of the data into several categories. This partitioning was modified and extended in [122] and this work adopts this taxonomy with several modifications.

6 Characterization

This chapter describes proposed taxonomy for data and code characterization. Taxonomy is described in an abstract form, because specific implementation for various languages or frameworks may vary. The presented examples are presented in abstract form or in C# because it is well readable even for people that do not have any experiences with programming. Support of attributes in C# also provides nice form of approach for characterization.

6.1 Data characterization

Before any specification of data characterization taxonomy, it is important to define how data is represented. Every piece of information is represented by object. Every numerical value, structure or set is an object. Every object has its data characterization as is described below. This approach is advantageous because representing information and its specification as objects can be easily integrated using object-oriented design.
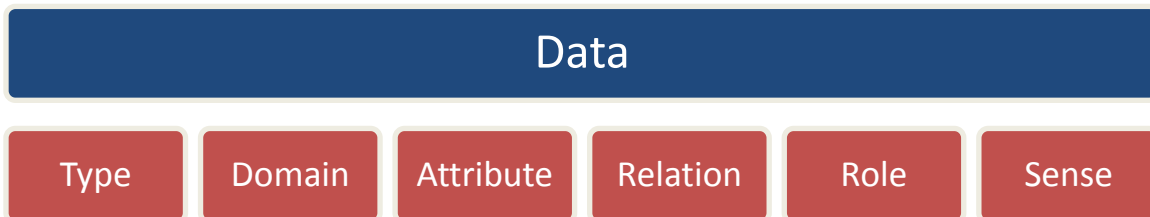
Figure 6.1 Data characterization categories

The data is described using several basic categories [122] – type, domain, attributes, relations, role, and sense. Data type is used to identify atomicity of information, whether it is divisible into smaller parts. Domain categorizes information in a semantic way, for example, whether it is abstract or real. Attributes describe objects properties such as shape, color, size etc. Relations describe connections among the data, which is important in database systems. Role characterizes data based on user information-seeking goals and sense is used to describe data for user visualization preference optimization.

6.1.1 Type

Data type is used to define basic representation of the data in the sense of its complexity. Main subdivision of data types is to atomic and composite.

- *Atomic* – Atomic objects are the most basic data units. In user interface, atomic object

is a single object. It cannot be further subdivided. In a real-time strategy game, a group of vehicles can be displayed and also used by a player as a single entity that does not consist of other subcomponents. On the other hand, a car race game can define car to be complex structure with subcomponents such as engine, transmission, wheels etc.

- *Composite* – Composite object consists of atomic and composite objects. For example, a previously mentioned group of vehicles can be composite object consisting of atomic or composite vehicles. However, simple definition of composite object is not sufficient. Group of vehicles can be just a group with no other relations but it can also be unit with its relations and hierarchy. For this reason, a further specification of composite objects is needed. A basic category types to solve the issue are set and structure.

  - *Set* - A set is a composite object, in which each component is considered only in relation to the set without taking into account relationships between other items of the set. Set composite objects are often displayed as lists or tables. Having items in lists or tables usually requires a kind of sorting. That is why the taxonomy should define an ordering technique to enable appropriate information retrieval.

  - *Structure* – A structure is a composite object, in which relations are important and must be taken into account. Thus, there are two types of relationships within a structure: relationships between the structure and its items (as in a set), and relationships among the items in the structure. A typical example of such relations can be in family tree. Each of the items in the structure is family member (relation item – structure), some of the members are children and have parents (relation item – item), and some of the members might be mother and father (relation item – item). It is now clear that presenting such family tree structure is not simple task and data and their relations inside structure must be taken into account.

Knowledge of the different characteristics of composite objects can facilitate the construction of interface elements to present them. Usually, the preferred representation for the structural relations among objects has structural properties based on those of the data. For example, a set might be presented using a table or list, but typically not a flow chart.
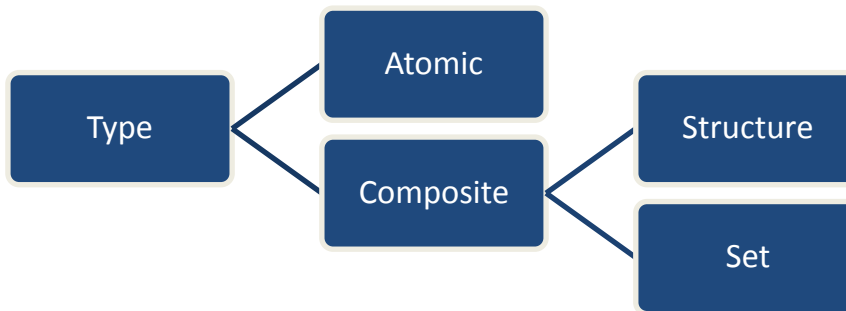
Figure 6.2 Data type

6.1.2 *Domain*

Data domain distinguishes among physical, abstract or other objects and is important for proper visualization technique choosing. Data domain has three top level categories which can be further subdivided.

- *Entity* – An entity is an object that exists independently and has unique identity. An entity can correspond to physical object in a real world (e.g. vehicle) or might be depicted as a physical object (e.g. fairy tale object).

- *Concept* – A concept is an abstract idea that either exists independently or must be attached to other objects. Unlike entity, concept is not physical object. For example, "weight" is concept that can exist independently, but when it is connected to physical object, it is describing how heavy the object is. Concept can be weight, age, speed, temperature and other.

- *Measurement* – A measurement is numeric or non-numeric value with or without its unit of measure. Measurement cannot exist alone but must be connected to a concept, such as above mentioned weight. For example, it is possible to specify object's weight as 100kg, but also as heavy. Using measurements is designed specifically for databases. Measurement represents certain value in the database while concept describes its meaning.
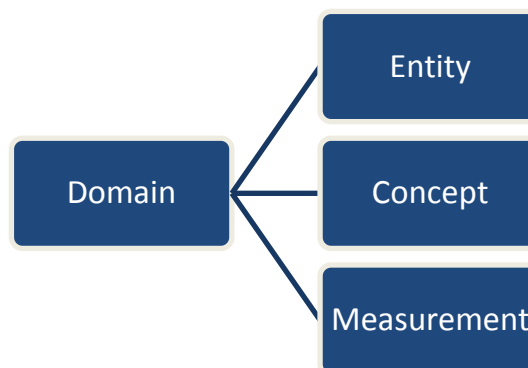
Figure 6.3 Data domain

6.1.3 *Attribute*

Goal of the attributes is to generalize aspects of presentation style requirements and abstract them to any information. Assigning attributes to data helps in choosing right presentation style as it can define some important properties. For example displaying a list of towns where each of the towns has name and location. It is possible to relate the list to image of a map so that towns will be displayed on the map. Towns on the map can be represented by small dot or city icon. Also, dimensions of the map can be scaled so that its dimensions do not correspond to locations of the towns. Attributes help in such cases to specify some extra information for better information presentation. Attributes can be common for all data types, but there are also attributes that are related to composite types only, specifically to sets.

Common

- *Location* affects the decision as where to display the object. It can be either spatial or temporal. Spatial location is represented by absolute or relative coordinates in 2D in 3D or by absolute or relative qualitative locations such as in front of or on the left. Temporal location can be represented similarly as absolute or relative with qualitative or quantitative values.
- *Material* is used to describe visual and non-visual properties of the objects such as attributes for lighting, shading and other.
- *Form* is related to object's shape and can be shaped, shapeless or none. Shaped objects have solid physical shapes such as vehicles. Shapeless objects might have a shape under some circumstances but it is so transitive that it is hard to describe its precise shape (e.g. water). All conceptual objects use none as their form, because they do not have predetermined visualization or symbol.
- *Transience* describes whether information varies in time. Static objects remain constant while dynamic objects change over time. When visualizing such dynamic objects, an animation can be used to display information change.
- *Importance* identifies information that is more important than other. In the real world, some information is really more important than other and usually is well marked. Information with higher importance can be visually marked or grouped if there are multiple important objects.

Composite

- Scalability differentiates two types of data – coordinates and amounts. Coordinates

specify points in some domain (e.g. start time, end time of some task) while amounts specify measurement to describe particular attribute of some object (e.g. length of task in hours). This distinction is important because representation of coordinates is always different from amounts.

- Continuity is used to describe quantitative data, whether they are continuous or discrete [127]. For example, set of coordinates can be continuous (GPS position) or discrete (start time, end time). On the other hand, set of amounts can be also continuous (measured temperatures during day) or discrete (days in week).

- Ordering can be characterized as quantitative, ordinal, or nominal:

  o In *quantitative* sets, elements are ordered numerically (e.g., the set of euro amounts ranging from €0.01 to €100.00. Knowing that a set is quantitative tells a system that the elements can be displayed effectively by a graphical technique with a quantitatively varying visual dimension, such as position along an axis, angles in a pie chart, the areas of circles, or the brightness of gray-scale levels.

  o In *ordinal* sets, element ordering is important to the semantics of the set, as in ratings: {poor, fair, good, excellent}. In contrast to quantitative sets, presentation of ordinal sets requires techniques which can enumerate explicitly every element (because it is not possible to interpolate intermediate values).

  o In *nominal* sets, elements are not ordered (for example the set of brands). These can be expressed effectively using symbols, which don't vary quantitatively. In contrast to quantitative sets, nominal sets may be misperceived as ordered when using quantitative techniques for their presentation.
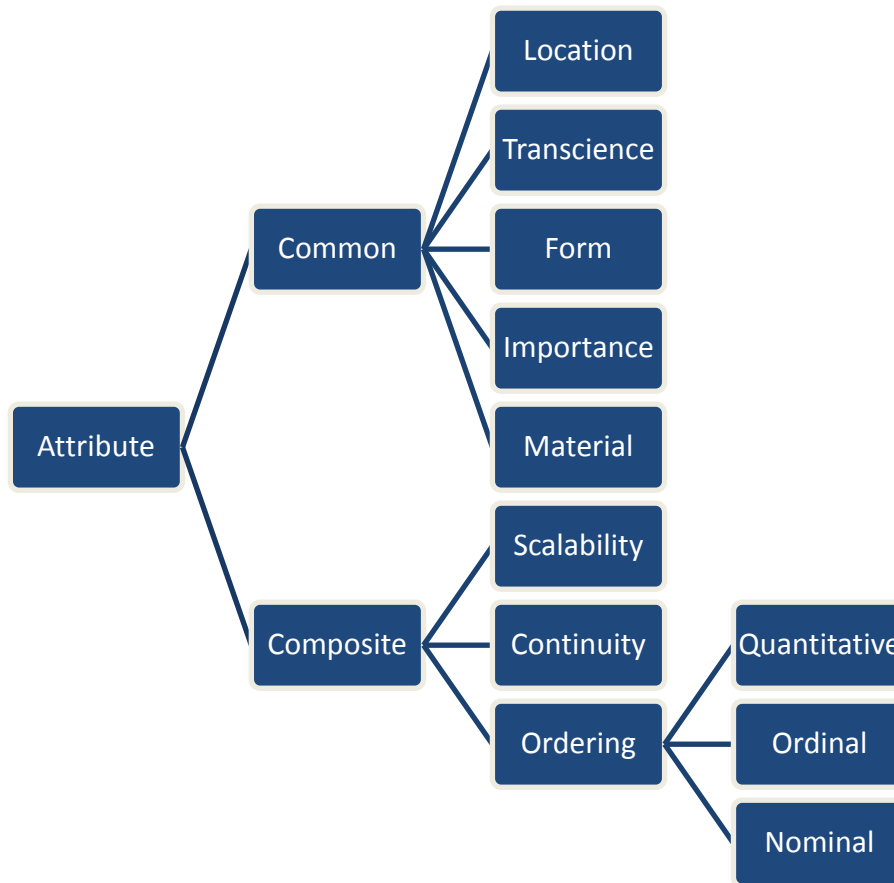
Figure 6.4 Attributes

6.1.4 *Relation*

Relations are used to capture various types of relationships among objects. Basic scope of data relations mentioned in [121] was extended at [122] to include semantic relations among data. For example, if it's required to visualize set of cities or towns and its number of inhabitants, it is important to know functional dependency relation between towns and the numerical values of their inhabitants, as well as attributive relation between the inhabitants and the towns. Four basic relations are: functional dependency, constituency, attributive relation and enumeration.

- *Functional dependency* is used to capture one-to-one mapping relationship from one set to another. For example, every town has number of inhabitants. This type of relation is used by certain visualization techniques (e.g. a line graph) [121][123].

- *Constituency* corresponds to the has-part relation and can be further partitioned into physical constituency and conceptual constituency. Physical constituency either indicates the physical components of a physical object (e.g., a network node has several ports among its physical constituents) or refers to conceptual component that can be treated like physical ones (e.g., virtual links). Conceptual constituency, on the

other hand, implies that concept is conveyed by a set of sub-concepts (e.g., town is conveyed by age, inhabitants, and other sub concepts).

▪ In an *attributive relation*, some objects describe certain aspects (attributes) of other objects. For example, in a computer network, a virtual path segment is a virtual connection specified within a physical network link, and has attributes such as capacity and utilization. Furthermore, these relations can be effectively encoded by the corresponding image's visual attributes, such as its shape, size, or color.

▪ *Enumerative relation* shows class inclusion (e.g. instance relations, kind or type relations – hatchback is a vehicle) or class membership (e.g. Renault Clio is instance of hatchback). Ideally, to convey information comprehensively, a generic concept representing classification is shown first (e.g. button Hatchbacks), and then all the instances of specific subclasses of the concept are enumerated in detail.
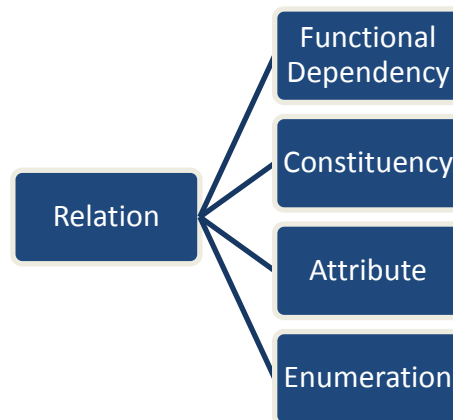
Figure 6.5 Relations

All the relations discussed above usually do not exist independently from each other. More often, one or more types of relations coexist in one piece of information. If there is request to display several relations together, it is necessary to take into account the interconnections between those relations in order to integrate all related information in one presentation. Understanding complex relations enables to perceive both explicit and implicit connections among data, and guides us to select the best possible graphical techniques to convey the information.

Relation Attributes

Attributes of relation are described by dimensionality [127], coverage, cardinality, and uniqueness [121].

- *Dimensionality* describes decomposability of information into simpler components. Some information, for example database record, can be decomposed as a vector of simple components, others, such as image, have a complex internal structure which is not decomposable. Thus dimensionality can be:
    - *Simple* – Atomic items of information, such as an indication of presence of unread mail.
    - *Single* – The value of some measurement, such as the amount of gasoline left.
    - *Double* – Pairs of information, such as coordinates, or domain range pairs.
    - *Multiple* – More complex information structures of higher dimension, such as home address.
    - *Complex* – Information with internal structure that is not decomposable, such as images.
- *Coverage* conveys whether every element of a set can be mapped to at least one element of another set. This is important because some visualization techniques have no way to express the absence of data without adding a special marks or leaving a blank space, which can be misperceived, for example as low value.
- *Cardinality* expresses the number of elements of a set, to which a relation can map from an element of another set. In other words, it expresses the number of values that can occur for an attribute of an object. For example the has-part relation maps from one element to a variable number of others. Cardinality should not be confused with the concept of arity [128], which refers to the number of different domains in a relation.
- *Uniqueness* refers to whether a relation maps to a unique value or values for each element of a set. For example has-part relation can map uniquely to set of unique parts (i.e. parts have only one parent).
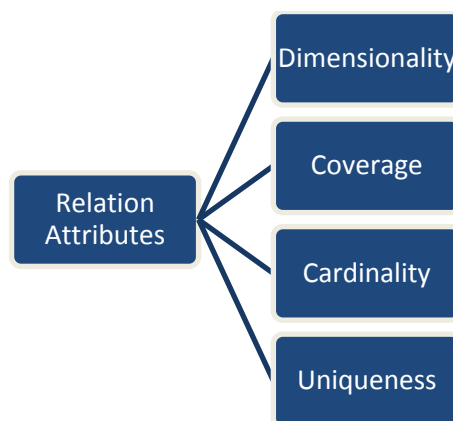

Figure 6.6 Relation Attributes

6.1.5 *Role*

Data role specifies particular information seeking tasks. It can be given by a user during interaction, or could be expressed by domain experts during knowledge acquisition. Affecting presentation design by user's information seeking goals is discussed in [129], while [121] also analyze data properties based on the user's information seeking goals. In [122] is adopted the taxonomy of goals proposed in [124], which is a superset of functions listed in [121]. These tasks include:

- *categorization* (categorizing information)
- *clustering* (grouping information)
- *identification* (revealing information identity)
- *distinguishing* (revealing the difference among information)
- *comparison* (comparing one type of information to another)
- *association* (associating one type of information to another)
- *ranking* (comparing all information in a particular order)
- *correlation* (correlating one type of information to another)
- *distribution* (partitioning information)

For example, towns in a set of towns can be distinguished by their name.
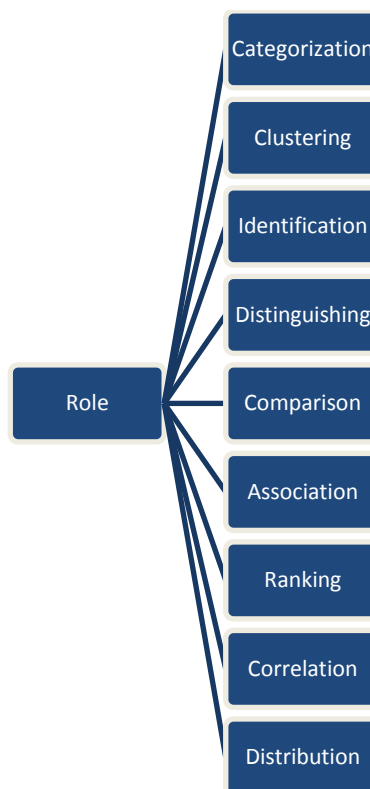

Figure 6.7 Role

71

6.1.6 *Sense*

Data sense was proposed to support construction of effective visual presentation because user's, or application's, visual preference should also be taken into account [122]. For example, one way to distinguish the elements of a small set of objects is to use a different color for each. However, such a presentation would not be effective for a color-blind user. Presentation preferences might also be dependent on a specific application domain because users in that domain have been trained to use a particular presentation style.

The word "sense" can be used to refer to one of a set of meanings for a word or phrase, as in a dictionary. For example, the word "person" can mean a "human being" in one sense, and "human body" in another. This meaning of sense was adopted to coin the term "data sense", which signifies a preferred way to present the data visually.

Like data role, data sense can be specified by an end user to indicate his individual preference, or captured in the process of knowledge acquisition. Alternatively, data sense might be inferred, base on other data characteristics. Data sense can be extended to include a wide variety of low-level visual preferences. In [122] is data sense limited only to five high-level presentation styles described below:

- *Label* – A textual label is used to display information. A label can be enriched by adding graphics, such as underscoring or button shape to convey additional information.
- *List* – In contrast to the label sense, the list sense states that a composite object should be displayed in a tabular form in which all its components are listed textually as table entries. A typical list representation is the results of the exam (every student has rating). Furthermore, in the list sense, all components of composite objects are either listed as an attribute-value pair (e.g. additional sub-task rating) or as a textual string (e.g. rating comments).
- *Plot* – Both plot sense and list sense indicate that there is a composite object and that all its components need to be represented. The major difference between plot sense and list sense is in the primitives used in the representation. In the list sense, information is represented textually, while graphical primitives such as markers and lines are employed to encode different kinds of information in plot sense. Quantitative information can usually be effectively expressed in plot sense.

- *Symbol* – Representing a piece of information in symbol sense involves the use of a concrete shape object to symbolize the information. For example, a thermometer icon with its scale can be used to symbolize the temperature concept. Both physical and conceptual objects can be represented in symbol sense. For example, a town can be symbolized by town photo or abstract symbol representing the town.
- *Portrait* – Compared to symbol sense, visual representations require much more detail and precision when the information needs to be displayed as a realistic. Such precise and detailed visual representation is necessary for people to carry out certain tasks, such as the design of a product's physical appearance.
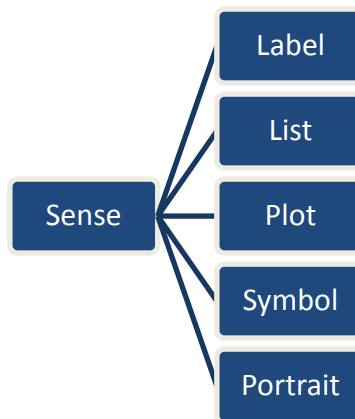
Figure 6.8 Sense

6.1.7 *Example*

This example demonstrates how a piece of information can be characterized for automated user interface generation process using aforementioned taxonomy. An object "town" contains information about its geographical location, name, number of inhabitants, and heraldry sign. The information about town is a composite set (data type) entity (data domain) specified to be represented symbolically (data sense) using heraldry sign. It has a location (data attribute) specified in terms of geographical location. The town's ordering attribute is quantitative, which specifies alphabetical order among other towns. The town has conceptual constituency relation with its location, name, number of inhabitants, and heraldry sign. Number of inhabitants is atomic concept, represented as label, with clustering role to allow searching all towns with same number of inhabitants, quantitative ordering. The town name is atomic concept, represented as label, with high importance attribute, quantitative ordering attribute to allow alphabetical ordering, and identification data role to identify the town using the name. The heraldry sign is atomic concept, represented as portrait, with identification data role to identify the town using the sign.

Set of towns will be visualized on a picture because of its location attribute, represented by heraldry sign picture and name label. Towns can be listed using name and sorted by number of inhabitants or name.

It is obvious that data characterization concept is required for proper user interface generation because it can add useful semantic information about the data and its relations. This information is even more important in case of database records, where relation and structuring in tables is not as well defined as in object oriented programming languages. In object oriented programming languages classes, structures, unions, and other objects contains some information that already conveys some information inherently.

6.2 Code characterization

Code characterization is designed to describe code of the application. Because it is not possible to detect what methods or functions will play any role in the final user interface, it is important to describe all involved methods and functions and their parameters so that user interface generator can take all the data and operations into consideration when generating user interface. Taxonomy in this chapter is divided into several categories. Each category contains set of annotation tags that are designed to describe particular aspects of methods or functions in the application code.
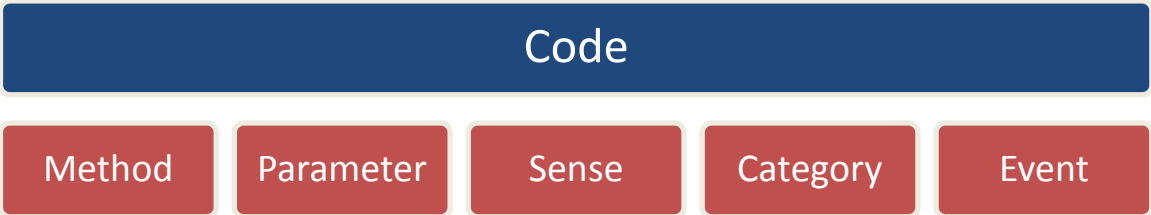
Code				
Method	Parameter	Sense	Category	Event

Figure 6.9 Code characterization categories

Before any specification of code characterization taxonomy, it is important to define how methods and functions are represented. Every piece of information is object that has its properties and methods. Methods can perform operations with the object - depending on their functionality. Because objects can be further divided into smaller components – sub-objects – which are accessible as properties, it is possible to create complex hierarchy of specialized methods that work with object and are grouped into logical groups. All of the public methods, that are visible outside of the objects, can be characterized using this taxonomy.

6.2.1 *Method attributes*

Attributes express various information about methods that are implemented by the objects. Although it might seem that information according to each of the methods are very different, they have quite lots of common attributes that need to be defined for proper creation of user interface. If some object implements a method that should participate in user interface, basic attribute "Name" should be defined.

- *Name* attribute contains name of the method that should be presented to user in user interface. In fact, name attribute is identifying attribute which means that every public method of the object, characterized with the name attribute, is processed and contained within the user interface. Name is not required for methods that already have its category, because it already has name attribute. Name attribute can also be specified in multiple languages so that user can choose which of the languages he prefers.

- *Description* attribute contains information, describing functionality of the command. As well as name attribute, it is represented by string. Description can be used in cases where user requires more information about the function of the command. Description should not be confused with name, because name is also used as tool-tips because it represents the name of the command. Description on the other hand can be used when generating help contents for the application because it can contain more information about required parameters etc. Description is optional attribute and does not have to be specified in order of proper user interface generation process.

- *Importance* identifies commands that are more important than others. Objects, that are used more often than others should be easily accessible to enable effective usage. In graphical user interfaces important commands are placed in menus or toolbars such way, that it is quickly accessible and has simple shortcuts. Definition of important commands is usually problematic because it depends on many factors – what functions will be used by most of the users etc. It is also possible to alter the importance of commands during runtime so that importance of commands can change based on user's application usage, leading in tailoring user interface to fit most often used commands by particular user. A typical example of commands with high importance can be a line tool in vector image editing software because lines are one of the most often drawn vector primitives.

- *Representation* defines a symbol that represents certain commands. Representation is especially important for commands that have high importance. Representation can be

defined as icon or image. For commands that belong to a category, representation is most usually defined in category definition file. In this case, another definition of representation overrides default representation.

- *Dependency* – Every method in the code has some dependencies that must be satisfied before the code is executed. This is most usually checked by developer programming user interface. In automatic generation of user interface, the user interface has to know when command is enabled or disabled. So dependency expresses conditions which have to be satisfied to allow execution of selected command. Dependency is very important because without proper dependence checking, user can have access to unavailable commands. An example can be playing of media file although there is no media file open. By default, if dependency attribute is not defined, the command is always available to the user and user can execute the command at any time. Dependency attributes can be defined multiple times to reflect various rules. Based on the implementation of dependency attribute, rule can be defined as regular expression or it can be a method name, which can return flag signaling whether all conditions of rule were met. Dependency attribute is one of the key attributes that controls availability of commands in user interface. For example, an application has menu item called "play" to play multimedia file. However menu item must be disabled as long as there is not multimedia file loaded. The first rule can check if there is already loaded multimedia file.

- A *modal* operation is operation that requires the user to interact with it before he can return to operating the parent object who executed the operation, thus preventing the workflow on the object or whole application. In visual user interfaces, modal operations are represented by modal windows and are often called heavy windows or modal dialogs because the window is often used to display a dialog box. Because it is not possible to recognize which method of particular object should or should not be modal, it is important to mark the method as modal. For example, application with open command requires as parameter information about file. In case the performing method is characterized as modal, application presents user a dialog with file browsing capability. User cannot switch to other application content as long as file is selected and confirmed. The open method receives file information parameter and performs file opening. In case that performing method is not characterized as modal, application presents user a dialog with file browsing capability. User can switch to application content and select open command repeatedly, showing multiple dialogs with file

browsing capability. Modal operations are important to keep application workflow. Frequent uses of modal operations include:

- o drawing attention to vital pieces of information
- o blocking the application flow until information required to continue is entered, as for example a password in a login process
- o collecting application configuration options in a centralized dialog – in such cases, typically the changes are applied upon closing the dialog, and access to the application is disabled while the edits are being made
- o warning that the effects of the current action are not reversible

- *Dialog* – To support blocking behavior for warnings and questions before operation can be performed, it is necessary to present dialog with question or information to the user who can decide whether the operation should or should not be performed. Some examples of such behavior were described in previous paragraph. However because it is important to present dialog to the user before the operation is executed, additional information must be specified to enable such behavior. For this reason a dialog attribute was defined. Dialog attribute informs user interface generation process that it should display a dialog before the method is executed. Dialog contains information about type of reply (yes or no, ok or cancel) and indication at what reply should be method executed. For example a method called "close" performing closing the application defines dialog attribute with question whether user wants to close the application and indication that it should be executed when user replies yes. When user tries to close the application, user interface presents modal dialog with the question and in case user replies yes, method is executed and application is closed. In the later case, when user replies no, method is not executed.
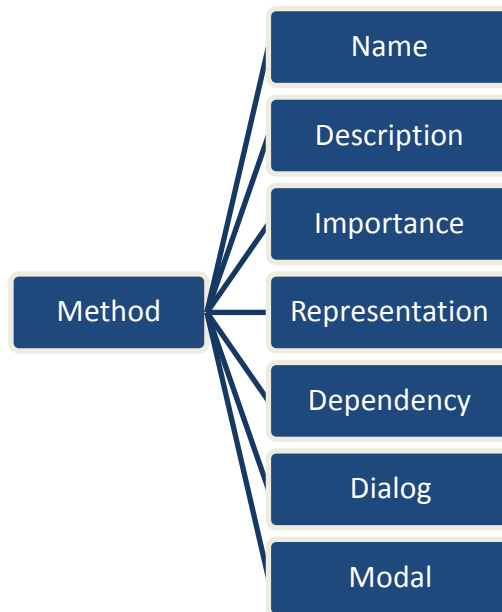
Figure 6.10 Method Attributes

6.2.2 *Parameter Attributes*

Almost every implemented method has some parameters. Parameters init method and method most usually works with available parameters or internal variables. That is why its characterization is very important. Following parameters are basic parameters that are important for proper calling of selected method. An example can be drawing into an image. A method that can draw a line has three main parameters: color, starting point and ending point. User interface based on parameter characterization knows that the form, which is displaying the image, has a color palette to which is related color in draw line method. User interface will not require user to specify this parameter manually and take the value from the color palette. Because points are data characterized and related to image, user interface knows that the location should be taken from the displayed image and when user selects draw line command, user interface expects user to select two points in the area. After both points are selected, command is executed. Following parameters are important for proper parameter characterization.

- *Name* - Describes name of the parameter and is similar to name in method attribute. Name is not always required but is very important in user interfaces that are not based on visual interfaces (e.g. for blind people). As well as name of the method, name can be used for creating dialogues of what information command requires from the user. Name can also be used when generating help or documentation,

- *Description* attribute describes what kind of value is required for the parameter. As well as name attribute, it is represented as a string. Description can be used to define

78

parameter value more precisely by showing tips and describing value that is required. Description should not be confused with name, because name can be used as a label or other parameter descriptor in user interface. Description, on the other hand, can be used when generating help contents for the application because it can contain more information about parameter and its relations. Description is optional attribute and does not have to be specified in order of proper user interface generation process.

- *Relation* is important for the parameters because it describes relation between parameter and another object in environment. When relation is specified, user interface automatically takes result of relation as input for the parameter. An example is color from color palette. Drawing methods require color which is taken from color palette automatically. If there is no relation specified, user interface creates new color selection dialog just for the purpose of getting required information.

- *Default value* – Some parameters, most usually quantitative, require certain value. Although user can change the value, sometimes it is more convenient to show user some kind of default value that can be used for the command too. An example is number of passes of some computational task. Although user can specify much greater value, algorithm can have good results in five passes. Default value can tell user what values are optimal. In case a relation attribute is specified, default value might or might not be used, depending on the type of data and relation.

- *Maximum and Minimum Value* – It is possible to define maximum and minimum value as a boundary around default value. In such cases, user interface control checks if the value, specified by user is in valid boundaries.
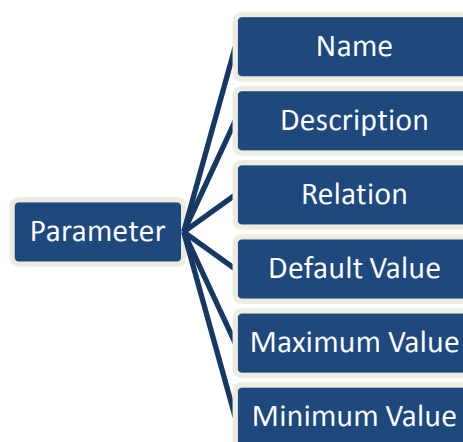

Figure 6.11 Parameter Attributes

6.2.3 *Sense*

Sense helps to distinguish how are methods executed. By default, user selects command, user interface asks for parameters and method is executed with all specified parameters. Such default sense is called *command*. When *tool* sense is specified, user interface runs method again and again as long as the command in context is selected. Typical example is selecting some tool – e.g. a draw line tool – from a toolbox. As long as user keeps up with specifying points in the image and as long as the tool is selected, user interface calls draw line command repeatedly and user draws lines. Command and tool sense cover most of the types of method executions and that is why various types of user interfaces can be generated.

For some objects there also exists *default action* that can be executed whenever user selects object and launches default action. In graphical user interfaces this is most usually done by double-clicking. For blind users, there is usually voice command run or open. In this case, it is possible to specify which command is default and should be used in such cases.
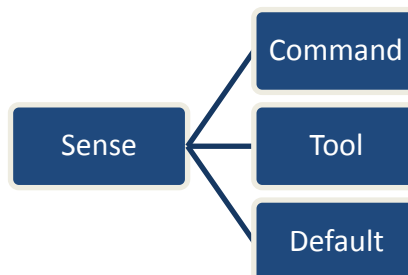

Figure 6.12 Sense

6.2.4 *Category*

Command category defines set of operations or commands that can be executed on the object. A typical example of command category can be set of methods that allow playing, stopping, pausing of recorded data (might be audio or video playback, or any other data that object can replay or record). Set of such commands have usually default symbolic representation and users got used to the concept from a real world or other applications. That is what command categories are designed for – keep the concept of control from real world to be consistent. Command categories can be easily defined in XML format and extend existing set of categories. Categories can be represented in user interface using concept of smart-templates [130] which contain information about default symbolic representation and also default graphical design. Main benefit of smart-templates is its platform independence and ability to

adapt to devices capabilities and properties. Because object can be of various categories, multiple categories can be defined for a single object. Following categories are basic for most applications and should always be implemented.

Collection

One of the most often used components in user interfaces are collections or lists because they can contain multiple items and using lists is natural for every user. Every human is used to sort any information or objects and store them together based on their purpose. For example, photo album contains photos that were shot at the same event; shops have goods of similar type in their shelves so that customers can quickly find goods of the same type but different brand etc. All these items can be effectively presented in a list user control. Most of the lists, or collections, contain markers that can be used to highlight selected item, or multiple selected items, to allow user to control the content of the list, modify it, add or remove items etc. Data characterization process allowed creation of list user control but with very limited functionality. Because there is no additional information about what can be done with the items in the list, it was impossible to define proper operations over the data in the list itself. For example, in a playlist of a multimedia player there are multiple items in the list, representing media files with properties such as author, track name, album name etc. In the list user interface control with data characterization only, adding new item would mean creation of structure or database record with new item, asking user for author, track name, album name etc. Playlist is, however, built on media files, showing their properties rather than list of properties without any relation to the real file. Characterizing method for adding item to the list, taking as a parameter media file can solve the problem and allows creation of dependent operation that can be performed in certain conditions with wide range of options. Characterized add method is also one of the default list methods and can be presented in a consistent manner nearby the list.
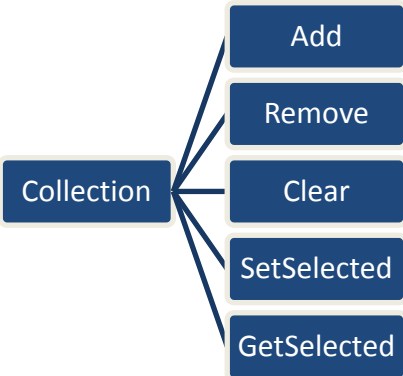

Figure 6.13 Collection Methods

81

Basic operations over the list or collection are addition and removal, and cover most of the operations done by the user on the list of items, because user usually needs to fill the data and remove them when needed. Clear operation can be specified when there is need to remove all the items with single operation, replacing need to select all items in the list and remove them manually. Last operations on the list or collection are bound to selection. Without knowledge of selected items in the list, it would be just presentation with no other functionality. Propagating information about selected items can enrich functionality. For example, multimedia player can start playing selected media file when user executes play command.

Collection category can also be implemented on objects that are not data characterized as composite sets. If an object that is not characterized as composite set implements collection category, user interface can place commands in consistent way nearby user control, representing characterized data.

Storage

Goal of almost every application is to create new content, or save, and open already created content. With data characterization so far, it was not possible to create application that would have behavior that could enable opening and saving various content. User interfaces based on data characterization presented existing data and did not allow opening new kinds of data, because of need to have existing data ready for presentation. For example, in an image editing software, user wants to open existing image for editing. In case of data characterization, application would have a concept button, representing already populated set of all images. User would have to click the button and select the image from the list. Selected image would be shown because it is atomic entity data with portrait sense. In case of code characterization, user can execute open command that takes as a parameter file – an open file dialog is presented to allow choosing any file, even from dynamically mounted storages.

Storage methods are usually represented by common symbols (e.g. save by diskette icon, new by empty document icon) and are by default characterized with high importance. By default, storage category commands are present in the main menu and toolbar to be quickly accessible.
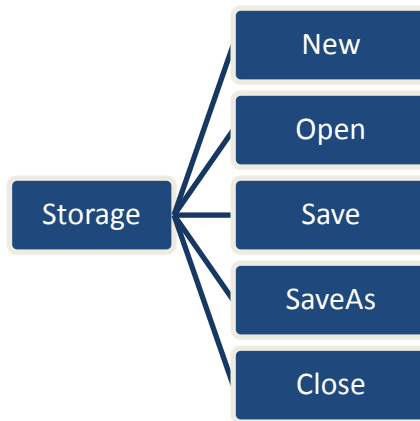
Figure 6.14 Storage Methods

Navigation

Navigation category is related to collections but can be defined separately from collection on any object that needs to support navigation functionality. Navigation describes operations that allow change of some actual, current, or selected object in the means of moving to next, previous, first or last one. Navigation should be done sequentially so that moving to next and back always follows same order. When composite set object has code characterized navigation methods, user interface contains controls bound to the presented collection, because methods of the object should work over the data contained within the collection. In case of atomic object, user interface creates controls independently and functionality is dependent on implementation of the navigation methods. For example, multimedia player has a playlist with multiple media files. Multimedia player also has buttons next and previous for playing next and previous media file from the list but the implementing methods are not implemented in the playlist object but in the player object. Executing next or previous command performs change in the selection of the playlist. Playlist throws new event selection changed that is handled by the player object which, in case of playing other media file, starts playing newly selected media file. Similar example can be implementation of next and previous in an internet browser, which allows returning to previously visited pages etc.
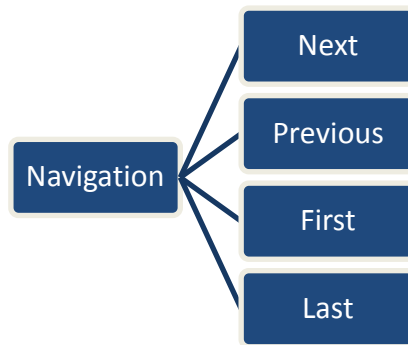
Figure 6.15 Navigation Methods

Player

This specific category is defined for the purpose of applications that need to perform operations that can be described as playing of records. For example, a multimedia player is application designed for playing of multimedia files. It contains play, stop, and pause buttons to perform control of playing selected media file. Another example can be application that is performing saving network communication during simulation (e.g. for the purpose of replaying and reviewing simulation). It contains play, stop, and pause buttons to perform playback of the simulation during review and evaluation of the simulated exercise.

Player category can have methods for start (play), stop and pause of playback. Special method is joint play and pause when executing play command when media player is playing, it leads to pausing the playback. All player methods are usually represented by standard symbols but their representation can be altered by custom symbols to represent design that fits the functionality of the application.
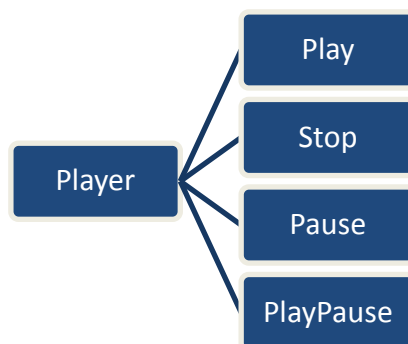

Figure 6.16 Player Methods

Clipboard

Clipboard is widely used technique for data copying and moving and should be kept consistent in all platforms. Clipboard category defines methods that can be used for this

purpose. For the copy and cut operations, it is important to define dependency of the method, at which conditions it is possible to copy or cut operation perform. The reason for this behavior is that it is not possible to define when application can copy or cut selected data. Rather, every time object changes its state or properties, dependency is reevaluated and copy and cut operations are enabled or disabled based on the dependency result. Similar behavior is for paste operation. It is hard to define what kind of data application can process and thus every time clipboard content has changed, dependency is reevaluated and paste operation is enabled or disabled based on the dependency result. This way, it is possible to control behavior of clipboard operations without knowledge of data, that application, or objects of the application can process.
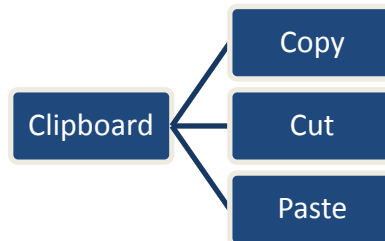

Figure 6.17 Clipboard Methods

With data characterization only, clipboard operations were possible on the presented data only. For example, remember the playlist example with multiple columns, representing author, name etc. Copy operation would copy entry from the collection and paste the entry with the data copied from the playlist without any relation to media files. In case that other application stores path to media file into clipboard, collection in data characterized presentation is unable to handle paste of such data because it is not compatible with entry structure in the collection. On the other hand, thanks to characterized code, paste operation is possible because dependency evaluation confirms ability to process such kind of data, loads header of the media file based on the path in the clipboard and adds new entry to the playlist. It is now clearer how code characterization helps with clipboard operations.

Drag-and-Drop

Drag-and-drop or DnD is the action of (or support for the action of) picking a virtual object and dragging it to a different location or onto another virtual object. In general, it can be used to invoke many kinds of actions, or create various types of associations between two abstract objects. As a feature, support for drag-and-drop is not found in all software, though it is

85

sometimes a fast and easy-to-learn technique for users to perform tasks. In case of data characterization, DnD was possible but with limited functionality for objects. In a clipboard example a problem with copy and paste operation was described. Similar problem occurs when dragging object of different type onto composite set. Drop operation cannot be handled because the data type is different and user interface cannot add new item to the collection. Code characterization defines methods handling drag and drop operations so that user interface can check whether object can handle this type of object.
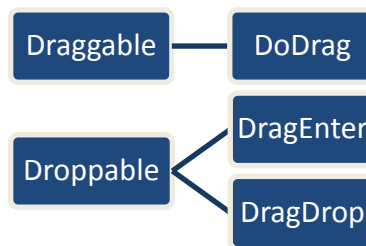

Figure 6.18 Draggable and Droppable Methods

Object with method DoDrag that belongs to draggable category informs that object can be dragged. When an object can be dragged, user interface allows user to pick the object and start dragging. Execution of DoDrag method leads to filling information about object and his type for the droppable target. Droppable target is recognized by implementing code characterized methods DragEnter and DragDrop of droppable category with logic for checking whether dragged object can be dropped, and acquiring dropped object. When dragged object is over droppable, DragEnter is called to inform droppable that there is a dragged object with certain information. Droppable evaluates format of the dragged data and informs what kind of operation can be done with the dragged data. After finishing the drop by the user, DragDrop is called and droppable gets the dragged data (just in the case that it can handle dragged data as specified in DragEnter). This way, there is no need to define explicitly what kind of objects can be dropped and all checking remains on the side of the object and its implementation.

6.2.5 *Events*

Events represent a mechanism that informs user interface or application code that there have been some changes in the data from the user or from the application code. Although various approaches exist for implementing such behavior using callbacks [130], TAPS [131] or ORB [132], event mechanism is very simple and works well for duplex communication. Besides

events, modern programming languages support property descriptors [133] which can be used to detect property changes. However, property changes are detected only when property is assigned to and not in case that property has been changed inside of the object. Main benefit of event descriptors is that there is no need for the object to register any event. Property descriptors can be registered in intermediate framework during initialization. On the other hand, registering event on the object guarantees that user interface will be informed about all changes of the object internal data.

Base event, that should be implemented for every object, is "changed". Such event should be thrown always when data of certain object have been changed. In case of data that is changing infrequently, it is important to be notified about the change so that user interface can redraw its content. For data, that have data characterization with high dynamic transience [122], user interface can check its state very often and optimize the performance. Event mechanism can be used for static data or data with low dynamic transience.

Because user interface set the data that are changed in user interface immediately to the object's properties, it is not necessary to inform object about changes in public data. However, for languages that do not support concept of properties (e.g. C++) it is necessary to implement set method that should be characterized by "userChanged". This event contains also definition for what data is event designed so that user interface can react correctly when user changes any data.

6.2.6 *Entry point*

Running application from characterized code with no entry point is little tricky. Without entry point, there is no information about what object should be created as the first instance that would be understood as root object for the user interface, from which user interface generation and user control process starts. That is why the entry point attribute is defined – to tell the generator that this is the root object for the application. This, of course, means that there has to be only one object with such attribute because in case of multiple objects behavior could be unpredictable. In case there is no object with entry point attribute, user interface generator would be unable to start the application because there would be no object to start from. This is one of the limits of current approach – however in current applications, it is usually not possible to launch different applications from single executable file and so the restriction with just one entry point is not critical.

7 Generating user interface from characterized code

A code characterized using presented taxonomy in the previous chapter can be processed by a framework that is capable of the characterization analysis associated with the code and generate the user interface on the fly. This chapter is presenting all the processes required to create the user interface in an abstract form because concrete implementations can vary among various platforms and languages. For example, in case of .NET Framework, it is possible to define custom attributes that can be used as data and code characterization tool to characterize necessary information. This information can be included in the final assembly which contains not only the code but also metadata, including type information and its attributes. Because .NET Framework supports reflection, it is possible to retrieve the type information from the assembly using the reflection at any time with standard means of the language. On the other hand, C++ does not allow creation of such output. That is why a preprocessor would be required that would have to extract characterization information from the source files and generate besides the output dynamic library another file with necessary characterization information. Because there is no standard way how to load characterization from the custom file generated by preprocessor, a code characterization factory have to be created to support such functionality. Such information file can be saved in a standard human-readable form (e.g. XML) or in a binary form. Because storing attributes as metadata in an assembly is very different from possible custom output using custom preprocessor, this work will not discuss particular implementation but will present information in an abstract or formal way.

Because generating user interface is very complex task, it is divided into several steps. First by, it is important to analyze user context. This information is used later in the process of identification of suitable user interface elements. With knowledge of contexts, code characteristics is loaded and stored in a structure representing abstract information about data and operations. Objects are then grouped using context information into containers that define what pieces of information should be kept together. Abstract information about data and operations is then transformed into particular user interface controls in a way that it meets all constraints. A final layout of controls is designed and user interface is ready for use. Because it is possible to create various user interfaces based on device capabilities, produced user interface can be designed for voice, pointer, touch or other controls as human computer interaction or user interface industry proposes novel approaches or techniques to work with

devices. Following paragraphs describe each of the user interface generation steps in more detail.

7.1 User context analysis

First step in automated user interface generation process is user context analysis. User context is any information that can be used to characterize the situation of an entity [134]. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves. Because user context can change in time, a static (e.g. used for desktop computing) and a dynamic (e.g. mobile computing) user context exists. This means that user context can change in time as user changes his attention or focus to other tasks. In such cases, user context analysis should be done regularly and user interface should be modified to suit the context of the user. For example, user is working with device equipped with touch screen. During the operation of the device user enters a car and starts driving. Because user is visually occupied by driving, device has to change modality to voice commands and audio output because user cannot check the display anymore. Thus it can be said that changing user context can lead to change in modality.

User context can be divided into two main parts, device context and human context. Division is possible because both device and user can be handled separately and also both device and user are in same space (user is using the device directly, not remotely). Thanks to same space, device can be used to gather information about the context of the user.

7.1.1 Device context

Device context represents hardware and software capabilities of the device.

Hardware

Because there is huge variety among devices that can be used by users, it is important to analyze hardware features and limits of the device. Based on the capabilities and features of the device, a proper user interface can be created. The most important hardware properties of the devices are:

- Computational power (processor, architecture)
- Display (display size, resolution, color depth)
- Memory (RAM, storage)
- Input / Output (keyboard, microphone, speaker, touch)

89

Computational power affects modality used to communicate with the user. Some user interfaces (e.g. based on video input) can require more computational power while other (e.g. touch input) require less computation power. Display parameters affect visual user interfaces – their sizes and layout because user interface should fit the display. Memory affects user interfaces that are memory intensive (e.g. video input / output, audio input / output). Input / Output type affects modality, used for interaction with the user.

Software

By software, operating system and its installed libraries and applications that are capable of various functionalities that can be important for presenting the user interface, is usually meant. Ideally, software that is capable to present the user interface to user should be operating system independent. However the situation in the real world is a bit complicated – it is not easy to produce the software that is capable to run on all available platforms. Fortunately, there are some runtime environments supporting large variety of platforms, such as Java or .NET Framework.

Although framework for automatic generation of user interface is also software, it is not the default part of the device. However, the framework contains data that are somehow connected to the device – abstract (AIOs) and concrete (CIOs) interface objects which can be considered as a part of the software. More information about AIOs and CIOs is in the next chapters.

7.1.2 *Human context*

In the contrary to the device context, the human context cannot be expressed exactly because computers, so far, are not able to understand the meaning of user actions. However it is possible to estimate what is the user willing to do. In this process of estimation, a three parts play key role: personal capabilities, environment, and modality.

Personal Capabilities

This part covers user's personal abilities and disabilities, such as visual impairment, user's working with the application and other personal settings. Because it is not possible to track user's personality, it is possible to create a set of predefined templates and apply them when needed.

Environment

Environment surrounding user plays important role when designing user interface. It is

because it can be very variable and unpredictable. Environment features can limit or support user in performing some actions, including operation of a device. For example, in noisy environment, it is more convenient to present user interface visually then using audio output, because sound could be hardly heard in noisy environment. This way environment affects modality.

Modality

This part expresses what modality is user willing to use when communicating and operating the device. Most often used modalities are visual and audible.

7.2 Parsing code characteristics

After the context analysis is done, code characteristic is being loaded. At first, all data characterized types are loaded and for each of them a description is created. Description consists of a characterized data tree and a characterized code tree. Structure with all data characterized types is called characterization tree. An example of a characterized data and code is in Figure 7.1.

```
[CodeCharacterization.FirstObjectSelected]
[DataCharacterization.DataType(DataTypes.Atomic)]
[DataCharacterization.DataDomain(DataDomains.Entity)]
[DataCharacterization.DataForm(DataShapes.Shaped)]
[CodeCharacterization.CodeName("Media Player")]
public class MediaPlayer
{
  …
}
```

Figure 7.1 Example of the characterized source code from the simple media player.

7.2.1 Data

Characterized data tree is built from public properties of the object which are characterized composite data or entities. Because composite data can consist of sub-structures, data are not stored in a list but in a tree that contains structural information about the object. In ideal case, when there is one entry point object and all other objects are its sub-structures, entry point object is also the root of the whole tree. This, however, is not common because most usually, thanks to inheritance, some sub-structures are abstract classes or interfaces. In such case it is not possible to load characteristics of all objects, because some of them might be defined in other libraries. Libraries, that are directly referenced, can be processed and loaded during startup, however dynamically loaded libraries are unknown during startup and thus it can

happen that an instance of a new inherited type is created and additional code characteristics must be loaded and interface creation process have to be performed incrementally to support new functionality and properties of the new object.

7.2.2 *Code*

The characterized code tree is built from public methods of the object. Description of every method consists of its attributes that describe command name and its behavior. Because some methods can have parameters, these are stored in a parameter tree. Every parameter is a node with attached attributes, describing parameter. Parameter can reference node from data tree, because it can be data characterized object for which specific data characterization have been specified.

7.2.3 *Algorithm*

Algorithm presented in Figure 7.2 demonstrates steps needed to load the characteristics. A high level description is used to present the algorithm.

```
foreach(DataType in CharacteristicsFile)
{
      if (DataType has no characteristics) Continue;
      Create instance of data type characteristics.
      if (DataType is inherited) Get Characteristics of base data type
      Parse Characteristics of DataType
      foreach(Variable in DataType) {
            if (Variable has no characteristics) Continue;
            if (Variable is from CharacteristicsFile) {
                  Get Data Type Characteristics of Variable From the Tree
            }
            Parse Specific Characteristics of Variable
            Insert characteristics to instance.Data
      }
      foreach(Method in DataType) {
            if (Method has no Name attribute) Continue;
            Create instance of Method characteristics.
            Parse Specific Characteristics of Method
            foreach(Parameter of Method) {
                  Create instance of parameter characteristics
                  if (Parameter is from CharacteristicsFile) {
                        Get Characteristics of Parameter From Tree
                  }
                  Parse Specific Characteristics of Parameter
                  Insert characteristics to Method.Params;
            }
            Insert characteristics to instance.Code
      }
      Insert data type characteristics to Tree
}
```

Figure 7.2 Algorithm for parsing characteristics

Data and code that has been characterized must be loaded to provide information for the next steps in the automated user interface generation process. Each data type from an assembly or other source characteristics file is processed. If the data type has no characteristics, it is not taking part in the user interface and will be skipped. A structure holding loaded characteristics of the data type is created and all characteristics – data and code – are stored within. All the characteristics nodes of the data type are parsed and their values are stored in the structure. The process of loading characteristics is repeated recursively for any internal variable and its data type. These loaded characteristics are stored into the structure as sub-nodes, creating tree with the original data as the root and characterized variables as sub-nodes.

After the internal variables of the data type are parsed, characterized methods are processed. Because characterized method will be visible in the user interface, it has to define characterization tag *Name*, which will be used to represent the command in the user interface. Methods not having this tag defined are skipped and will not be used in the process of automated user interface generation. For any characterized method, a structure containing its characteristics is created and filled with the parsed characterization values. Depending on the type of method – with or without parameters, existing parameters are parsed for the characterization tags and stored in the method's structure. Complete structure with method characteristics and its parameters is stored in the parent data type structure. This ensures all the characterized data types and its methods are stored in a tree structure, defining basic structure of the user interface in the means of grouping objects in the user interface.
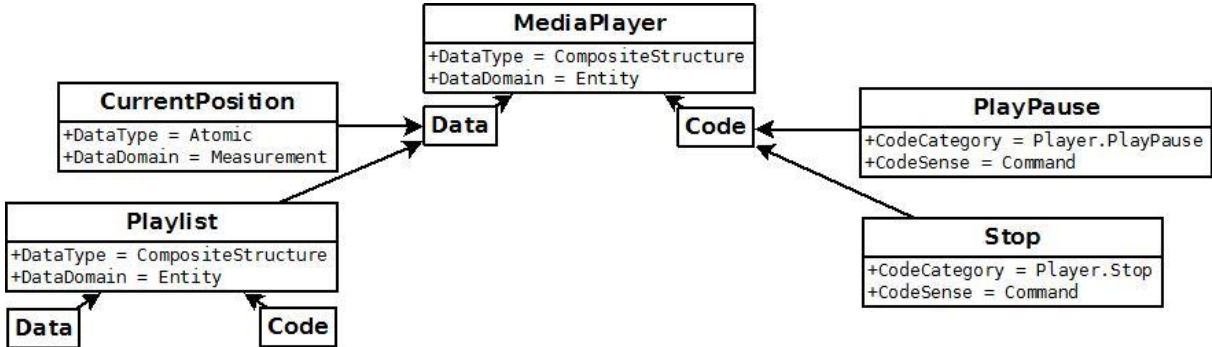

Figure 7.3 Simplified characteristics tree of a media player

Figure 7.3 depicts simplified tree of a media player application, where only few attributes and only small number of data and code is displayed. Actually, all objects contain multiple

characterization attributes and many of them have complex data and code sub-trees. Although this is not complete example, it is sufficient for demonstration of loaded characterization in aforementioned paragraphs. MediaPlayer object is characterized as entry point, thus it is defined as a root of the tree, and contains all own data and code characterization attributes. In the figure, only two attributes are displayed to make the example well readable. The object of media player contains properties and methods that are characterized and should be used in the final user interface. In the data tree of the MediaPlayer, there are multiple data types, but for the demonstration purposes only two are displayed. The "CurrentPosition" data type is a concept representing position in the played media file. It does not have additional data or code sub-trees because it is native atomic value type. The "Playlist" is a list with entries, representing items that can be played by the media player. Playlist have data and code sub-trees containing definition internal variables and methods performing various actions. In the code tree of the MediaPlayer, there are multiple methods, but for the demonstration purposes only two are displayed. The "PlayPause" method is from the player category and indicates that it is used to play, or pause, playback of selected media file in the playlist. It has no parameters so no data sub-tree is present. The "Stop" method is from the player category too and indicates that it is used to stop the playback. It has no parameters so no data sub-tree is present.

7.3 User interface creation process

The characterization tree with loaded information about the characterized code and data contains structured information about application, its data and functions. This structured information can be used to automatically generate user interface. The process of automatic generation of the user interface has several steps. First, object of the tree must be abstracted so that they can describe possible user interface elements. Abstraction must be general enough to be platform independent but allow application of constraints, which will be used later to choose proper concrete user interface element. Abstracted object then must be grouped together in to logical units, which will be presented to user together in a single block (e.g. in a window or dialog in case of desktop user interface). Grouped abstract objects are concretized using constraints to concrete user interface elements. User interface elements are then placed to a proper part of user interface based on used modality. In case it is not possible to place the objects in a single block, another iteration of concretizing has to be performed to find optimal solution. After completing the layout, evaluation should follow to get metrics about quality of generated user interface. Whole process will be described in a more detail in the next sub-

sections.

7.3.1 *Abstraction of objects*

As soon as characterization tree is loaded, it is necessary to abstract characterized data types and code to objects that can be manipulated and presented in the user interface. To represent these objects, a concept of concrete interaction object (CIO) [135] is defined as any part of user interface that users can perceive (e.g. text, image, animation, sound) and manipulate (e.g. button, text box, check box). It could be an object from a library or provided by some toolkit. Available user controls from across platforms, various libraries, and toolkits, seem to be different, but when looking closer, they are similar. Figure 7.4 depicts different forms of CIOs: top line contains checkboxes with a different look but the same feel (Java Swing, Windows Forms, and Motif respectively), second line contains checkboxes with the same look and different feel (Microsoft Windows, Motif, and Garnet respectively).
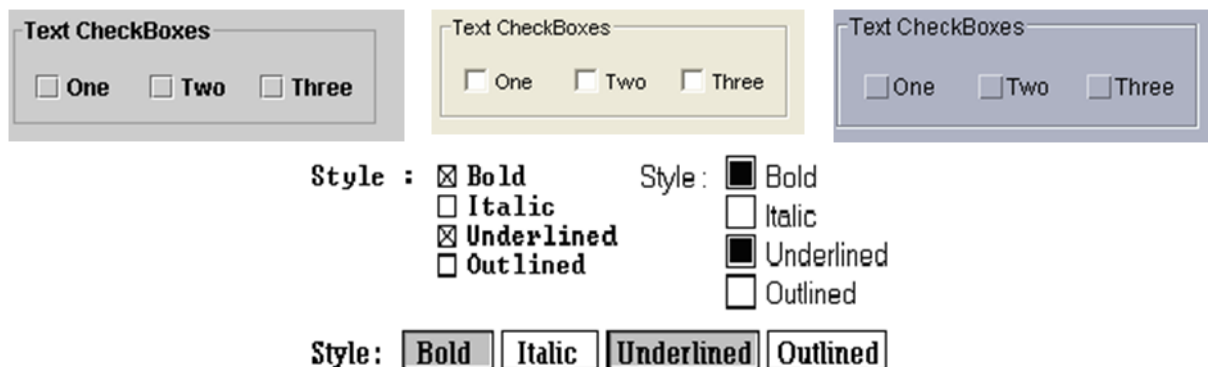

Figure 7.4 Variations of concrete interaction objects as shown in [135]

Abstract Interface Objects

While different CIOs may exhibit different look and feel, they have same behavior. And that is the reason why a concept of abstract interaction object (AIO) was introduced [135] and is defined as an abstraction of a set of CIOs with respect to a set of common properties [136]. AIOs contain general attributes (such as height, width etc.), events (such as checked etc.), and primitive functions (such as Enable, Check etc.). By definition, AIOs do not have any particular representations.

Linking AIOs with CIOs

Before data and code can be abstracted to AIOs, a library of AIOs has to be defined. Definition of individual AIOs cannot be done automatically. Every AIO should be given

understandable name and depending on its purpose a set of suitable user controls (CIOs) that are capable of providing required functionality must be linked to the AIO. Thus AIO should contain subset of attributes from well known predefined set (such as height, width etc.) that should be mapped to each CIO and a subset of well known predefined methods that can provide required functionality. Again, each method should be linked to a method of particular CIO. Linking AIO with CIOs is important because user interface generation process must know how to use attributes and methods of AIO to be able to control final CIO. Thus assignment to property or executing method of AIO is mapped to assignment to concrete property or execution of method of instantiated CIO.

Mapping data and code to AIOs

Each characterized data and code in the characterization tree must be mapped to AIO which represent information about future user interface control that will be used for interaction with user. Choosing appropriate AIO for each method or data type is based on a set of rules. If there is no rule that could be applied to data type or method, then it is not possible to find AIO and create resulting user interface. That is why a set of rules must be complex and stable. By stability is meant that when applying rules to find ideal AIO, there must be just one AIO that fits the rules.

For example, a string data type can have data characterization as atomic data type, entity domain and static transience. Set of rules that leads to AIO named "text label" could be:

1. data type is string
2. data characterization type is atomic
3. data characterization attribute transience is static

<div align="center">**Figure 7.5 Set of rules to select AIO named "text label"**</div>

This means that AIO "text label" will be used in case of all properties which are strings and are characterized as atomic (do not have internal structure) and are static (content is not changing) . In the case when transience could be dynamic, it would be possible to choose AIO with functionality that is similar to text box, where value can be changed. Thanks to rule sets it is possible to map various AIOs. For example characterized atomic data which are concepts named "timeline" can be mapped to AIO called timeline, which can be represented by track bar, or progress bar, or other special user control allowing changing of value in horizontal direction. Similarly, characterized methods are mapped to AIOs which can be represented in

resulting user interface by buttons, menu items or other CIOs.

Algorithm

The algorithm in the high level description in Figure 7.5 presents a way of converting characterization tree to AIOs.

```
Load AIO Database from repository
CreateAIOs(dataChar, userPrefs, context)
{
    foreach(aio in AIO database) {
        Score = EvaluateRules(aio, dataChar, userPrefs, context);
        StoreScore(aio, score);
    }
    PickAIOwithHighestScore();
    PerformAIOSpecificSetup(dataChar, userPrefs, context);
    foreach(dch in dataChar.Data) {
        CreateAIOs(dch, userPrefs, context);
    }

    foreach(cch in dataChar.Code) {
        if(cch has defined category AND
           smart template for the category exists) {
            Get smart template AIO for the category
            Link cch with smart template AIO
        } else {
            foreach(aio in AIO database) {
                Score = EvaluateRules(aio, dataChar, userPrefs,
                                      context);
                StoreScore(aio, score);
            }
            Pick AIO from AIO database with highest Score;
        }

        if (cch.Params > 0) {
            Create Dialog AIO
            foreach(parameter in the cch.Params) {
                CreateAIOs(parameter, userPrefs, context);
                Add created AIOs into Dialog
            }
            Link Dialog with cch
        }
    }
}
```

Figure 7.6 Algorithm for conversion of characterization tree to AIOs

First, definitions of available AIOs are loaded from the repository. Method *CreateAIOs* is called for the root data characteristics in the characterization tree. The parameters of the method are, besides the data characteristics, user preferences and the context, used in the process of conversion to AIOs. For each AIO, defined in the AIO database, is computed a score representing suitability of the AIO for the given characteristics. After all AIOs were evaluated, AIO with highest score is selected and initialized based on the data or code

characteristics, user preferences and context.

Each characterization tree node consists of the data and code sub-trees representing internal data and commands and must be converted to AIOs too. If the internal data and commands are not converted, the user interface would not be complete. First, all nodes of the data sub-tree are converted, and then all nodes of the code sub-tree. In case of the data sub-tree, the method *CreateAIOs* is called recursively to assure all data characteristics are converted to AIOs in a straight way. For the code sub-tree, conversion is a bit more complex. Because some methods can be part of some category (for example methods for collection, player etc.), code characteristics should be tested if it belongs to some category. If it does and a smart template AIO for the category exists, smart template AIO is updated with the new code characteristics to reflect new method implementing individual command of the smart template AIO. In other cases (code characteristics does, or does not, define a category, or smart template AIO does not exist), scores for all AIOs in the AIO database are evaluated and the one with highest score is selected. This mechanism of computing score takes into consideration rules applying to code categories and allows picking of smart template AIO which is the best for the categorized code.
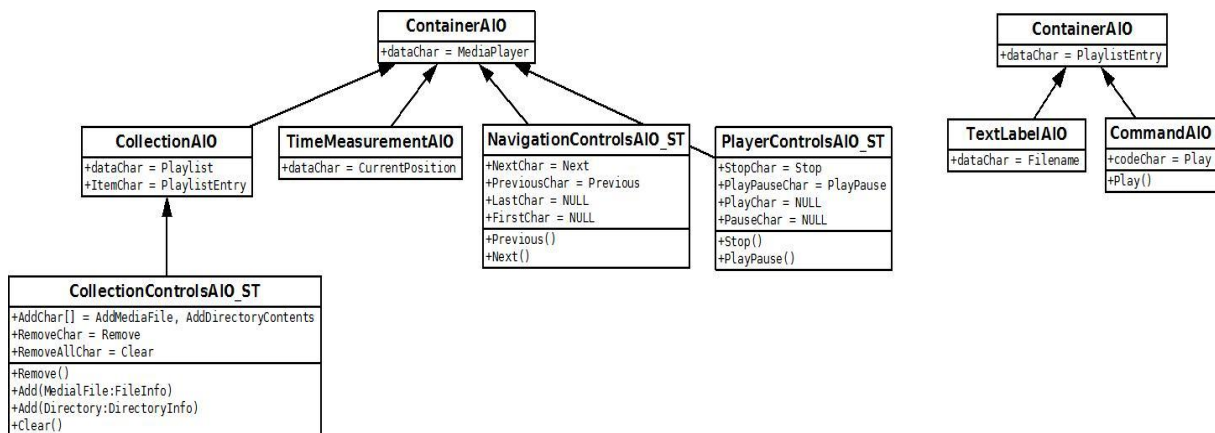

Figure 7.7 Example of conversion of the characterization tree of the simple media player to the AIOs.

Now, all the data and its methods are converted to AIOs except parameters of the methods. Parameters of the methods can refer to certain data or request user input and must be converted to AIOs too. That is why a container AIO (a container for user interface elements required for the parameter input) is created and filled with AIOs of characterized parameters. Container, containing the method parameters, will be extended by AIO allowing committing or cancelling set values to allow command execution when all parameters are set properly.

7.3.2 *Object grouping*

Because many modern programming languages are using object oriented programming, their paradigms can be used as a benefit for grouping objects that should be together because they form particular control and interaction logic. That is because one of the most important paradigms of object oriented programming is solving tasks with respect to reality [137]. With these paradigms in mind, programmers should design classes and objects containing information that should be reasonable for the classes and objects. Thus properties of each object should be grouped together to form a dialog with all necessary information about the object.

However, presentation abilities of various output devices are always limited in some way. For example, devices with display providing graphical output can show only limited amount of information at a time. Limit is affected by display size, its resolution, depth of color, font size etc. That is why objects should be grouped together into containers that can be abstracted for the user so that it is possible to control resulting user interface without problems. To reduce amount of information presented at the same time, internal structure of object can be used as a benefit, because programmer already abstracted information that should remain together by creation of substructures or internal properties of object. It can be said that almost every substructure can be abstracted as container containing properties of the substructure. In cases when objects have many properties that cannot be effectively presented thanks to limits of presentation device, these should be regrouped into smaller containers which can be presented independently. For example, in case of small display with small resolution, it is possible to group AIOs into several containers inside the tab control.

Grouping of objects must consider, besides the capabilities of the device, the importance attributes. Objects and commands with high importance should be always easily accessible and that is why it is important to place such objects into containers that will be easily accessible too. In case of commands with high importance, it is possible to place the commands in the main menu to enable easy accessibility. In the means of characterization tree, command is duplicated and moved to another container, for example abstracted as a "tool bar" AIO. This duplication means that command over the object can be performed from various user interface places. Because command duplicates all the information, including dependency attributes, availability of the command is same in all containing parts of the user

interface.

7.3.3 *Abstract objects concretization*

Because AIOs do not represent concrete user interface element, conversion to CIOs must be performed.

Selecting modality

Modality defines a way of communication between the user and the device. Before any AIO can be concretized to CIO, a modality must be chosen so that appropriate CIO can be selected that suits the chosen modality. Information about capabilities is analyzed to induce which modalities are available. For example, for a device that has touch display, speaker and microphone, but no keyboard, several modalities are available:

- visual output – display can present information to user
- sound output – speaker can play sound so information can be converted to speech
- touch input – display is touch sensitive, user inputs by hand
- voice input – microphone can record voice that can be recognized as commands

Next, information about user preference is used. In case user did not specify his preference, selection of modality will be further influenced by surrounding environment. In the other case, modality will be chosen depending on selected preferences. A problem might occur when user selects modality that device is not able to perform. For example, user requires sound output and voice input but device does not have built-in microphone. Although device can present information by sound, it is not able to process audio commands from the user. Creation of user interface that would satisfy preferences is not possible and user interface cannot be created. It is, however, possible to create user interface that would accept touch input – this could mean that user interface would not be suitable for blind or visually impaired user who could set the preferences for this reason.

Context of environment is taken into account when user has not specified any preferences. Based on parameters of environment, appropriate modality is chosen from available ones. For example, user is in environment with high levels of noise. In this case, sound output and voice input will be ignored because in environment with high amount of noise it is hard to hear sound output properly and also voice recognition is difficult. Visual output and input by touch or keyboard is preferred because environment does not affect these modalities. Another example of environment context is user performing some activity that occupies user's sight –

e.g. driving a car. Because user that has occupied sight cannot control a device with visual output, other modality must be selected to allow comfortable communication with the device, such as sound output. For input modality, controlling keyboard, mouse or touch area of the device is not possible because user has occupied also his hands during driving – voice commands are the best solution. From the examples above it is obvious that selecting appropriate modality requires a lot of information to be taken into consideration. However, basic principle remains the same – best modality is the one that will cause user less restrictions and compromises in his actions when using the device in current context.

A problem might occur when there are no CIOs defined that would be suitable for selected modality. That is why a selection of modality must consider availability of suitable user interface controls when deciding which modality is best in current context. For example, a device allows multiple modalities, including audio output and input, touch control, visual output, and user has preference for audio output and voice input – but because there is available just one library with desktop user controls, then only visual output is available.

Filtering CIOs

Based on the selection of the proper modality, it is possible to filter out all unsuitable CIOs. This is possible because each AIO is linked to multiple CIOs from different platforms and libraries. After filtration is finished, each AIO should have at least one corresponding CIO because if there is no CIO, it would not be possible to choose proper user control to present in user interface and thus it would not be possible to create resulting user interface.

Filtering of CIOs is driven by set of rules that describe what kind of user interface control is suitable for selected modality and what is not. For example, for visual output, CIOs with graphical representation should be preferred over those CIOs with textual representation. In case of 3D visual output, all user interface controls that are not represented by 3D models (but 2D graphics) should be omitted. After filtration, only CIOs that satisfy all rules defining suitability of CIO for selected modality are present.

Selecting CIOs

In case there is just one CIO left, selection of CIO is straightforward and new instance of CIO is created. In case there are multiple CIOs suitable for representation of the object, a set of rules should be used to choose the best one. A set of rules is dependent on selected modality

and usage of the AIO. For example, a property of enumerated data type is atomic concept, enumerated type has three values. Suitable CIOs are represented by a combo box or a set of radio buttons. Both CIOs are suitable for selection of value of the property but each of the CIOs have different look and feel, combo box requires less space, set of radio buttons allow more comfortable selection. A score, rating suitability of each of both CIOs, is required to decide which of the CIOs is more suitable. A score is computed depending on rules associated to each of CIOs. Rules can consider requirements on display size and resolution, whether it is related to other data, whether it can express relation to other data (selection of other value leads into change of data in the user interface) etc.

From the Figure 7.8, a set of radio buttons will be selected in case that resolution and display size are big enough – because it is well-arranged, does not express relation (enumeration does not express relation), expresses enumeration, and is suitable for small number of items. Combo box would be better choice in case of small display and resolution because its size is constant for any number of items.

	CIO "combo box"	CIO "set of radio buttons"
tabular	*false*	*true*
constant size	*true*	*false*
expresses relation	*true*	*false*
expresses enumeration	*true*	*true*
large number of items	*true*	*false*
requires precision	*false*	*false*

Figure 7.8 Example of rules for score computation

Designing layout

After each AIO was concretized to CIOs, it is possible to design layout of the resulting user interface. Although it might seem that designing layout is placing of user controls in a form or other desktop container, it has more general sense. By designing of layout of user interface is meant positioning of user interface elements in the previously specified containers. This is not generally placing user interface elements in a window. Because user interface can be created for various modalities, it can be positioning in 2D or 3D space, or in time. Thus designing layout must consider different rules when placing user interface elements in various spaces. Because the goal of this work was focused on data and code characterization, following

subsection does not describe automated layout techniques, it rather refers to sources describing these techniques.

In case of 2D visual user interface, user interface elements must be placed optimally and follow some general rules, for example place user interface objects in a way that user does not need to move the mouse much in order to get some work done. Many automated layout methods have been presented in [138][139]. In case of 3D visual user interface, placing of user interface elements is more complicated than in 2D. Various occlusions might occur and control of user interface in 3D is more complicated because objects can move in various directions and can have complex shapes. Some automated layout methods have been presented in [140][141]. In case of audible modality using sound output and voice input, it is important to remember that user has to keep in mind what is he doing and must remember lots of information about the action he is performing. This means that user interface should present requests sequentially, always report what user is doing etc. For example, in case of blind user willing to send new email – by default, in visual user interface, user clicks "write new email" button and fills in address, subject, and text. In case of blind user, user interface has to tell that he is using email client. User replies for new email command and user interface sequentially asks for address, subject, and text and waits for user to confirm sending. Methods for modeling of user interface based on audible modality are presented in [142].

Algorithm

Following algorithm in the high level description presented in Figure 7.9 expresses possible way of AIOs conversion to CIOs. The presented algorithm is generally enumeration of all possible ways of a choosing and inserting of user interface elements. This approach hides well distinguishable steps of object grouping and layout designing. However, the goals of this work do not cover precise algorithms for the user interface generation process but rather provide proof of the concept for which is this approach sufficient.

The goal of the algorithm is to choose solution with the smallest effort needed for the interaction. Method *ConvertToCIOs* takes as a parameters characterization tree node, respective AIO to be converted to CIO, context of the user and the device capabilities. Information about the user context and device capabilities is used for selection of the right CIOs and also to compute the cost function which returns cost of the interaction. The cost function generally computes value (based on the selected modality, devices capabilities, and

so far converted AIOs) that reflects effort required for the interaction of user with the user interface. For example, in case of visual user interface, total cost can reflect total distance of cursor travelled among various user interface elements and number of clicks that needed to be performed in order to control the user interface elements. After evaluation of the current cost, value is compared with the best cost so far and continues if current cost is better than the best cost. For other cases there is no reason to continue as final user interface would have higher cost. If all AIOs have been converted to CIOs and the current cost is better than the best cost so far, solution is stored and algorithm continues with other possible enumerations to test remaining possibilities.

```
ConvertToCIOs(ChTree, AIO, Context, Device)
{
      if (CurrentCost(ChTree, Context, Device) >= BestCost) return;
      if (AllCIOsApplied()) {
            BestCost = Cost;
            BestCIOs = ChTree.CIOs;
            return;
      }
      CIOs = GetCIOs(AIO, Context, Device);
      foreach(CIO in CIOs) {
            if(ApplyCIO(CIO, AIO, Device)) {
                  subAIOs = GetSubAIOs(AIO);
                  SortByImportance(subAIOs);
                  foreach(subAIO in subAIOs) {
                        ConvertToCIOs(ChTree, subAIO, Context, Device);
                  }
            }
            UndoLastCIO();
      }
}

AIOsToCIOs()
{
      foreach(SubTree in CharacterizationTree) {
            while(true) {
                  ConvertToCIOs(SubTree, SubTree.AIO, Context, Device)
                  if (ConversionComplete(SubTree)) break;
                  RegroupLowestImportanceContainer(SubTree);
            }
      }
}
```

Figure 7.9 Algorithm for AIO to CIO transformation

GetCIOs returns all possible CIOs that are defined for the transformed AIO. For each of these CIOs, sub-AIOs of the currently transformed AIO are converted to CIOs. Thus, all variants of the user interface are evaluated and the best cost is found. The algorithm selects high importance AIOs first and places them in the top bottom order. Although this solution is not

ideal, the goal of this work is to proof the concept of the data and code characterization and for this purpose sorting of important AIOs is sufficient.

AIOsToCIOs is starting method performing conversion of the whole characterization tree. It is converting every sub-tree of the characterization tree. At first, the sub-tree is converted to CIOs. This, however, might fail because of constraints of the device – for example screen dimensions do not allow placement of other user interface elements. For this reason, a regrouping of the AIOs is implemented and lowest importance AIOs are grouped together into container (for example, parameters with low importance will be grouped in a new dialog so that user can show them after pressing a button). The regrouping continues until conversion to CIOs is successful. Figure 7.10 demonstrates tree of CIOs converted from the AIOs of the simple media player.
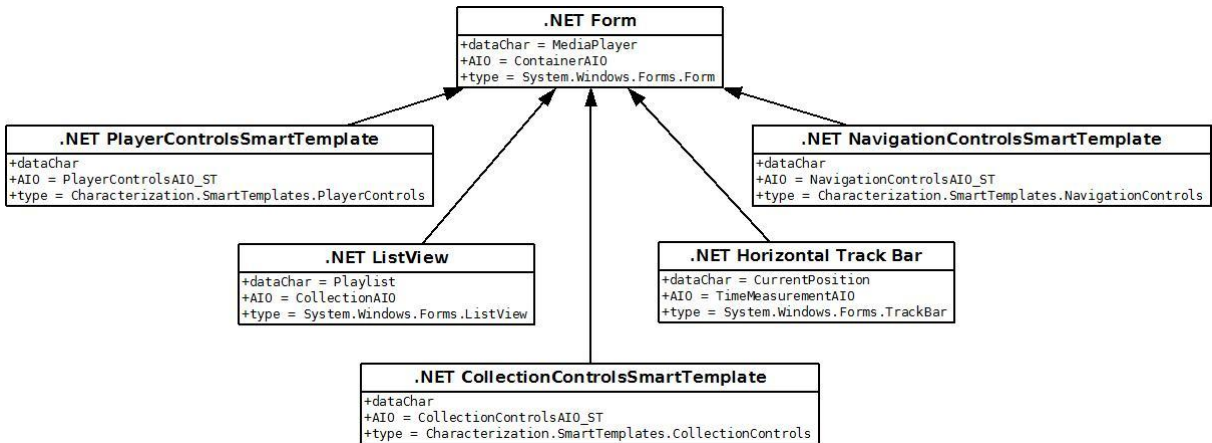
Figure 7.10 An example of CIOs converted from AIOs for the .NET Framework platform.

7.3.4 *Instantiation of the user interface*

Process of instantiation of the user interface creates fully functional user interface from the characterization tree and CIOs. All the CIOs are from the same framework or library that will be used for the user interface. In this moment is also known modality for the interaction because selected CIOs support interaction using the modality. Instances of the user interface elements (based on CIOs) should have the same hierarchical structure as AIOs. The structure is result of object oriented programming and regrouping of the objects during the process of the user interface generation.

Instantiation of the user interface performs two distinctive tasks. The goal of the first task is to create instances of the user interface elements, setup their properties and manage their

hierarchy. Result of the first task is user interface that has the right structure but no functionality. For example, visual user interface would have its correct shape and layout but pressing buttons would perform no actions. The goal of the second task is to ensure all user interface elements will have the right behavior performed by proper method calls, value updates etc. After both tasks are complete, the user interface can be used for the purpose the application was created for.

Ensuring the right behavior of the user interface elements is not simple. Lots of additional code must be dynamically generated that will handle changes of the data or user interface elements states. A code performing communication with user for commands with multiple parameters must be generated, including various test of new value validity. A code performing tests over command availability must be generated to allow execute only those commands which have all conditions met. Modern programming frameworks such as .NET Framework or Java allow dynamic code creation and compilation during runtime. Older programming languages or frameworks that do not support dynamic code creation and compilation have to define many templates that can be used in various scenarios.

Algorithm

Following algorithm in the high level description presented in Figure 7.11 performs instantiation of the CIOs obtained from the output of the previous algorithm performing conversion from AIOs to CIOs in Figure 7.9.

First, instance of the user interface element from the CIO is created as a sub-object of the parent object of the user interface. Then, parameters of the CIO are set to the instance of the newly created user interface element. For the CIOs that represent data, events and handlers are registered for dependencies, object, and instance changes. For transient CIOs (change in time frequently), regular update of the user interface element is scheduled. For the CIOs that represent code, events and handlers are registered for dependencies. Because code can have characterized dialog attribute, handler is generated that can show question or asterisk to the user before the code will be executed. In case of the code CIO that is linked to container of other CIOs, handler showing dialog is generated based on the type of the container and type of the characterized code it represents. For a common code, a handler is generated that executes the code.

106

```
Instantiate(CIO, parent)
{
      instance = parent->CreateInstance(CIO);
      instance->SetParameters(CIO);
      if (Characteristics(CIO) is CODE) {
            RegisterDependences(instance);
            if (Characteristics(CIO) has Dialog attrib) {
                  GenerateQuestionCode(CIO);
            }
            if (CIO is linked to Dialog CIO) {
                  dlg = Instantiate(Dialog CIO, null);
                  RegisterCallingRoutine(instance, dlg);
            } else {
                  RegisterCallingRoutine(instance);
            }
      } else {
            if (Characteristics(CIO) has relation) {
                  obj = GetRelatedObject(CIO);
                  GenerateUpdateRelatedObject(instance, obj);
            }
            RegisterDependences(instance);
            RegisterInstanceChanged(instance);
            RegisterObjectChanged(instance);
            if (CIO is transient) RegisterRegularUpdate(instance);
      }
      foreach(subCIO in CIO children) {
            Instantiate(subCIO, CIO);
      }
}
```

Figure 7.11 Algorithm for instantiation

Figure 7.12 demonstrates final user interface of the simple media player, obtained by using instantiation algorithm described above.
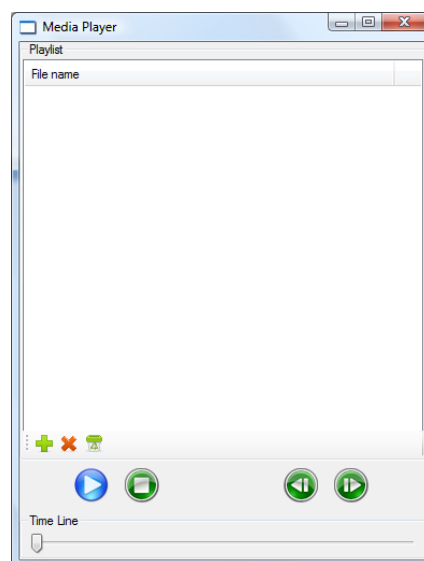
Figure 7.12 Example of final user interface obtained from instantiation of the CIOs from the simple media player example.

107

7.3.5 *Summary*

To automatically create user interface from characterized data and code, it is important to consider many aspects. A proper data and code characterization, building characterized tree, selection of abstract interface objects, instantiating concrete interface objects, and setting their layout is required. Process presented here is simple but enables automated creation of simple user interfaces. Every single step in creation of automated user interface would require more investigation as the task is very complex.

7.4 Constraints

The technique presented here for automated user interface generation from characterized code has several drawbacks. One of the most important drawbacks is unavailability of redesigning of user interface based on its evaluation. During common process of creation of user interface, a user interface designer or expert creates user interface and let the interface evaluate. Evaluation can be performed by users or design experts and must be well prepared. Evaluation techniques and process is well described in [6]. After the evaluation process, user interface designer considers effectivity, usability and other aspects of created user interface and modifies user interface so that it reflects acquired results. Then another round of evaluations follows. This is not possible with proposed user interface generation process, because proposed process of generation of user interface does not take into account usage of the generated user interface. Usage of the user interface could provide metrics that could be possible to analyze and with a help of the analysis, user interface generator could modify layout in a way which would be more effective and usable than previous one. More research in this functionality and its implementation is required to increase quality of resulting user interface.

A limitation of the presented process in user interface creation is a problem with creation of the general user interface in 3D space such as interface in a virtual or augmented reality. It is not possible to create general description that would be able to render any object in 3D space because representation of the data may be different among various 3D engines or libraries. For example, information about the vertices and their properties are stored differently in Direct3D and OpenGL. Direct3D is using structure with vectors describing location, color, normal, and texture coordinates, while OpenGL is using array lists where each of array lists contain different data – location, or color, or normal etc. This means it is not possible to create

platform and library independent solution for rendering 3D user interfaces. It is, of course, possible to use one of the libraries and use it for rendering of user interface. This, however, means that all the data will be always rendered in the same way because developer would not have a chance to modify rendering of the objects to produce more visually attractive rendition of the data. As long as developer cannot supply own rendering engine, it is not possible to create rich 3D applications, which are built on the own effects and details supported by the engine. That is why creation of user interfaces in 3D space is limited. Although it is possible do create user interface in 3D space, visual appearance will not be appealing.

7.5 Target group of applications

The proposed here code characterization and user interface generation process is not targeted to general application context. Code and data characterization proposed here is not capable of expressing enough information about data and code characteristics about all applications to an artificial intelligence user interface generator. Even with very high quality user interface generation process, it will be very hard to create application of such extents as, for example, word processor. The reason is that text processor requires many custom user controls and behaviors that cannot be generally expressed by code or data characteristics.

The aim of the code and data characterization is to support applications that can be run on various devices with various platforms. Characterization taxonomy allows fast application development and even testing. Developers can characterize data and code during the development and run the application. Because it already contains characterized code and data, user interface can be created to reflect data and operations with the data that has been characterized and application can be tested even in early development stage. On the contrary, using task models for automated user interface generation of user interface of developed application is not usually possible and also analysis of the tasks takes more time than simple data characterization.

Thanks to code characterization, it is easy to extend functionality of the application. Adding or removing methods of objects leads during runtime to automated generation of new user interface that contains all available features that has been characterized. Other approaches such as task modeling do not enable such behavior because adding new functionality breaks task model tree which must be redesigned to reflect new functionality. The same problem arises with adding new parameters to functions or methods of characterized objects. While

thanks to code characterization there is no additional cost than characterizing new parameter and launching application, task models have to redesign task model tree to be able to generate user interface that reflects modified method parameters.

8 Conclusions

The goal of this work was to define approach for automated user interface generation that would be simple to use and allow automated creation of user interface for various platforms. This goal was fulfilled.

A taxonomy and processes proposed and described in this work allow new way of application development. While developers were forced to redesign user interface every time a major change in the application has been done, using code characterization together with automated user interface generation process can shorten development time and allow developers to test new methods and functions as the application is being developed even in early stage of development. The proposed taxonomy and its usage lead to benefits in the user interface design. The key benefits of the proposed characterization taxonomy are:

- no expert skills required,
- application code independence on user interface,
- simple application extensibility,
- good maintainability and readability of the application code,
- more effective and faster algorithm development,
- independence on user interface generation process,
- platform independency.

The taxonomy is ideal for creation of user interfaces on multiple platforms. Automated user interface generation process can generate optimal user interface for any device. To proof the concept of the characterization taxonomy, a process of automated user interface generation was presented and demonstrated on examples.

At the moment, the weakest point of the automated user interface generation is generation of layout of user interface elements. An experienced graphics designer is able to produce layouts that are aesthetically superior and more effective than current automatically generated user interfaces. Future work in the problematic of characterization taxonomy should be focused on automated layout techniques to enable higher quality user interfaces, and creation of complex applications to tweak the data and code characterization taxonomy to cover any information required by the automated user interface generation process. Future work should also consider implementation of the user interface generation process for other modalities to take the advantage of the presented approach.

9 Bibliography

[1] Apple iPod Shuffle, Apple Inc., 2007, web pages, Available from http://www.apple.com/ipodshuffle/

[2] Brooks, A, "It's Only Rock and Roll", Apple News - World of Apple Events, 2009 http://news.worldofapple.com/category/world-of-apple-events/

[3] Delahunty, J, "Apple profit makes huge rise due to iPod success", afterdawn.com, 2005, http://www.afterdawn.com/news/archive/6296.cfm

[4] Bainbridge, W. S., "Berkshire Encyclopedia of Human-Computer Interaction", Berkshire Publishing Group, 2004, ISBN: 0974309125, 1000 pages

[5] Thyssen, M, "Hammer", Wikimedia Commons, 2009, http://commons.wikimedia.org/wiki/User:Malene

[6] Stone, D., Jarrett, C., Woodroffe, M., Minocha, S., "User Interface Design and Evaluation", Morgan Kaufmann, 2005, ISBN: 0120884364, 704 pages

[7] Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S., Carey, T., "Human-Computer Interaction: Concepts And Design", Addison Wesley, 1994, ISBN: 978-0201627695, pages: 816

[8] Wickens, Ch.D., Lee, J.D., Liu, Y, Gordon-Becker, S.E., "Introduction to Human Factors Engineering (2nd Edition)", Prentice Hall, 2003, ISBN: 978-0131837362, pages: 608

[9] Chen, Z., Marx, D., "Experiences with Eclipse IDE in programming courses", In Journal of Computing Sciences in Colleges, Volume 21, ACM Portal, 2005, ISSN:1937-4771, pages: 104 – 112

[10] Gajos, K., Weld, D. S., "SUPPLE: Automatically Generating User Interfaces", in Proceedings of IUI'04, Funchal, Portugal, 2004

[11] Nichols, J., Myers, B.A., Litwack, K., "Improving Automatic User Interface Generation with Smart Templates", In Intelligent User Interfaces, Funchal, Portugal, 2004, pages: 286-288

[12] Roberts, D., "Signals and Perception: The Fundamentals of Human Senses", Palgrave Macmillan, 2002, ISBN: 0333993640, 320 pages

[13] Human sensory reception, Encyclopædia Britannica Online, 2009 http://www.britannica.com/EBchecked/topic/534831/human-sensory-reception

[14] MindDrive, The Other 90% Technologies Inc., 2007, web pages, Available from http://www.other90.com/

[15] Holmes, J. N., Holmes, W. J., "Speech Synthesis and Recognition", Taylor & Francis,

2002, ISBN: 0748408576, 320 pages

[16] Jacko, J.A., Sears, A., "The human-computer interaction handbook: fundamentals, evolving technologies and emerging applications", CRC, 2002, ISBN: 978-0805844689, pages: 1312

[17] Sight, New World Encyclopedia, 2009, web pages http://www.newworldencyclopedia.org/entry/Sight

[18] Schematic diagram of the human eye, Wikimedia Commons, 2009 http://commons.wikimedia.org/wiki/File:Schematic_diagram_of_the_human_eye_en.svg

[19] Koch, K., McLean, J., Segev, R., Freed, M.A., Berry, M.J., Balasubramanian, V., Sterling, P., "How much the eye tells the brain", Curr Biol, 2006, pages: 1428-1434

[20] The Hermann Grid, BrainConnection, 2007, web pages, Available from http://www.brainconnection.com/teasers/?main=illusion/hermann

[21] Human ear, Encyclopædia Britannica Online, 2009, web pages http://www.britannica.com/EBchecked/topic/175622/ear

[22] Human ear, Wikimedia Commons, 2009, web pages http://commons.wikimedia.org/wiki/File:HumanEar.jpg

[23] Brewster, S. A., "Using non-speech sounds to provide navigation cues", ACM Transactions on Computer-Human Interaction, vol. 5, 1998, ISSN: 1073-0516, pages: 224-259

[24] Robles-De-La-Torre, G., "The Importance of the Sense of Touch in Virtual and Real Environments", IEEE MultiMedia, Volume 3, IEEE Computer Society Press, 2006, ISSN: 1070-986X, pages: 24-30

[25] Tactile Feedback, VWN Virtual Dictionary, 2009, web pages http://www.virtualworldlets.net/Resources/Dictionary.php?Term=Tactile%20Feedback

[26] Force Feedback, VWN Virtual Dictionary, 2009, web pages http://www.virtualworldlets.net/Resources/Dictionary.php?Term=Force%20Feedback

[27] Rumble Feedback, VWN Virtual Dictionary, 2009, web pages http://www.virtualworldlets.net/Resources/Dictionary.php?Term=Rumble%20Feedback

[28] Wasinger, R., "Multimodal Interaction with Mobile Devices: Fusing a Broad Spectrum of Modality Combinations (Dissertations in Artificial Intelligence: Infix)", IOS Press, 2007, ISBN: 978-1586037123, pages: 246

[29] Smellovision, 2009, web pages http://www.nowinsmellovision.com/

[30] CFBF 4D motion cinema, CFBF Amusement Equipment Co., Ltd. 2005, web pages

http://www.cfbf.cn/main/4D_Motion_Cinema.htm

[31] Memory, Wikipedia, The Free Encyclopedia, 2009, web pages http://en.wikipedia.org/wiki/Memory

[32] Cowan, N., "The magical number 4 in short-term memory: A reconsideration of mental storage capacity", Behavioral and Brain Sciences, Vol. 24, Cambridge University Press, 2000, pages: 87-114

[33] Moeslund, T. B., Granum, E., "A Survey of Computer Vision-Based Human Motion Capture", Computer Vision and Image Understanding, Volume 81, Issue 3, Elsevier Science Inc., 2001, ISSN: 1077-3142, pages: 231 – 268

[34] Fejtová, M., Fejt, J., Štěpánková, O., "Eye as an Actuator", Lecture Notes in Computer Science, Volume 4061/2006, Springer Berlin / Heidelberg, 2006, ISBN: 978-3-540-36020-9, pages: 954-961

[35] Baudel, T., Beaudouin-Lafon, M., "Charade: remote control of objects using free-hand gestures", Communications of the ACM, Volume 36 , Issue 7, ACM, 1993, ISSN: 0001-0782, pages: 28 – 35

[36] Allen, J. F., Byron, D. K., Dzikovska, M., Ferguson, G., Galescu, L., Stent, A., "Toward Conversational Human-Computer Interaction", AI Magazine, Volume 22 , Issue 4, American Association for Artificial Intelligence, 2001, ISSN: 0738-4602, pages: 27 – 37

[37] Nakagawa, S., "A Survey on Automatic Speech Recognition", IEICE TRANSACTIONS on Information and Systems, Vol.E85-D No.3, IEICE, 2002, ISSN: 0916-8532, pages: 465 – 486

[38] Neural Impulse Actuator, OCZ Technology, 2009, web pages http://www.ocztechnology.com/products/ocz_peripherals/nia-neural_impulse_actuator

[39] Hochberg, L. R., Serruya, M. D., Friehs, G. M., Mukand, J. A., Saleh, M., Caplan, A. H., Branner, A., Chen, D., Penn, R. D., Donoghue, J. P., "Neuronal ensemble control of prosthetic devices by a human with tetraplegia", Nature, 2006, web pages, Available from http://www.nature.com/nature/journal/v442/n7099/abs/nature04970.html

[40] Mouse (computing), Wikipedia, The Free Encyclopedia, 2009, web pages http://en.wikipedia.org/wiki/Mouse_(computing)

[41] Logitech Performance Mouse MX, Logitech, 2009, web pages http://www.logitech.com/index.cfm/mice_pointers/mice/devices/5845&cl=us,en

[42] Keyboard (computing), Wikipedia, The Free Encyclopedia, 2009, web pages http://en.wikipedia.org/wiki/Keyboard_(computing)

[43] Bellis, M., "History of the Computer Keyboard", Inventors Guide, About.com, 2009,

web pages http://inventors.about.com/od/computerperipherals/a/computer_keyboa.htm

[44] Cassingham, R., "The Dvorak Keyboard", This is True Inc., 2007, web pages, Available from http://www.dvorak-keyboard.com/

[45] Liebowitz, S., Margolis, S. E., "The Fable of the Keys", Journal of Law and Economics, Vol. 30, No. 1, University of Chicago Press, 1990, pages: 1 – 25, available from http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1069950

[46] Brown, C.M., "Human-computer interface design guidelines", Ablex Pub. Corp., 1988, ISBN: 9780893913328, pages: 236

[47] Bailey, R.W., Bailey, L.M., "Reading speeds using RSVP", UI Design Newsletter – February, 1999, Human Factors International, 1999, web pages http://www.humanfactors.com/library/feb99.asp

[48] McWhirter, N., "The Guinness Book of World Records", 23rd US edition, Sterling Publishing, New York, 1985, ISBN: 978-0806902647, pages: 480

[49] Maltron Keyboards, PCD Maltron Ltd., 2007, web pages, Available from http://www.maltron.com/

[50] Hardware Image Gallery: Microsoft Keyboard Products, Microsoft, 2009, web pages http://www.microsoft.com/Presspass/gallery/hardware-keyboard.mspx

[51] Curran, K., Woods, D., Riordan, B. O., "Investigating text input methods for mobile phones", Telematics and Informatics, Volume 23 , Issue 1, Pergamon Press, Inc., 2006, ISSN: 0736-5853, pages: 1 – 21

[52] Hotas Cougar, Thrustmaster, 2009, web pages http://www.thrustmaster.com/product.aspx?ProductID=11

[53] Webcam C300, Logitech, 2009, web pages http://www.logitech.com/index.cfm/webcam_communications/webcams/devices/ 5863&cl=roeu,en

[54] Daugman, J., „Face and Gesture Recognition: Overview", IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume 19, Issue 7, IEEE, 1997, ISSN: 0162-8828, pages: 675-676

[55] Park, J.Y., Yi, J.H., "Gesture Recognition Based Interactive Boxing Game", International Journal of Information Technology, Vol.12, No. 7, Singapore Computer Society, 2006, ISSN: 0218-7957, pages: 36 – 44

[56] Tong, X., Duan, L., Xu, Ch., Tian, Q., Lu, H., "Local Motion Analysis and Its Application in Video based Swimming Style Recognition", Proceedings of the 18th International Conference on Pattern Recognition - Volume 02, IEEE Computer Society,

2006, ISSN: 1051-4651, ISBN: 0-7695-2521-0, pages: 1258 – 1261

[57] MacIntyre, B., Feiner, S., "Future Multimedia User Interfaces", Multimedia Systems, Volume 4, Number 5, Springer Berlin / Heidelberg, 1996, ISSN: 0942-4962, pages: 250 – 268

[58] Microphone, Wikipedia, The Free Encyclopedia, 2009, web pages http://en.wikipedia.org/wiki/Microphone

[59] PJ, "Audio-Technica AT8015a shotgun microphone", Wikimedia Commons, 2009, http://commons.wikimedia.org/wiki/File:Shotgun_microphone.jpg

[60] Rabiner, L., Juang, B.H., "Fundamentals of Speech Recognition", Prentice Hall PTR, 1993, ISBN: 978-0130151575, pages: 496

[61] History of computing hardware, Wikipeida, The Free Encyclopedia, 2009, web pages http://en.wikipedia.org/wiki/History_of_computing_hardware

[62] Canon PIXMA iX5000 Inkjet Printer, Canon, 2009, web pages http://www.canon-europe.com/For_Home/Product_Finder/Printers/Inkjet/PIXMA_iX5000/index.asp

[63] Computer monitor, Wikipedia, The Free Encyclopedia, 2009, web pages http://en.wikipedia.org/wiki/Computer_monitor

[64] Monochrome Display Adapeter, Wikipedia, The Free Encyclopedia, 2009, web pages http://en.wikipedia.org/wiki/Monochrome_Display_Adapter

[65] IBM PC 5150, Wikimedia Commons, 2009, web pages http://commons.wikimedia.org/wiki/File:IBM_PC_5150.jpg

[66] Samsung XL2370, Samsung, 2009, web pages http://www.samsung.com/cz/consumer/ monitors-peripherals-printers/monitor/lcd/LS23EFPKFV/EN/ index.idx?pagetype=prd_detail

[67] Sony Ericsson C702, Sony Ericsson, 2009, web pages http://www.sonyericsson.com/cws/products/mobilephones/overview/c702?cc=cz&lc=cs

[68] Eee PC T91, ASUSTeK Computer Inc., 2009, web pages http://eeepc.asus.com/global/productt91.html?n=0

[69] Sony Ercisssson Xperia X10, Sony Ericsson, 2009, web pages http://www.sonyericsson.com/cws/products/mobilephones/overview/xperiax10?cc=cz&lc=cs#a

[70] Loudspeaker, Wikipedia, The Free Encyclopedia, 2009, web pages http://en.wikipedia.org/wiki/Loudspeaker

[71] Logitech Sound System Z520, Logitech, 2009, web pages http://www.logitech.com/index.cfm/speakers_audio/home_pc_speakers/devices/5859&cl

=roeu,en

[72] PAC Mate Portable Braille Display, Freedom Scientific, 2009, web pages http://www.freedomscientific.com/products/fs/pacmate-braille-product-page.asp

[73] I-Tech Virtual Laser Keyboard, Power Positioning Ltd, 2009, web pages http://www.virtual-laser-keyboard.com/

[74] Olsen, H., "Supporting customers' decision-making process", GUUUI, ISSUE 06 - Q3 2003, web pages: http://www.guuui.com/issues/02_03.php

[75] Eckard, R., "Survey: Family Cars - The Controversy of Style vs. Functionality", Kelley Blue Book, 2009, web pages: http://mediaroom.kbb.com/index.php?s=43&item=18

[76] Norman, D., "Complexity versus difficulty: where should the intelligence be?", Proceedings of the 7th international conference on Intelligent user interfaces, ACM New York, USA, 2002, ISBN: 1-58113-459-2, pages: 4 – 4

[77] Adobe Photoshop, Adobe Systems Incorporated, 2007, web pages, Available from http://www.adobe.com/products/photoshop/

[78] HP Photosmart Essential 2.0, Hewlett-Packard Development Company, L.P., 2007, web pages, Available from http://www.hp.com/united-states/consumer/digital_photography/ free/software/index.html

[79] Yoon, H, Kim, E, Lee, M, "A User context awareness Model for mobile environment processing", 5th ACIS International Conference on Software Engineering Research, Management & Applications, 2007, ISBN: 0-7695-2867-8, pages: 293 – 296

[80] Bennett, J., "Managing to Meet Usability Requirements: Establishing and Meeting Software Development Goals", Visual Display Terminals, Englewood Cliffs, Prentice-Hall, 1984, pages: 161-184

[81] Quesenbery, W., "What Does Usability Mean: Looking Beyond 'Ease of Use'", in Proceedings of the 48th Annual Conference Society for Technical Communication, Chicago, 2001

[82] International Standard ISO 9241-11, "Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability", ISO 9241-11:1998(E), web pages: http://www.eihms.surrey.ac.uk/robens/erg/methods%20module/ ISO_9241-11.pdf

[83] Windows API, Microsoft Corporation, 2007, web pages, Available from http://msdn2.microsoft.com/en-US/library/aa383750.aspx

[84] Windows Forms, Microsoft Corporation, 2007, web pages, Available from http://msdn2.microsoft.com/en-us/netframework/aa497342.aspx

[85] .NET Framework, Microsoft Corporation, 2007, web pages, Available from http://msdn2.microsoft.com/en-us/netframework/default.aspx

[86] Integrated development environment, Wikipedia, The Free Encyclopedia, 2009, web pages: http://en.wikipedia.org/wiki/Integrated_development_environment

[87] Visual Studio Developer Center, Microsoft Corporation, 2007, web pages, Available from http://msdn2.microsoft.com/en-us/vstudio/default.aspx

[88] MFC Reference, Microsoft Corporation, 2007, web pages, Available from http://msdn2.microsoft.com/en-us/library/d06h2x6e(vs.71).aspx

[89] Scripting language, Wikipedia, The Free Encyclopedia, 2009, web pages: http://en.wikipedia.org/wiki/Scripting_language

[90] TCL Developer Xchange, 2009, web pages: http://www.tcl.tk/

[91] The Perl Programming Language, 2009, web pages: http://www.perl.org/

[92] PHP: Hypertext preprocessor, The PHP Group, 2009, web pages: http://php.net/

[93] Coombs, J. H., Renear, A. H. Renear, DeRose, S. J., "Markup systems and the future of scholarly text processing", Communications of the ACM 30, 1987, pages: 933-947

[94] Home of the User Interface Markup Language, Organization for the Advancement of Structured Information Standards, 2002, web pages, Available from http://www.uiml.org/

[95] Abrams, M., Phanouriou, C., Batongbacal, A.L., Williams, S.M., Shuster, J.E., „UIML: An Appliance-Independent XML User Interface Language", in Proceedings of the Eighth International World Wide Web Conference, 1999, web pages, Available from http://www8.org/w8-papers/5b-hypertext-media/uiml/uiml.html

[96] Extensible Markup Language (XML), World Wide Web Consortium, 2008, web pages, Available from http://www.w3.org/XML/

[97] Mozilla Project, Mozilla Corporation, 2008, web pages, Available from http://www.mozilla.org/

[98] Cascading Style Sheets, World Wide Web Consortium, 2008, web pages, Available from http://www.w3.org/Style/CSS/

[99] Kent, P., Kent, J., "Official Netscape Javascript 1.2", Ventana Communications Group, 1997, ISBN: 1566046750, 520 pages

[100] Document Object Model, World Wide Web Consortium, 2008, web pages, Available from http://www.w3.org/DOM/

[101] Mozilla Layout Engine, Mozilla Corporation, 2008, web pages, Available from http://www.mozilla.org/newlayout/

[102] Windows Presentation Foundation, Microsoft Corporation, 2008, web pages, Available from http://msdn.microsoft.com/en-us/netframework/aa663326.aspx

[103] The GTK+ Project, The GTK+ Team, 2008, web pages, Available from http://www.gtk.org/

[104] OpenLaszlo, OpenLaszlo Systems, Inc., web pages, Available from http://www.openlaszlo.org/

[105] Helander, M. G., Landauer, T. K., Prabhu, P. V., "Handbook of Human-Computer Interaction", North Holland, 1997, ISBN: 978-0444818621, pages: 1602

[106] Molin, L., "Wizard-of-Oz prototyping for co-operative interaction design of graphical user interfaces", in Proceedings of the Third Nordic Conference on Human-Computer interaction, Tampere, Finland, 2004, pages: 425-428

[107] Clerckx, T., "Integrating Task Models in Automatic User Interface Generation", Master Thesis, 2003, web pages, Available from http://research.edm.luc.ac.be/tclerckx/clerckxconinx-integratingtaskmodels.pdf

[108] Baron, M., Girard, P., "SUIDT, A task model based GUI-Builder", in Proceedings of the TAMODIA : Task, 2002, pages: 64–71

[109] ISO, Information Processing Systems, Open Systems Interconnection, "Lotos - A Formal Description Technique Based on the Temporal Ordering of Observational Behavior", 1989, ISO 8807:1989

[110] Paterno, F., Mancini, C., Meniconi, S., "ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models", in Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction, Sydney, 1997, pages: 362-369

[111] Bodart, F., Hennebert, A.M., Leheureux, J.M., Provot, I., Vanderdonckt, J., "A Model-Based Approach to Presentation: A Continuum from Task Analysis to Prototype", in Proceedings of the First International Eurographics Workshop - Design, Specification, and Verification of Interactive Systems, 1994, pages: 25–39

[112] Dlodlo, N., Bamford, C., "Separating Application Functionality from the User Interface in a Distributed Environment", In: Proceedings of the 22nd EUROMICRO Conference, 1996, Prague, Czech Republic, pages: 248 – 256

[113] Fowler, M., "Patterns of Enterprise Application Architecture", Addison-Wesley Professional, 2002, ISBN: 0321127420, 560 pages

[114] Burbeck, S., "Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC), Smalltalk Archive, University of Illinois in Urbana-Champaign (UIUC), 1997, web pages, Available from: http://st-www.cs.uiuc.edu/users/smarch/st-

docs/mvc.html

[115] Potel, M., "MVP: Model-View-Presenter, The Taligent Programming Model for C++ and Java", Taligent, Inc., 1996, Available: http://www.wildcrest.com/Potel/Portfolio/mvp.pdf

[116] Puerta, A., Eisenstein, J., "XIML: A Common Representation for Interaction Data", in Proceedings of IUI2002, ACM Press, 2002, pages 214–215

[117] Schneider, K. A., Cordy, J. R., "AUI: A Programming Language for Developing Plastic Interactive Software", in Proceedings of HICSS–35', 2002, pages 281 – 291

[118] Jelinek, J., Slavik, P., "GUI Generation from Annotated Source Code", in Proceedings of Tamodia'04, Prague, Czech Republic, 2004

[119] Randem, H. O., Jokstad, H., Linden, T., Kvilesjo, H., Rekvin, S., Hornæs, A., "ProcSee - The Picasso Successor", Enlarged Halden Programme Group Meeting, Lillehammer, Norway, 2005

[120] Gajos, K., Weld, D. S., Wobbrock, J. O., "Decision-Theoretic User Interface Generation", in Proceedings of AAAI'08, Chicago, IL, USA, 2008

[121] Roth, S. F., Mattis, J., "Data characterization for intelligent graphics presentation", in Proceedings of the 1990 Conference on Human Factors in Computing Systems, , New Or-leans, LA, 1990, pages 193–200

[122] Zhou, M. X., Feiner, S. K., "Data Characterization For Automatically Visualizing Heterogeneous Information", in Proceedings of INFOVIS '96 (1996 IEEE Symposium on Information Visualization), San Francisco, CA, 1996, pages 13-20

[123] Mackinlay, J., "Automating the design of graphical presentations of relational information", ACM Transactions on Graphics, 1986, pages 110 – 141

[124] Wehrend, R., Lewis, C., "A problem-oriented classification of visualization techniques", in Proceedings of the First IEEE Conference on Visualization: Visualization 90. IEEE, Los Alamitos, CA, 1990, pages 139–143

[125] Feiner, S., "Improvise – knowledge based visual presentation system", Computer Graphics & User Interfaces Lab, Columbia University, web pages, Available at: http://www1.cs.columbia.edu/~zhou/project/project.html

[126] Java ™, Sun Microsystems, Inc., 2009, web pages http://www.java.com/

[127] Arens, Y., Hovy, E., Vossers, M., "The knowledge underlying multimedia presentations", Intelligent multimedia interfaces, American Association for Artificial Intelligence, USA, 1993, pages 280 – 306

[128] Ullman, J. D., "Principles of Database and Knowledge-Base Systems", Computer Science Press, New York, 1990, ISBN: 071678162X, pages: 511

[129] Castner, S., "A task-analytic approach to the automated design of graphic presentations", ACM Transactions on Graphics (TOG), Volume 10, 1991, pages 111 – 151

[130] Pope, G. E., Krasner, S. T., "A cookbook for using the Model-View-Controller user-interface paradigm in Smalltalk-80", Jour of Object-oriented programming 1 (3), 1988, pages: 26-49

[131] Berlege, T., "Using TAPS to separate the user interface from the application code", ACM SIGGRAPH UIST'92, 19925, pages: 191-198

[132] Nihe, K., Seki, K., Nakamura, H., Yagishita, R., Shimojima, T., "Distributed object system framework ORB", NEC Research and Development, 1994, pages 292-297

[133] PropertyDescriptor, MSDN, Microsoft Corporation, 2009, web pages, Available from: http://msdn.microsoft.com/en-us/library/

system.componentmodel.propertydescriptor.aspx

[134] Abowd, G. D., Dey, A. K., Brown, P. J., Davies, N., Smith, M., Steggles, P., "Towards a Better Understanding of Context and Context-Awareness", In Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing, Germany, 1999, ISBN: 3540665501, pages: 304 – 307

[135] Vanderdonckt, J. M., Chieu, Ch. K., Bouillon, L., Trevisan, D., "Model-based Design, Generation, and Evaluation of Virtual User Interfaces", In Proceedings of the ninth international conference on 3D Web technology, Monterey, California, 2004, ISBN: 1581138458, pages: 51 – 60

[136] Vanderdonckt, J. M., Bodart, F., "Encapsulating knowledge for intelligent automatic interaction objects selection", In Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems, Amsterdam, The Netherlands, 1993, ISBN: 0897915755, pages: 424 – 429

[137] Armstrong, D. J., "The quarks of object-oriented development", Communications of the ACM, Volume 49 , Issue 2, New York, NY, USA, 2006, ISSN: 00010782, pages 123 – 128

[138] Lok, S., Feiner, S., "A survey of automated layout techniques for information presentations", In Proceedings of SmartGraphics Symposium '01, 2001, pages: 61-68

[139] Lok, S., Feiner, S., Ngai, G., "Evaluation of visual balance for automated layout", In Proceedings of the 9th international conference on Intelligent user interface, Funchal,

Madeira, Portugal, 2004, ISBN 1581138156, pages: 101 − 108

[140] Lee, W. L., Green, M., "A layout framework for 3D user interfaces", In Proceedings of the ACM symposium on Virtual reality software and technology, Monterey, CA, USA, 2005, ISBN: 1595930981, pages: 96 − 105

[141] Lee, W. L., Green, M., "Automatic layout for 3D user interfaces construction", In Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications, Hong Kong, China, 2006, ISBN: 1595933247, pages: 113 − 120

[142] Alonso, F., Fuertes, J. L., González, Á. L., Martínez, L., "User-Interface Modelling for Blind Users", Springer Berlin / Heidelberg, 2008, ISBN: 9783540705390, pages: 789-796

10 Appendix – Simple Media Player Example

This example demonstrates the proposed taxonomy on simple media player example running on desktop environment. Media player example (see Figure 10.1) consists of the MediaPlayer class, a Playlist object and a PlaylistEntry.
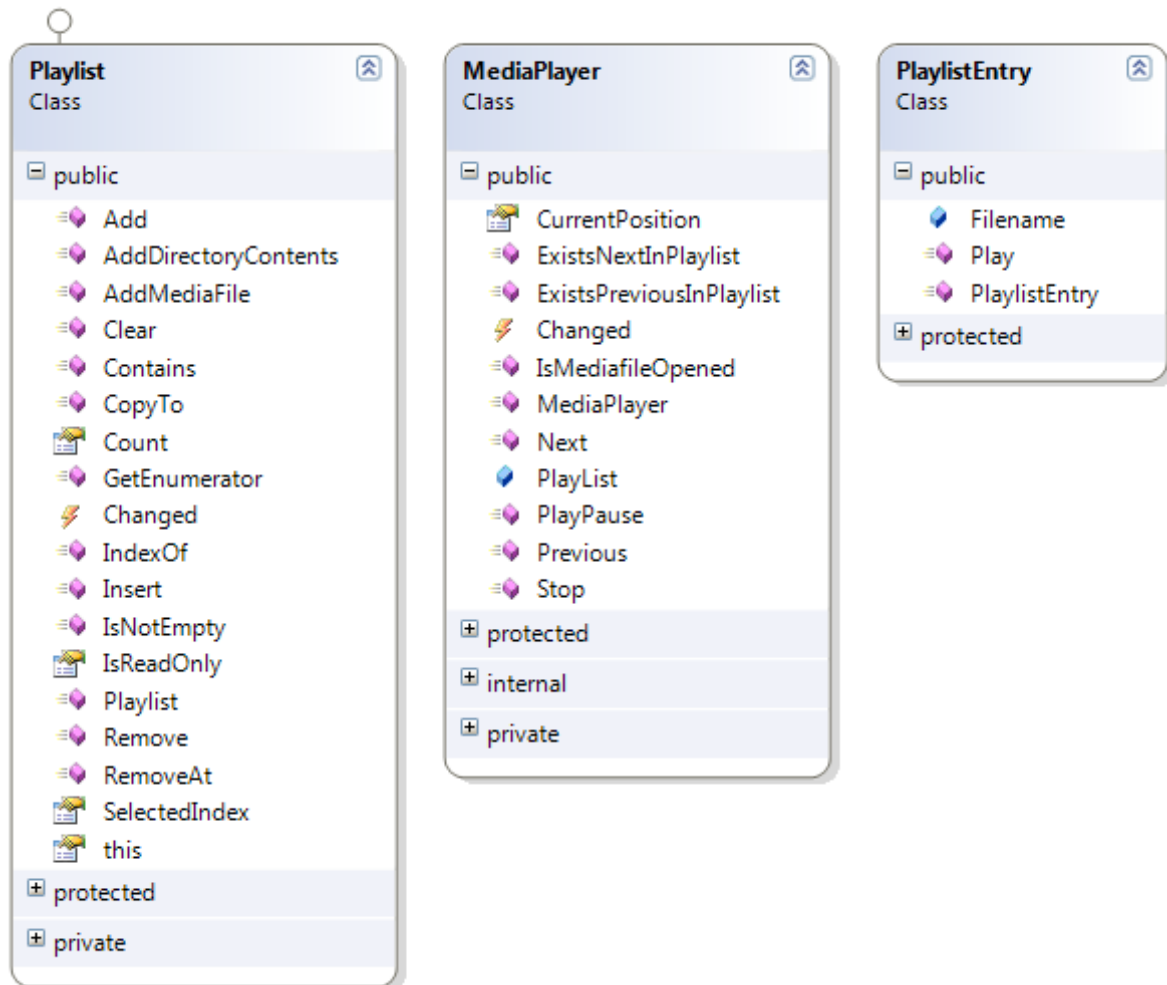
Figure 10.1 Simple Media Player classes

```
[CodeCharacterization.FirstObjectSelected]
[DataCharacterization.DataType(DataCharacterization.DataTypes.Atomic)]
[DataCharacterization.DataDomain(DataCharacterization.DataDomains.Entity)]
[DataCharacterization.DataForm(DataCharacterization.DataShapes.Shaped)]
[CodeCharacterization.CodeName("Media Player")]
public class MediaPlayer
{
    [CodeCharacterization.CodeName("Playlist")]
    public Playlist PlayList;

    [CodeCharacterization.OnChanged]
    public event EventHandler Changed { … }

    [CodeCharacterization.CodeCategory("Player", "PlayPause")]
    [CodeCharacterization.CodeImportance(0)]
    [CodeCharacterization.CodeSense(CodeCharacterization.CodeSenseTypes.Command)]
    [CodeCharacterization.CodeDependence("IsMediafileOpened")]
    public void PlayPause() { … }

    [CodeCharacterization.CodeCategory("Player", "Stop")]
    [CodeCharacterization.CodeImportance(0)]
    [CodeCharacterization.CodeSense(CodeCharacterization.CodeSenseTypes.Command)]
    [CodeCharacterization.CodeDependence("IsMediafileOpened")]
    public void Stop() { … }

    [CodeCharacterization.CodeCategory("Navigation", "Next")]
    [CodeCharacterization.CodeSense(CodeCharacterization.CodeSenseTypes.Command)]
    [CodeCharacterization.CodeImportance(0)]
    [CodeCharacterization.CodeDependence("ExistsNextInPlaylist")]
    public void Next() { … }

    [CodeCharacterization.CodeCategory("Navigation", "Previous")]
    [CodeCharacterization.CodeSense(CodeCharacterization.CodeSenseTypes.Command)]
    [CodeCharacterization.CodeImportance(0)]
    [CodeCharacterization.CodeDependence("ExistsPreviousInPlaylist")]
    public void Previous() { … }

    [DataCharacterization.DataType(DataCharacterization.DataTypes.Atomic)]
    [DataCharacterization.DataDomain(DataCharacterization.DataDomains.Measurement,
SpecificName="TimeLine")]
    [DataCharacterization.DataContinuity(DataCharacterization.ContinuityTypes.Continuous)]
    [DataCharacterization.DataTransience(DataCharacterization.DataTransienceType.Dynamic)]
    [DataCharacterization.DataImportance(0)]
    [CodeCharacterization.CodeName("Time Line")]
    [CodeCharacterization.CodeDependence("IsMediafileOpened")]
    [CodeCharacterization.ParameterRange(0,0)]
    public ulong CurrentPosition { … }
}
```

Figure 10.2 Characterized source code of the simple MediaPlayer class

```
[DataCharacterization.DataType(DataCharacterization.DataTypes.Structure)]
[DataCharacterization.DataDomain(DataCharacterization.DataDomains.Entity)]
[DataCharacterization.DataTransience(DataCharacterization.DataTransienceType.Static)]
public class PlaylistEntry
{
    [DataCharacterization.DataType(DataCharacterization.DataTypes.Atomic)]
    [DataCharacterization.DataDomain(DataCharacterization.DataDomains.Concept)]
    [CodeCharacterization.CodeName("File name")]
    public string Filename;

    [CodeCharacterization.CodeName("Play")]
    [CodeCharacterization.CodeDescription("Starts playing of this media file.")]
    [CodeCharacterization.CodeSense(CodeCharacterization.CodeSenseTypes.Command,
Default=true)]
    public void Play(){ … }
}
```

Figure 10.3 Characterized source code of the PlaylistEntry

```
[DataCharacterization.DataType(DataCharacterization.DataTypes.Atomic)]
[DataCharacterization.DataDomain(DataCharacterization.DataDomains.Entity)]
[DataCharacterization.DataForm(DataCharacterization.DataShapes.Shaped)]
[DataCharacterization.DataContinuity(DataCharacterization.ContinuityTypes.Discreete)]
[DataCharacterization.DataOrdering(DataCharacterization.DataOrderingTypes.Quantitative)]
[DataCharacterization.DataRole(DataCharacterization.DataRoleTypes.Identification)]
[DataCharacterization.DataSense(DataCharacterization.DataSenseTypes.List)]
[DataCharacterization.DataImportance(0)]
public class Playlist : IList<PlaylistEntry>
{
    [CodeCharacterization.OnChanged]
    public event EventHandler Changed { … }

    [CodeCharacterization.CodeName("Add media file")]
    [CodeCharacterization.CodeDescription("Adds media file to the playlist")]
    [CodeCharacterization.CodeCategory("Collection","Add")]
    [CodeCharacterization.CodeImportance(1)]
    public void AddMediaFile(
        [CodeCharacterization.ParameterName("Media File")]
        FileInfo file)
    { … }

    [CodeCharacterization.CodeName("Add media file")]
    [CodeCharacterization.CodeDescription("Adds media files … to the playlist")]
    [CodeCharacterization.CodeCategory("Collection", "Add")]
    [CodeCharacterization.CodeImportance(1)]
    public void AddDirectoryContents(
        [CodeCharacterization.ParameterName("Directory with media files.")]
        DirectoryInfo directory)
    { … }

    [CodeCharacterization.CodeName("Remove")]
    [CodeCharacterization.CodeDescription("Removes selected item in playlist.")]
    [CodeCharacterization.CodeCategory("Collection", "Remove")]
    public void RemoveAt(int index) { … }

    [CodeCharacterization.CodeCategory("Collection", "Remove All")]
    [CodeCharacterization.CodeDependence("IsNotEmpty")]
    public void Clear() { … }
}
```

Figure 10.4 Characterized source code of the Playlist class

After the loading of the source code files, information about the characterized classes was loaded into the characterization tree (see Figure 10.5). Notice that only the methods that have been characterized were loaded and are contained in the characterization tree. This means that not every data or method of the classes will play any role in the user interface. In the characterization tree in Figure 10.5, only few characterization attributes are displayed.
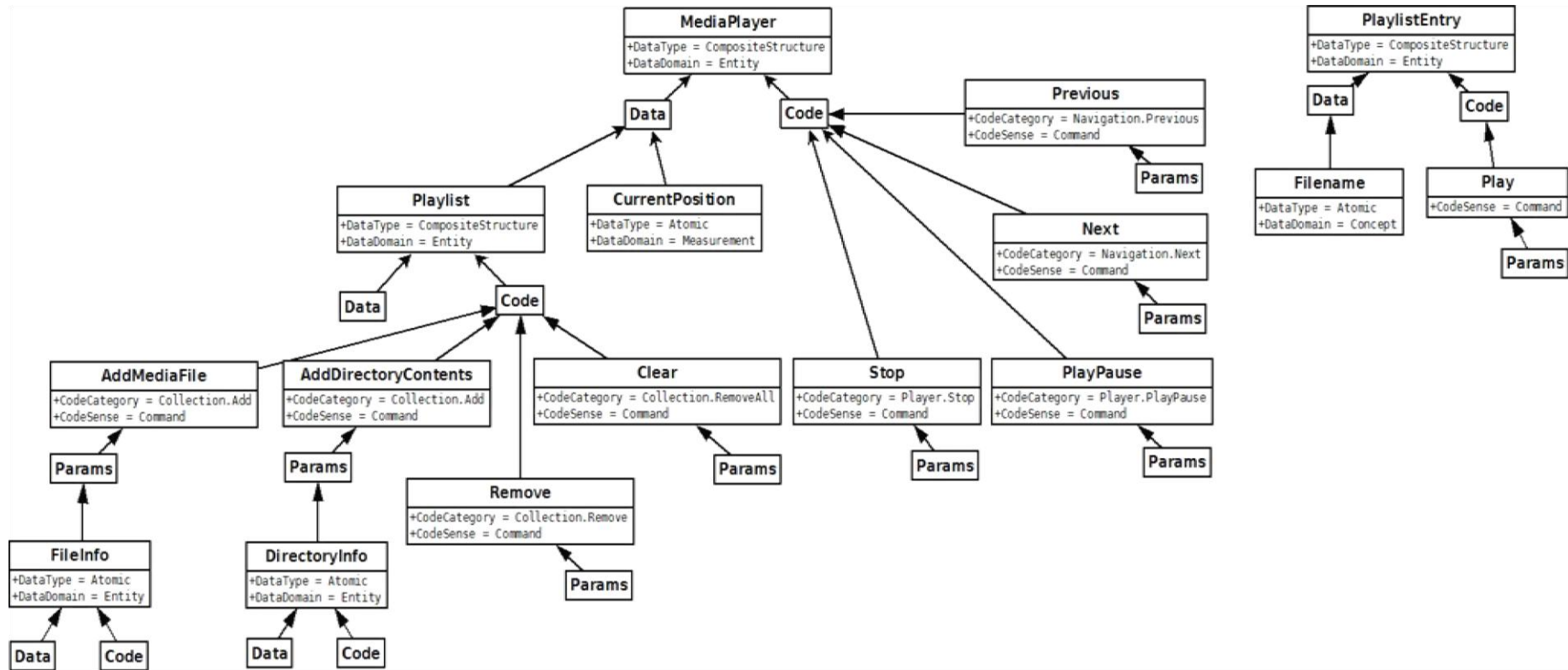
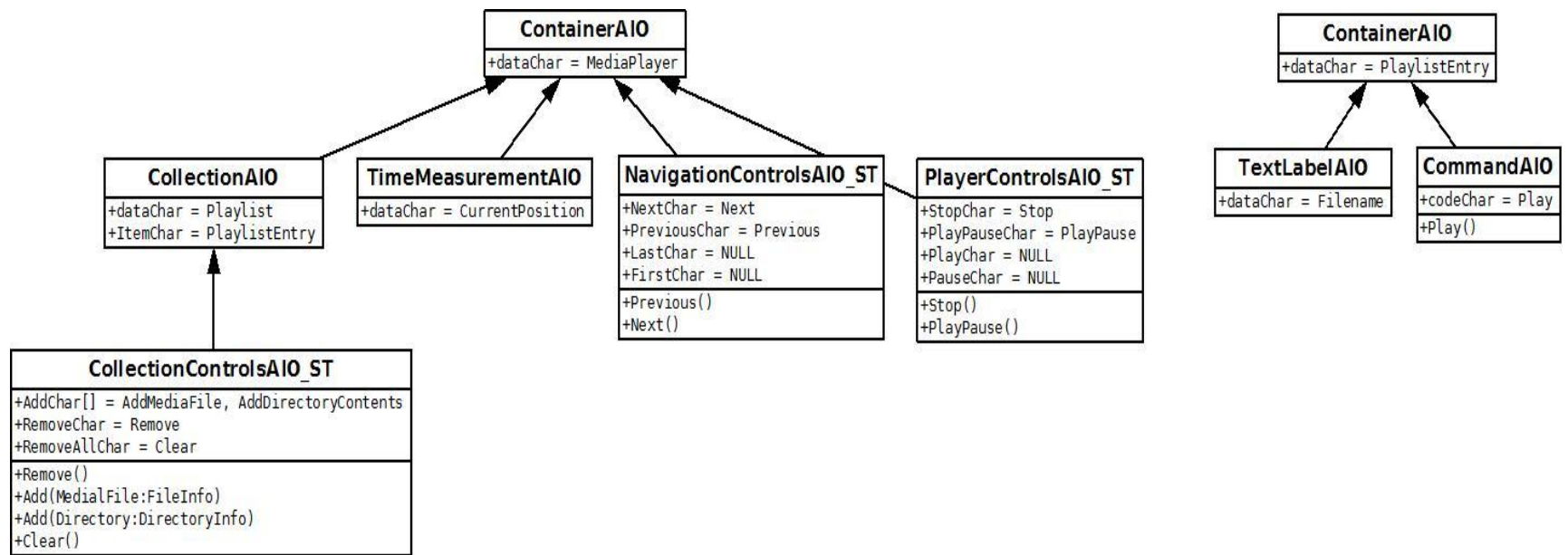**Figure 10.5 Simple Media Player characterization tree**

Figure 10.6 Simple Media Player AIOs

127

Conversion of the characterization tree to AIOs is demonstrated in the Figure 10.6. The main media player object is represented by a container, the playlist as a collection, and the current position as a time measurement (both are sub-objects of the media player class). The methods were linked together into three main AIOs thanks to smart template concept – smart templates group together commands with the same category. The "media player smart template" contains commands "play" and "stop", "the navigation smart template" contains commands "next" and "previous", and the "list smart template" contains commands "add", "remove", and "clear". The playlist entry was placed separately during collection initialization because it is used for the internal representation of the collection items and is not explicit part of the media player interface.
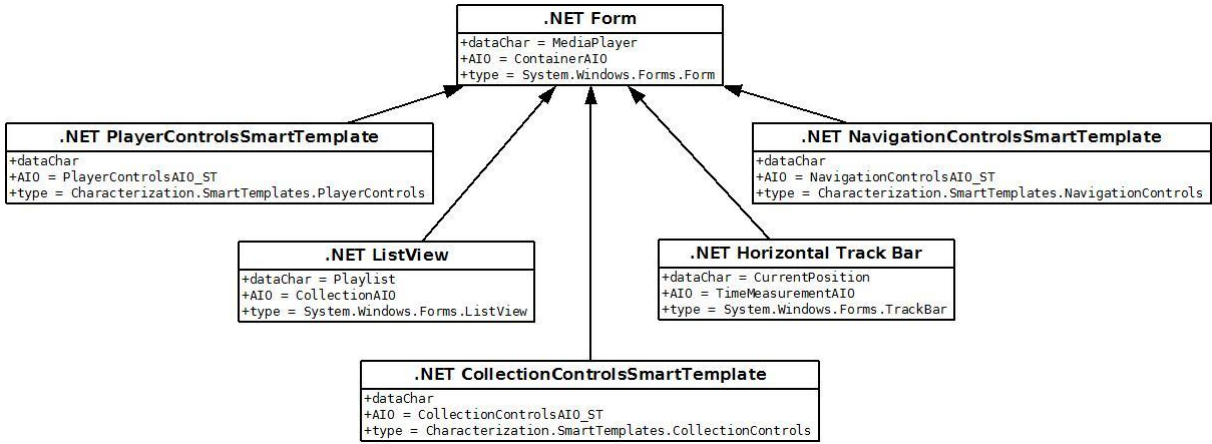
Figure 10.7 Simple Media Player CIOs

Figure 10.7 demonstrates the result of the AIOs conversion to CIOs. Main class of the media player is represented by a Form (window), containing instances of a ListView for the playlist, a track bar for the current position and the smart templates for the categorized commands.

Figure 10.8 shows graphical representation of each CIO from the CIO tree. Instantiating all CIOs from the CIO tree in Figure 10.7 creates final user interface that can be seen in Figure 10.9. All CIOs were placed in top-bottom and left-right order which represents highest to lowest importance. The playlist fills major part of the window because track bar and smart templates are of fixed sizes. Playlist is the first because it has higher importance than other data or methods. Collection controls span from left to right because .NET Collection Controls Smart Template is composed as a toolbar. The Navigation and Player Controls Smart Templates were placed besides each other because they have same importance. Horizontal Track Bar is placed last and spans from left to right filling rest of the free space on the form.

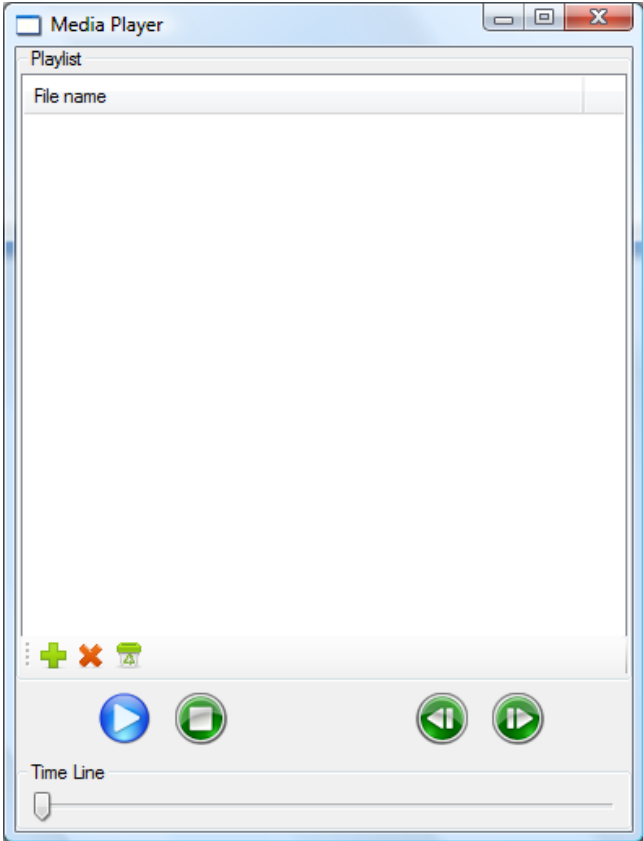**Figure 10.8 A graphical representation of the CIOs of the Simple Media Player Example**

Figure 10.9 Simple Media Player user interface

11 Appendix – A Regulator Example

This example demonstrates the proposed taxonomy on simple regulator example running on desktop environment. The Regulator example (see Figure 11.1) consists of the Regulator class, a RegulatorGains class, and a Setpoint class.
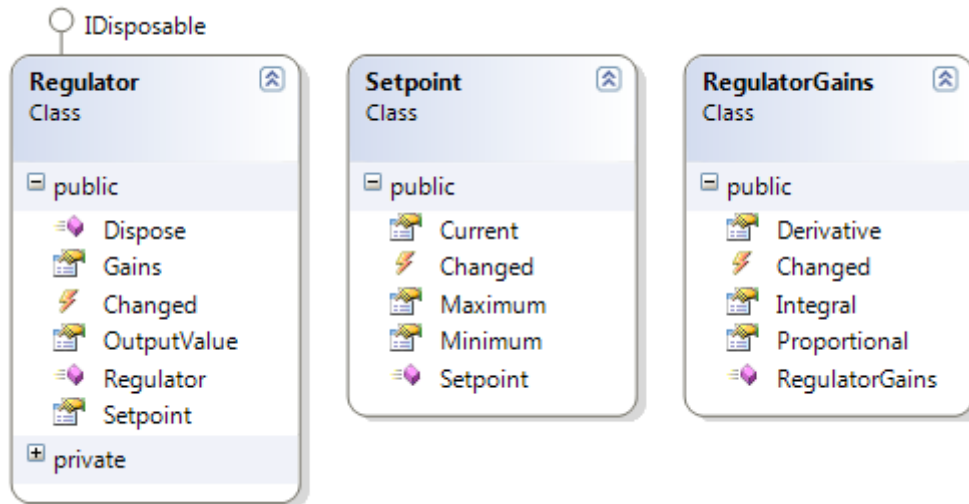

Figure 11.1 The Regulator example classes.

```csharp
[DataType(DataTypes.Structure)]
[DataDomain(DataDomains.Entity)]
[DataForm(DataShapes.Shaped)]
[DataTransience(DataTransienceType.Static)]
public class RegulatorGains
{
    [DataType(DataTypes.Atomic)]
    [DataDomain(DataDomains.Concept)]
    [DataForm(DataShapes.None)]
    [DataImportance(0)]
    [DataScalability(DataScalabilityTypes.Amounts)]
    [DataSense(DataSenseTypes.Label)]
    [DataTransience(DataTransienceType.Dynamic)]
    public double Proportional { … }

    [DataType(DataTypes.Atomic)]
    [DataDomain(DataDomains.Concept)]
    [DataForm(DataShapes.None)]
    [DataImportance(0)]
    [DataScalability(DataScalabilityTypes.Amounts)]
    [DataSense(DataSenseTypes.Label)]
    [DataTransience(DataTransienceType.Dynamic)]
    public double Integral { … }

    [DataType(DataTypes.Atomic)]
    [DataDomain(DataDomains.Concept)]
    [DataForm(DataShapes.None)]
    [DataImportance(0)]
    [DataScalability(DataScalabilityTypes.Amounts)]
    [DataSense(DataSenseTypes.Label)]
    [DataTransience(DataTransienceType.Dynamic)]
    public double Derivative { … }

    [CodeCharacterization.OnChanged]
    public event EventHandler Changed;
}
```

```csharp
[CodeCharacterization.FirstObjectSelected]
[DataCharacterization.DataType(DataCharacterization.DataTypes.Structure)]
[DataCharacterization.DataDomain(DataCharacterization.DataDomains.Entity)]
[DataCharacterization.DataForm(DataCharacterization.DataShapes.Shaped)]
[CodeCharacterization.CodeName("Regulator")]
public class Regulator : IDisposable
{
    [DataType(DataTypes.Atomic)]
    [DataDomain(DataDomains.Measurement, SpecificName="Temperature")]
    [DataImportance(3)]
    [DataSense(DataSenseTypes.Symbol)]
    [DataTransience(DataTransienceType.Dynamic)]
    [CodeCharacterization.CodeName("Output")]
    [CodeCharacterization.ParameterDefaultValue("10")]
    public double OutputValue { … }


    [CodeCharacterization.CodeName("Setpoint")]
    public Setpoint Setpoint { … }

    [CodeCharacterization.CodeName("Gains")]
    public RegulatorGains Gains { … }

    [CodeCharacterization.OnChanged]
    public event EventHandler Changed;
}
```

Figure 11.3 Characterized source code of the Regulator class

```csharp
[DataCharacterization.DataType(DataCharacterization.DataTypes.Structure)]
[DataCharacterization.DataDomain(DataCharacterization.DataDomains.Entity)]
[DataCharacterization.DataForm(DataCharacterization.DataShapes.Shaped)]
[DataCharacterization.DataTransience(DataTransienceType.Static)]
public class Setpoint
{
    [DataType(DataTypes.Atomic)]
    [DataDomain(DataDomains.Measurement, SpecificName="Temperature")]
    [DataForm(DataShapes.None)]
    [DataImportance(0)]
    [DataRole(DataRoleTypes.Comparison)]
    [DataScalability(DataScalabilityTypes.Amounts)]
    [DataSense(DataSenseTypes.Symbol)]
    [DataTransience(DataTransienceType.Dynamic)]
    public float Current { … }

    [DataType(DataTypes.Atomic)]
    [DataDomain(DataDomains.Measurement, SpecificName = "Temperature")]
    [DataForm(DataShapes.None)]
    [DataImportance(0)]
    [DataRole(DataRoleTypes.Comparison)]
    [DataScalability(DataScalabilityTypes.Amounts)]
    [DataSense(DataSenseTypes.Symbol)]
    [DataTransience(DataTransienceType.Dynamic)]
    [DataRelationAttributive("this", "Current", Type = AttributiveRelationTypes.Minimum)]
    public float Minimum { … }

    [DataType(DataTypes.Atomic)]
    [DataDomain(DataDomains.Measurement, SpecificName = "Temperature")]
    [DataForm(DataShapes.None)]
    [DataImportance(0)]
    [DataRole(DataRoleTypes.Comparison)]
    [DataScalability(DataScalabilityTypes.Amounts)]
    [DataSense(DataSenseTypes.Symbol)]
    [DataTransience(DataTransienceType.Dynamic)]
    [DataRelationAttributive("this", "Current", Type = AttributiveRelationTypes.Maximum)]
    public float Maximum { … }

    [CodeCharacterization.OnChanged]
    public event EventHandler Changed;
}
```

Figure 11.4 Characterized source code of the Setpoint class

131

After the loading of the source code files, information about the characterized classes are loaded into the characterization tree (see Figure 11.5). Notice that only the methods and data that have been characterized were loaded and are contained in the characterization tree. This means that not every data or method of the classes will play any role in the user interface. In a characterization tree in the Figure 11.5, only few characterization attributes are displayed.

Conversion of the characterization tree to AIOs is demonstrated in the Figure 11.6. The main regulator object is represented by a container because it has the *FirstObjectSelected* attribute and is entity, structure. The *Gains* is also represented as a container because it is also entity, structure. All the properties of the *Gains* (*Derivative*, *Integral*, and *Proportional*) are represented by a *TextInputAIO* because it is characterized as atomic concept with unspecified shape, and has label sense. The *OutputValue* property is represented by a *TemperatureAIO* because it is characterized as atomic temperature measurement with symbol sense. The *Setpoint* property is represented by a container because it is, as well as *Gains*, entity, structure. All the properties of the *Setpoint* (*Current*, *Maximum*, and *Minimum*) are represented by *TemperatureAIO* because it is characterized as atomic temperature measurement with symbol sense. The *Maximum* and *Minimum* contain attributive relation to the *Current* describing attributes of the Current property.
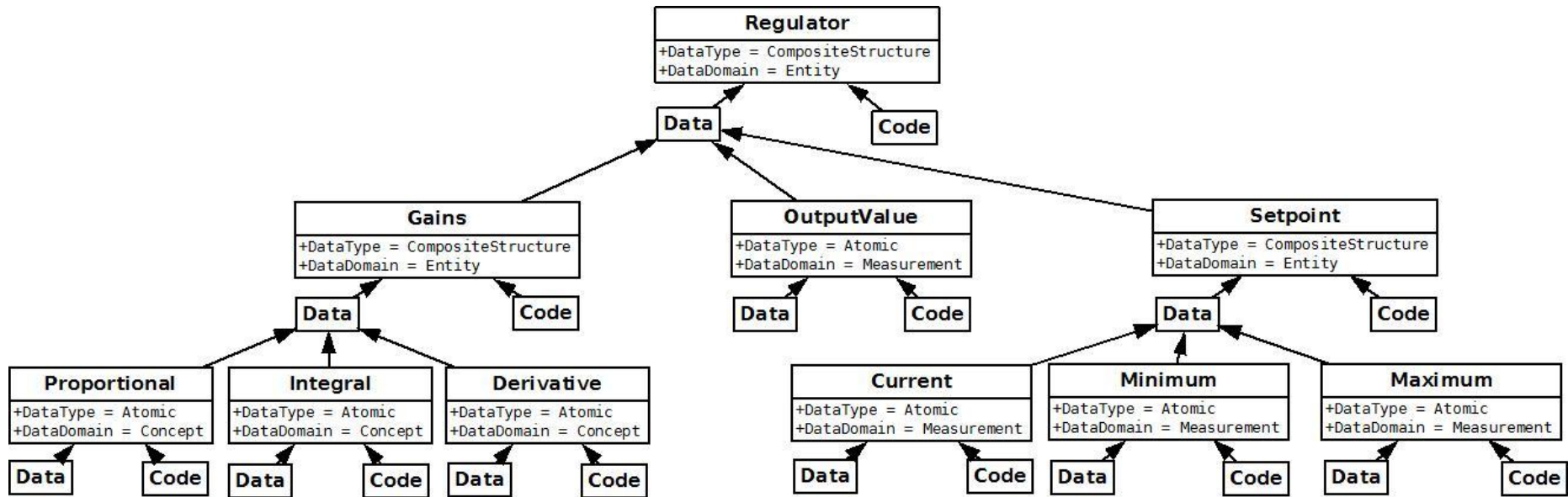
Figure 11.5 The Regulator characterization tree
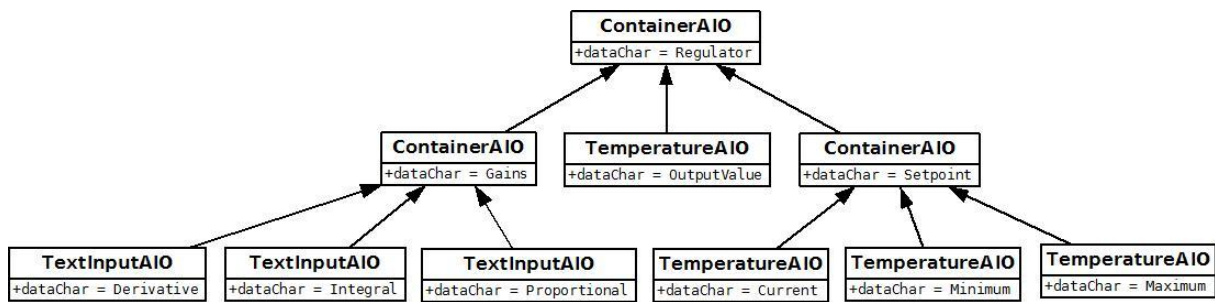
133

Figure 11.6 The Regulator example AIOs

Figure 11.7 demonstrates the result of the AIOs conversion to CIOs. The *Regulator* class is represented by a Form (window) because it is the topmost object in the hierarchy. The *Gains* container is converted to a GroupBox to reflect its internal structure. The *Derivative*, *Integral* and *Proportional TextInputAIOs* were converted to TextBox because all properties implement getters and setters and the TextBox allow required data input. Data type validity is performed when converting string to the data type of the properties. The *OutputValue* is converted to Vertical Track Bar because it is the only CIO associated to the AIO. The idea used to build the CIO comes from a thermometer common layout – vertical. The *Setpoint* container is converted to a GroupBox to reflect its internal structure. The *Current*, *Minimum*, and *Maximum* TemperatureAIOs are converted to the Vertical Track Bar as well as *OutputValue*. The *Current* property represents desired temperature, the *Maximum* and *Minimum* properties allow dynamic change of the *Current* limits.
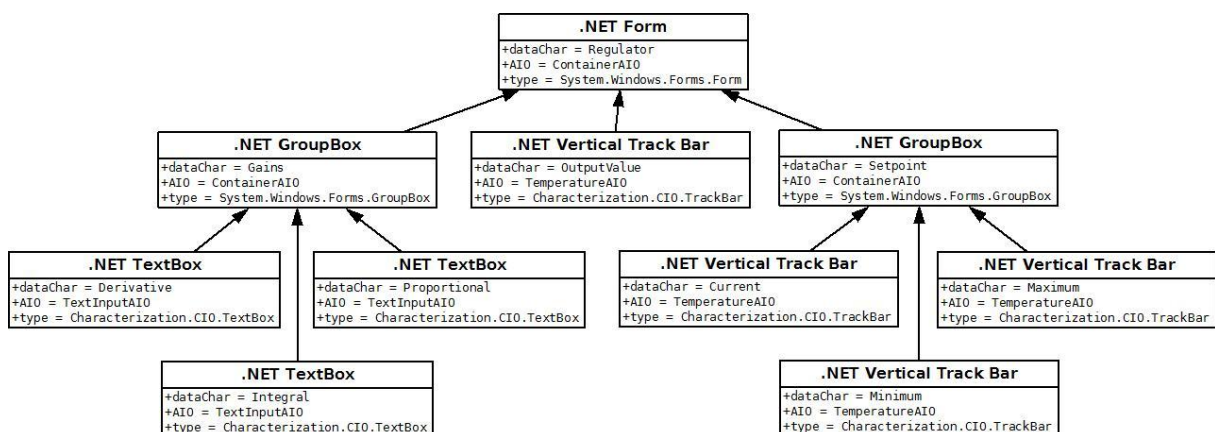

Figure 11.7 The Regulator example CIOs

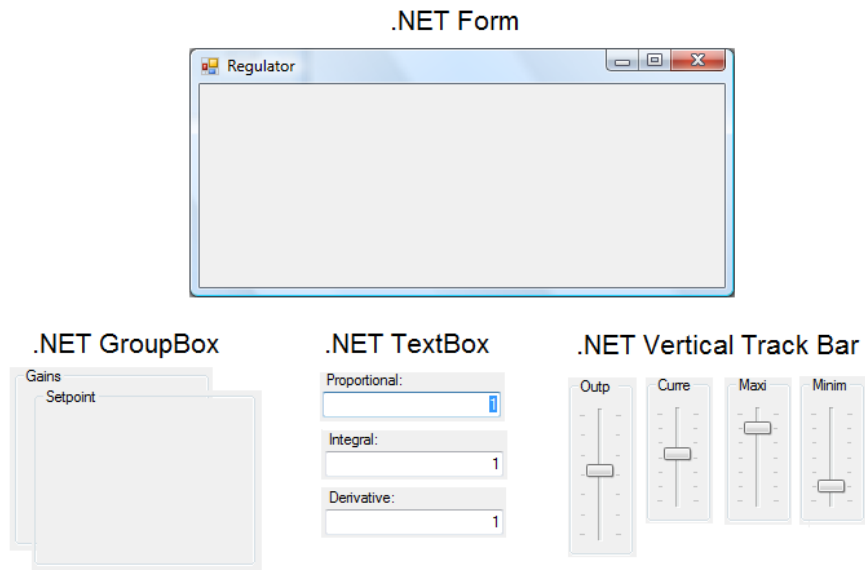Figure 11.8Figure 10.8 shows graphical representation of each CIO from the CIO tree.

Figure 11.8 A graphical representation of the CIOs of the Regulator example

Instantiating all CIOs from the CIO tree in Figure 11.7 creates final user interface that can be seen in Figure 11.9. All CIOs were placed in top-bottom and left-right order which represents highest to lowest importance. The *Gains*, *OutputValue*, and *Setpoint* are of the same importance and thus are placed side by side. Properties of the *Gains* (the *Proportional*, *Integral*, and *Derivative*) have decreasing importance and thus are placed in top-bottom order. The *Current*, *Maximum* and *Minimum* properties have the same importance and thus are side-by-side.
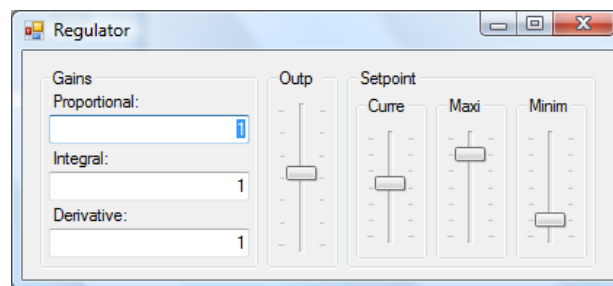
Figure 11.9 The Regulator example user interface

12 Appendix – A Radio Example

This example demonstrates the proposed taxonomy on simple radio example running on desktop environment. The Radio example (see Figure 12.1) consists of the Radio class, a Tuner class, a PresetsCollection and its PresetInfo class.
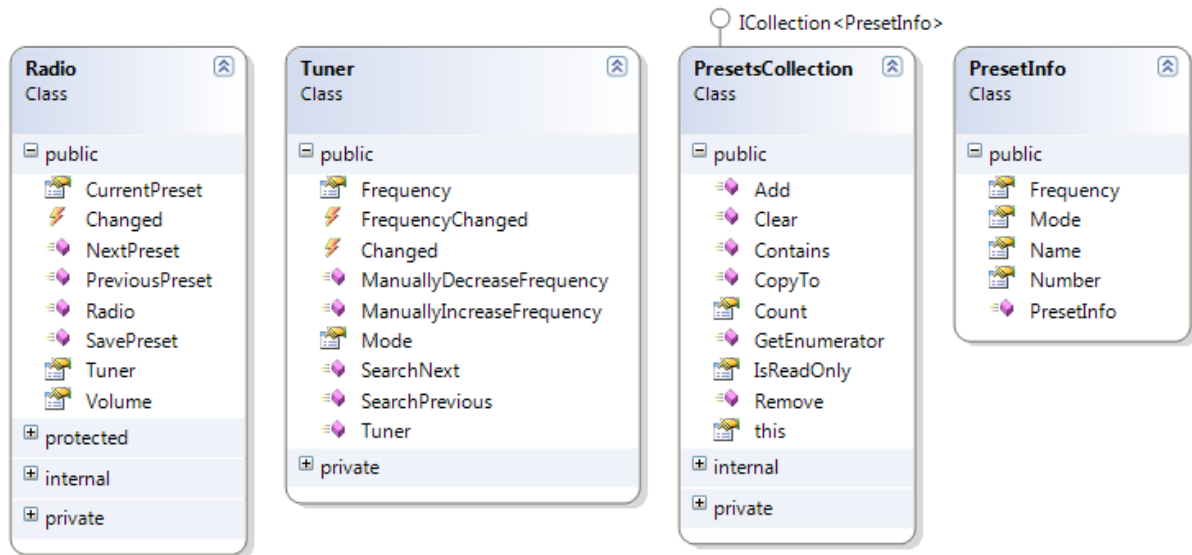

Figure 12.1 The Radio example classes

After the loading of the source code files, information about the characterized classes are loaded into the characterization tree (see Figure 10.5). Notice that only the methods that have been characterized were loaded and are contained in the characterization tree. This means that not every data or method of the classes will play any role in the user interface. In a characterization tree in the Figure 12.5, only few characterization attributes are displayed.

Conversion of the characterization tree to AIOs is demonstrated in the Figure 12.6. The main radio object is represented as a container because it has the *FirstObjectSelected* attribute and is entity, structure. The *CurrentPreset* is represented as a container because it is also entity, structure. The *Number* and *Name* properties of the *CurrentPreset* are represented by a *TextLabel* because both are atomic concepts with label sense, and implementing public getter method only. The *Volume* property is represented by *Volume* because it is atomic measurement of sound intensity. The *SavePreset* method is represented by abstract Command. The *NextPreset* and *PreviousPreset* methods are grouped into *NavigationControls* smart template because both are commands with the *Navigation* category.

```
[DataCharacterization.DataType(DataCharacterization.DataTypes.Structure)]
[DataCharacterization.DataDomain(DataCharacterization.DataDomains.Entity)]
[DataCharacterization.DataForm(DataCharacterization.DataShapes.Shaped)]
[CodeCharacterization.CodeName("Tuning")]
public class Tuner
{
    [DataCharacterization.DataType(DataCharacterization.DataTypes.Atomic)]
    [DataCharacterization.DataDomain(DataCharacterization.DataDomains.Concept)]
    [DataCharacterization.DataForm(Luigi.Characterization.Data.DataShapes.Shaped)]
    [DataCharacterization.DataSense(DataCharacterization.DataSenseTypes.List)]
    [DataCharacterization.DataRole(Luigi.Characterization.Data.DataRoleTypes.Association)]
    [DataCharacterization.DataTransience(DataCharacterization.DataTransienceType.Static)]
    [CodeCharacterization.CodeName("Mode")]
    [DataCharacterization.DataImportance(10)]
    public FrequencyBands Mode { … }

    [DataCharacterization.DataType(DataCharacterization.DataTypes.Atomic)]
    [DataCharacterization.DataDomain(DataCharacterization.DataDomains.Measurement)]
    [DataCharacterization.DataForm(DataCharacterization.DataShapes.None)]
    [DataCharacterization.DataImportance(0)]
 [DataCharacterization.DataScalability(DataCharacterization.DataScalabilityTypes.Amounts)]
    [DataCharacterization.DataSense(DataCharacterization.DataSenseTypes.Symbol)]
    [DataCharacterization.DataTransience(DataCharacterization.DataTransienceType.Dynamic)]
    [CodeCharacterization.CodeName("Frequency")]
    [CodeCharacterization.ParameterRange(88, 108)]
    [CodeCharacterization.ParameterDefaultValue("88")]
    public float Frequency { … }

    [CodeCharacterization.OnChanged]
    public event EventHandler Changed;

    [CodeCharacterization.CodeCategory("Tuner", "Increase")]
    [CodeCharacterization.CodeImportance(0)]
    [CodeCharacterization.CodeSense(CodeCharacterization.CodeSenseTypes.Command)]
    public void ManuallyIncreaseFrequency() { … }

    [CodeCharacterization.CodeCategory("Tuner", "Decrease")]
    [CodeCharacterization.CodeImportance(0)]
    [CodeCharacterization.CodeSense(CodeCharacterization.CodeSenseTypes.Command)]
    public void ManuallyDecreaseFrequency() { … }

    [CodeCharacterization.CodeCategory("Tuner", "SearchNext")]
    [CodeCharacterization.CodeImportance(0)]
    [CodeCharacterization.CodeSense(CodeCharacterization.CodeSenseTypes.Command)]
    public void SearchNext() { … }

    [CodeCharacterization.CodeCategory("Tuner", "SearchPrevious")]
    [CodeCharacterization.CodeImportance(0)]
    [CodeCharacterization.CodeSense(CodeCharacterization.CodeSenseTypes.Command)]
    public void SearchPrevious() { … }
}
```

Figure 12.2 Characterized source code of the Tuner class

```
[DataCharacterization.DataType(DataCharacterization.DataTypes.Structure)]
[DataCharacterization.DataDomain(DataCharacterization.DataDomains.Entity)]
[DataCharacterization.DataTransience(DataCharacterization.DataTransienceType.Static)]
public class PresetInfo
{
    [DataCharacterization.DataType(DataCharacterization.DataTypes.Atomic)]
    [DataCharacterization.DataDomain(DataCharacterization.DataDomains.Concept)]
    [DataCharacterization.DataSense(DataCharacterization.DataSenseTypes.Label)]
    [DataCharacterization.DataImportance(3)]
    [CodeCharacterization.CodeName("Number")]
    public uint Number { … }

    [DataCharacterization.DataType(DataCharacterization.DataTypes.Atomic)]
    [DataCharacterization.DataDomain(DataCharacterization.DataDomains.Concept)]
    [DataCharacterization.DataSense(DataCharacterization.DataSenseTypes.Label)]
    [CodeCharacterization.CodeName("Name")]
    [DataCharacterization.DataImportance(4)]
    public string Name { … }
}
```

Figure 12.3 Characterized source code of the PresetInfo class

137

```
[CodeCharacterization.FirstObjectSelected]
[DataCharacterization.DataType(DataCharacterization.DataTypes.Structure)]
[DataCharacterization.DataDomain(DataCharacterization.DataDomains.Entity)]
[DataCharacterization.DataForm(DataCharacterization.DataShapes.Shaped)]
[CodeCharacterization.CodeName("Radio")]
public class Radio
{
    [DataCharacterization.DataType(DataCharacterization.DataTypes.Atomic)]
    [DataCharacterization.DataDomain(DataCharacterization.DataDomains.Measurement)]
    [DataCharacterization.DataForm(DataCharacterization.DataShapes.None)]
    [DataCharacterization.DataImportance(3)]
 [DataCharacterization.DataScalability(DataCharacterization.DataScalabilityTypes.Amounts)]
    [DataCharacterization.DataSense(DataCharacterization.DataSenseTypes.Symbol)]
    [DataCharacterization.DataTransience(DataCharacterization.DataTransienceType.Static)]
    [CodeCharacterization.CodeName("Volume")]
    [CodeCharacterization.ParameterRange(0, 1)]
    [CodeCharacterization.ParameterDefaultValue("0.5")]
    public float Volume { … }

    [DataCharacterization.DataType(DataCharacterization.DataTypes.Structure)]
    [DataCharacterization.DataDomain(DataCharacterization.DataDomains.Entity)]
    [DataCharacterization.DataForm(DataCharacterization.DataShapes.Shaped)]
    [DataCharacterization.DataImportance(0)]
    [DataCharacterization.DataSense(DataCharacterization.DataSenseTypes.Label)]
    [DataCharacterization.DataTransience(DataCharacterization.DataTransienceType.Static)]
 [DataCharacterization.DataRole(Luigi.Characterization.Data.DataRoleTypes.Identification)]
    [CodeCharacterization.CodeName("Preset")]
    public PresetInfo CurrentPreset { … }

    [CodeCharacterization.CodeName("Tuning")]
    [CodeCharacterization.CodeImportance(2)]
    public Tuner Tuner { … }

    [CodeCharacterization.OnChanged]
    public event EventHandler Changed;

    [CodeCharacterization.CodeCategory("Navigation", "Next")]
    [CodeCharacterization.CodeSense(CodeCharacterization.CodeSenseTypes.Command)]
    [CodeCharacterization.CodeImportance(1)]
    public void NextPreset() { … }

    [CodeCharacterization.CodeCategory("Navigation", "Previous")]
    [CodeCharacterization.CodeSense(CodeCharacterization.CodeSenseTypes.Command)]
    [CodeCharacterization.CodeImportance(1)]
    public void PreviousPreset() { … }

    [CodeCharacterization.CodeName("Save Preset")]
    [CodeCharacterization.CodeSense(CodeCharacterization.CodeSenseTypes.Command)]
    [CodeCharacterization.CodeImportance(1)]
    [CodeCharacterization.Dialog("Saving will replace current preset. Continue?",
CodeCharacterization.DialogQuestion.YesNo)]
    public void SavePreset() { … }
}
```

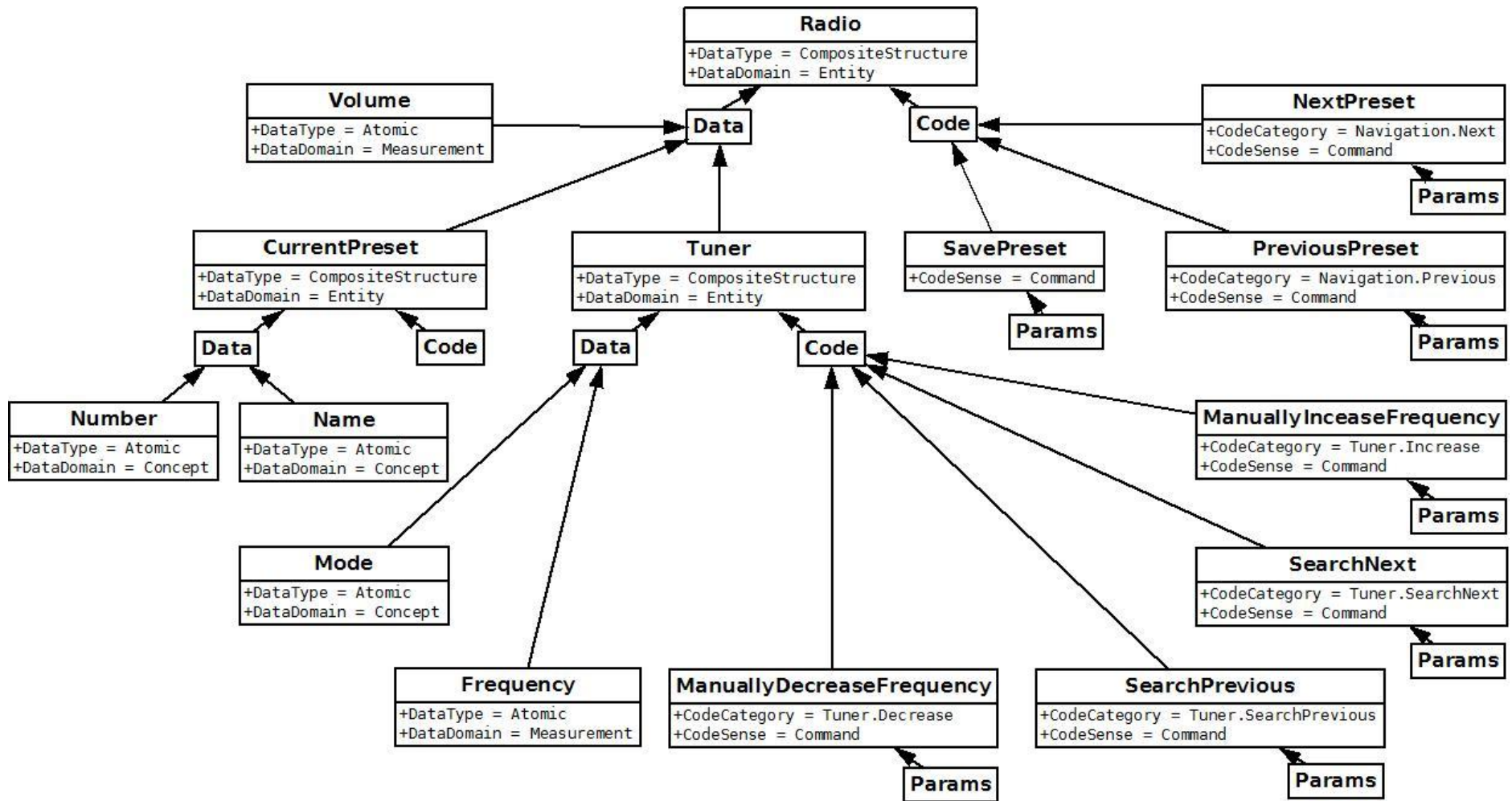**Figure 12.4 Characterized source code of the Radio class**

Figure 12.5 The Radio example characterization tree

The *Tuner* property is represented by a container because it is entity structure. Its property *Frequency* is represented by a *TextLabel* because it is atomic measurement with a label sense. The *Mode* enumeration is represented by *Enumeration* because it is atomic concept enum with list sense. All characterized methods (*ManuallyDecreaseFrequency*, *ManuallyIncreaseFrequency*, *SearchNext*, *SearchPrevious*) are all of *Tuner* category and are represented by *Tuner smart template*.
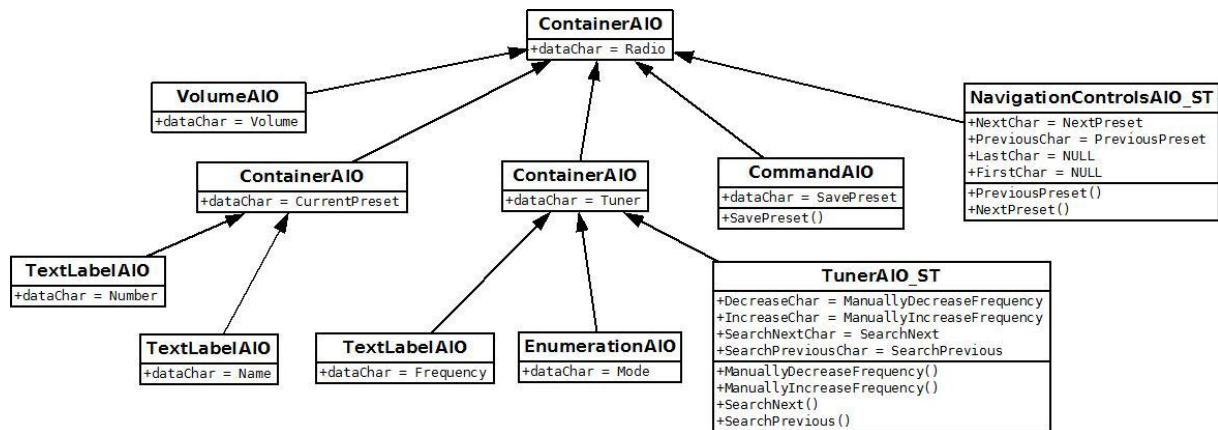
Figure 12.6 A Radio example AIOs

Figure 12.7 demonstrates the result of the AIOs conversion to CIOs. The *Radio* class is represented by a Form (window) because it is the topmost object in the hierarchy. The *Volume* is converted to Horizontal Track Bar because it is the only associated CIO for the AIO. Other implementation would suit the volume abstraction better. The *CurrentPreset* container is converted to GroupBox to reflect its internal structure. Both TextLabels (*Name* and *Frequency*) are converted to labels because both implement getter methods only. The *SavePreset* method is converted to standard button because it is simple command with no parameters and no dependencies. The *NavigationControl*s smart template is converted to a control associated with the AIO, implementing functionality of the navigation tasks.
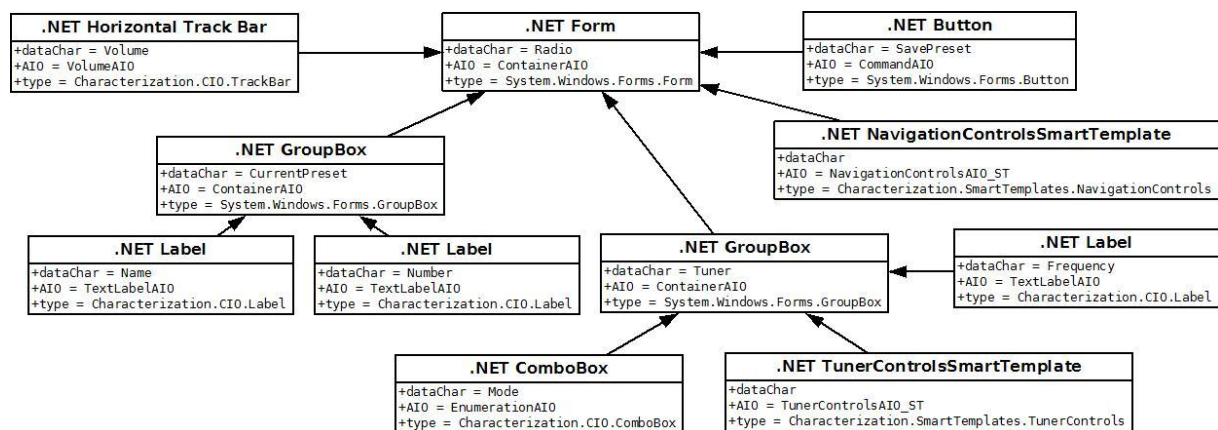
Figure 12.7 The Radio example CIOs

The *Tuner* is converted into group box because it can represent its internal structure. Its *Frequency* property and *Mode* property are converted to label because it implement getter method only and to combo box because it is enumeration, respectively. The *Tuner smart template* is converted to a control associated with the AIO, implementing functionality of the tuning tasks.
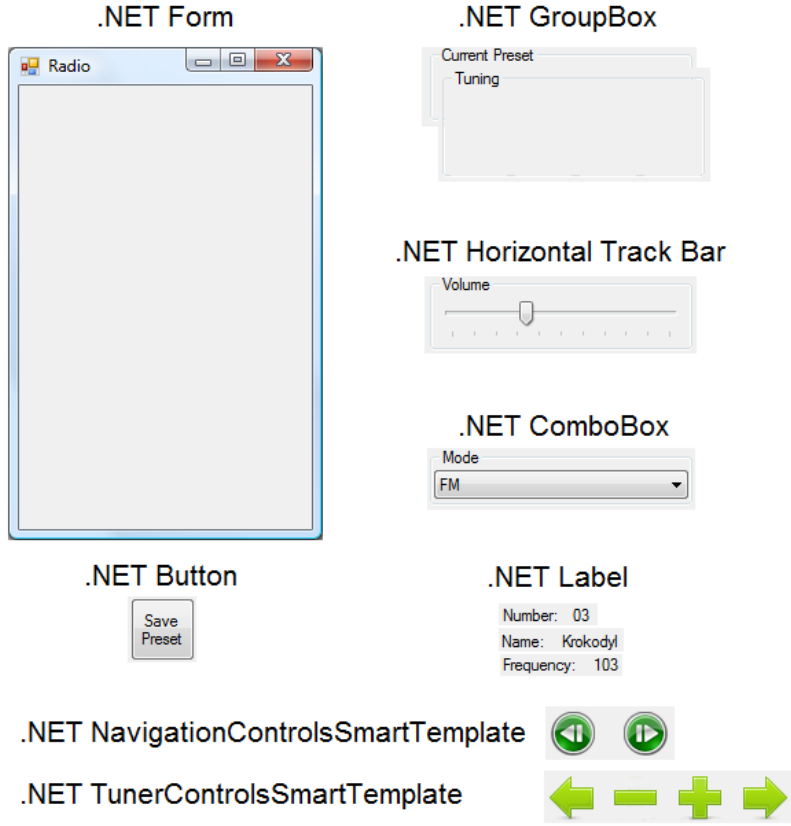

Figure 12.8 A graphical representation of the CIOs of the Radio example

Figure 12.8Figure 10.8 shows graphical representation of each CIO from the CIO tree. Instantiating all CIOs from the CIO tree in Figure 12.7 creates final user interface that can be seen in Figure 12.9. All CIOs were placed in top-bottom and left-right order which represents highest to lowest importance. The "Current Preset" group box is the first because it has the highest importance. Below is the Navigation smart template consisting of two navigation buttons performing preset switching. The Save Preset button is second on the row because it has lower importance than the navigation methods. Below is "Tuning" group box with frequency label (has the highest importance in the group box), the Tuning smart template, and the Mode combo box with the lowest importance in the group box. Last item in the Radio example is the Volume track bar with the lowest importance in whole example.
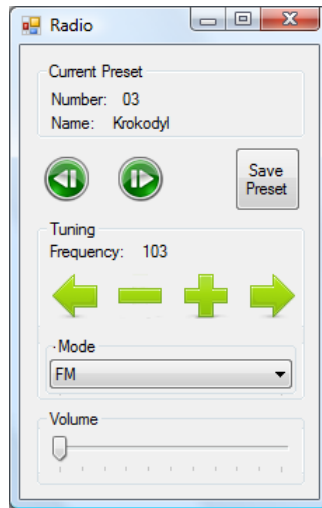
141

Figure 12.9 The Radio example user interface